DataNucleus AccessPlatform
v. 3.3
User Guide
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Table of Contents</td>
<td>i</td>
</tr>
<tr>
<td>2. General</td>
<td>1</td>
</tr>
<tr>
<td>2.1. What's New</td>
<td>2</td>
</tr>
<tr>
<td>2.2. Upgrade Migration</td>
<td>3</td>
</tr>
<tr>
<td>2.3. Getting Started</td>
<td>12</td>
</tr>
<tr>
<td>2.4. Dependencies</td>
<td>14</td>
</tr>
<tr>
<td>2.5. Persistence API Choice</td>
<td>17</td>
</tr>
<tr>
<td>2.6. Development Process</td>
<td>19</td>
</tr>
<tr>
<td>2.7. Compatibility</td>
<td>20</td>
</tr>
<tr>
<td>2.8. Services</td>
<td>22</td>
</tr>
<tr>
<td>2.9. ORM Relationships</td>
<td>23</td>
</tr>
<tr>
<td>2.10. Persistence Properties</td>
<td>26</td>
</tr>
<tr>
<td>2.11. Security</td>
<td>60</td>
</tr>
<tr>
<td>2.12. Logging</td>
<td>62</td>
</tr>
<tr>
<td>3. Datastore</td>
<td>67</td>
</tr>
<tr>
<td>3.1. Supported Features</td>
<td>69</td>
</tr>
<tr>
<td>3.2. RDBMS</td>
<td>74</td>
</tr>
<tr>
<td>3.2.1. Java Types (Spatial)</td>
<td>85</td>
</tr>
<tr>
<td>3.2.2. Datastore Types</td>
<td>91</td>
</tr>
<tr>
<td>3.2.3. Failover</td>
<td>98</td>
</tr>
<tr>
<td>3.2.4. Queries</td>
<td>100</td>
</tr>
<tr>
<td>3.2.5. JDOQL : Spatial Methods</td>
<td>104</td>
</tr>
<tr>
<td>3.2.6. Statement Batching</td>
<td>115</td>
</tr>
<tr>
<td>3.2.7. Views</td>
<td>116</td>
</tr>
<tr>
<td>3.2.8. Datastore API</td>
<td>119</td>
</tr>
<tr>
<td>3.3. ODF</td>
<td>123</td>
</tr>
<tr>
<td>3.4. Excel (XLS)</td>
<td>125</td>
</tr>
<tr>
<td>3.5. Excel (OOXML)</td>
<td>126</td>
</tr>
<tr>
<td>3.6. XML</td>
<td>127</td>
</tr>
<tr>
<td>3.7. HBase</td>
<td>130</td>
</tr>
<tr>
<td>3.8. MongoDB</td>
<td>133</td>
</tr>
<tr>
<td>3.9. Google AppEngine/Datastore</td>
<td>136</td>
</tr>
<tr>
<td>3.10. Neo4j</td>
<td>137</td>
</tr>
<tr>
<td>3.11. JSON</td>
<td>140</td>
</tr>
<tr>
<td>3.12. Amazon S3</td>
<td>142</td>
</tr>
<tr>
<td>3.13. GoogleStorage</td>
<td>143</td>
</tr>
<tr>
<td>3.14. LDAP</td>
<td>144</td>
</tr>
<tr>
<td>3.14.1. Relations by DN</td>
<td>148</td>
</tr>
</tbody>
</table>
## Table of Contents

3.14.2. Relations by Attribute ................................................. 152  
3.14.3. Relations by Hierarchy ............................................. 157  
3.14.4. Embedded Objects ................................................... 162  
3.15. NeoDatis ................................................................ 164  
4. JDO API ..................................................................... 167  
  4.1. Class Mapping .............................................................. 169  
  4.2. Datastore Identity .................................................... 171  
  4.3. Application Identity ................................................... 174  
  4.4. Nondurable Identity ................................................... 180  
  4.5. Compound Identity ..................................................... 181  
  4.6. Versioning ............................................................... 191  
  4.7. Inheritance ............................................................... 193  
  4.8. Fields/Properties ....................................................... 206  
    4.8.1. Java Types .......................................................... 209  
    4.8.2. Value Generation ............................................... 217  
    4.8.3. Sequences ......................................................... 230  
    4.8.4. Embedded Fields ............................................... 234  
    4.8.5. Serialised Fields .............................................. 247  
    4.8.6. Interface Fields ................................................. 255  
    4.8.7. Object Fields ................................................... 259  
    4.8.8. Array Fields ..................................................... 262  
    4.8.9. 1-to-1 Relations .................................................. 267  
    4.8.10. 1-to-N Relations ................................................. 271  
      4.8.10.1. Collections ............................................... 272  
      4.8.10.2. Sets .......................................................... 285  
      4.8.10.3. Lists ......................................................... 295  
      4.8.10.4. Maps ........................................................ 306  
    4.8.11. N-to-1 Relations .................................................. 315  
    4.8.12. M-to-N Relations ................................................ 317  
    4.8.13. Managing Relationships ...................................... 324  
4.9. MetaData Reference ..................................................... 331  
  4.9.1. XML ................................................................. 333  
  4.9.2. Annotations ....................................................... 369  
  4.9.3. MetaData API ..................................................... 402  
  4.9.4. ORM MetaData .................................................... 404  
4.10. Schema Mapping ....................................................... 406  
  4.10.1. Multitenancy ...................................................... 413  
  4.10.2. Datastore Identifiers ............................................ 414  
  4.10.3. Secondary Tables ............................................... 417
## Table of Contents

4..11. Constraints ................................................................. 422
4..12. Enhancer .................................................................. 427
4..13. Datastore Schema .................................................. 438
4..14. Bean Validation ......................................................... 448
4..15. API Javadocs ............................................................... 449
4..16. PersistenceManagerFactory .................................... 466
   4..16.1. L2 Cache .............................................................. 455
   4..16.2. Auto-Start .......................................................... 462
   4..16.3. Data Federation .................................................. 465
4..17. PersistenceManager .................................................. 470
   4..17.1. PM Proxy ............................................................ 471
   4..17.2. Object Lifecycle .................................................. 475
   4..17.3. Lifecycle Callbacks ............................................... 482
   4..17.4. Attach/Detach ..................................................... 489
4..18. Transactions ............................................................... 500
4..19. Fetch Groups ............................................................... 509
4..20. Query API ................................................................. 514
   4..20.1. Query Cache ......................................................... 520
   4..20.2. JDOQL ................................................................. 522
   4..20.3. JDOQL Declarative .............................................. 540
   4..20.4. JDOQL Typesafe .................................................. 546
   4..20.5. SQL ................................................................. 552
   4..20.6. Stored Procedures .............................................. 558
   4..20.7. JPQL ............................................................... 560
4..21. Guides ................................................................. 572
   4..21.1. Datastore Replication ......................................... 573
   4..21.2. JEE Environments .............................................. 577
   4..21.3. OSGi Environments ............................................ 586
   4..21.4. Troubleshooting ............................................... 603
   4..21.5. Performance Tuning ............................................ 608
   4..21.6. Monitoring ....................................................... 616
   4..21.7. Maven with DataNucleus .................................... 618
   4..21.8. Eclipse with DataNucleus ................................... 621
   4..21.9. Tutorial with RDBMS ......................................... 629
   4..21.10. Tutorial with ODF ........................................... 641
   4..21.11. Tutorial with Excel ........................................... 653
   4..21.12. Tutorial with MongoDB ..................................... 665
   4..21.13. Tutorial with HBase .......................................... 677
   4..21.14. Tutorial with Neo4j .......................................... 689
Table of Contents

4.21..15. 1-N Bidir FK Relation .............................................. 699
4.21..16. 1-N Bidir Join Relation ................................. 706
4.21..17. M-N Relation .......................................................... 717
4.21..18. M-N Attributed Relation ........................................ 721
4.21..19. Spatial Types Tutorial ........................................... 726
4.21..20. DAO Layer Design .................................................. 734

5. JPA API ........................................................................... 735
5.1. Class Mapping ............................................................... 737
5.2. Application Identity ...................................................... 742
5.3. Datastore Identity .......................................................... 744
5.4. Compound Identity ........................................................ 754
5.5. Versioning ................................................................. 756
5.6. Inheritance ................................................................. 766
5.7. Fields/Properties ............................................................ 769
  5.7..1. Java Types ................................................................. 777
  5.7..2. Value Generation ...................................................... 783
  5.7..3. Embedded Fields ..................................................... 791
  5.7..4. Serialised Fields ..................................................... 794
  5.7..5. Interface Fields ...................................................... 799
  5.7..6. Object Fields ......................................................... 802
  5.7..7. Array Fields ............................................................. 804
  5.7..8. 1-to-1 Relations ...................................................... 808
  5.7..9. 1-to-N Relations ...................................................... 819
    5.7..9..1. Collections ..................................................... 829
    5.7..9..2. Sets ................................................................. 839
    5.7..9..3. Lists .............................................................. 845
    5.7..9..4. Maps .............................................................. 848
  5.7..10. N-to-1 Relations .................................................. 852
  5.7..11. M-to-N Relations .................................................. 854
  5.7..12. Managing Relationships ........................................ 857
  5.7..13. Cascading ............................................................ 882
5.8. MetaData Reference ..................................................... 888
  5.8..1. XML .................................................................. 890
  5.8..2. Annotations ............................................................. 892
5.9. Schema Mapping .......................................................... 923
  5.9..1. Multitenancy ............................................................ 926
  5.9..2. Datastore Identifiers .............................................. 927
  5.9..3. Secondary Tables .................................................. 929
5.10. Constraints ................................................................. 931
5.11. Enhancer ................................................................. 935

© 2015, DataNucleus • ALL RIGHTS RESERVED.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.12. Datastore Schema</td>
<td>944</td>
</tr>
<tr>
<td>5.13. Bean Validation</td>
<td>953</td>
</tr>
<tr>
<td>5.14. API Javadocs</td>
<td>954</td>
</tr>
<tr>
<td>5.15. EntityManagerFactory</td>
<td>960</td>
</tr>
<tr>
<td>5.15.1. L2 Cache</td>
<td>966</td>
</tr>
<tr>
<td>5.16. Entity Manager</td>
<td>970</td>
</tr>
<tr>
<td>5.16.1. Object Lifecycle</td>
<td>972</td>
</tr>
<tr>
<td>5.16.2. Lifecycle Callbacks</td>
<td>974</td>
</tr>
<tr>
<td>5.16.3. Datastore Connection</td>
<td>983</td>
</tr>
<tr>
<td>5.17. Transactions</td>
<td>991</td>
</tr>
<tr>
<td>5.18. Entity Graphs</td>
<td>993</td>
</tr>
<tr>
<td>5.19. Query API</td>
<td>996</td>
</tr>
<tr>
<td>5.19.1. Query Cache</td>
<td>998</td>
</tr>
<tr>
<td>5.19.2. JPQL</td>
<td>1010</td>
</tr>
<tr>
<td>5.19.3. JPQL Criteria</td>
<td>1016</td>
</tr>
<tr>
<td>5.19.4. SQL</td>
<td>1018</td>
</tr>
<tr>
<td>5.19.5. Stored Procedures</td>
<td>1020</td>
</tr>
<tr>
<td>5.20. Guides</td>
<td>1021</td>
</tr>
<tr>
<td>5.20.1. Datastore Replication</td>
<td>1022</td>
</tr>
<tr>
<td>5.20.2. JavaEE Environments</td>
<td>1028</td>
</tr>
<tr>
<td>5.20.3. OSGi Environments</td>
<td>1031</td>
</tr>
<tr>
<td>5.20.4. Performance Tuning</td>
<td>1038</td>
</tr>
<tr>
<td>5.20.5. Troubleshooting</td>
<td>1043</td>
</tr>
<tr>
<td>5.20.6. Monitoring</td>
<td>1045</td>
</tr>
<tr>
<td>5.20.7. Maven with DataNucleus</td>
<td>1049</td>
</tr>
<tr>
<td>5.20.8. Eclipse with DataNucleus</td>
<td>1056</td>
</tr>
<tr>
<td>5.20.9. Tutorial with RDBMS</td>
<td>1068</td>
</tr>
<tr>
<td>5.20.10. Tutorial with ODF</td>
<td>1080</td>
</tr>
<tr>
<td>5.20.11. Tutorial with Excel</td>
<td>1092</td>
</tr>
<tr>
<td>5.20.12. Tutorial with MongoDB</td>
<td>1104</td>
</tr>
<tr>
<td>5.20.13. Tutorial with HBase</td>
<td>1116</td>
</tr>
<tr>
<td>5.20.14. Tutorial with Neo4j</td>
<td>1126</td>
</tr>
<tr>
<td>5.20.15. Eclipse Dali</td>
<td>1130</td>
</tr>
<tr>
<td>5.20.16. JPA Tutorial (TheServerSide)</td>
<td>1136</td>
</tr>
<tr>
<td>6. REST API</td>
<td></td>
</tr>
<tr>
<td>7. Guides</td>
<td></td>
</tr>
<tr>
<td>7.1. Use with IDEA</td>
<td></td>
</tr>
<tr>
<td>7.2. Use with Netbeans</td>
<td></td>
</tr>
<tr>
<td>7.3. Use with Ivy</td>
<td></td>
</tr>
<tr>
<td>7.4. Use with OSGi+SpringDM</td>
<td></td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>7.5</td>
<td>Enhancing with Ant</td>
</tr>
<tr>
<td>7.6</td>
<td>Jobfilter - JDO/JPA web app</td>
</tr>
</tbody>
</table>
1 General

1.1 DataNucleus AccessPlatform 3.3

DataNucleus AccessPlatform v3.3 provides persistence and retrieval of Java objects to a range of datastores using JDO/ JPA/ REST APIs, with a range of query languages and is fully-compliant with JDO and JPA specifications. It is Apache 2 licensed. No other persistence solution offers the same range of APIs, datastores and query languages whilst also being fully compliant.

DataNucleus AccessPlatform 3.3 Checklist

- MetaData/Mapping Supported: JDO, JPA
- Datastores Supported: RDBMS, Excel, OOXML, ODF, XML, HBase, MongoDB, AppEngine/DataStore, Neo4j, JSON, Amazon S3, GoogleStorage, LDAP, NeoDatis
- JRE required: 1.6 or above
- Specifications: JDO3.1, JPA2.1

- Beginners: The first thing to do is to visit the Getting Started Guide
- Migrating from older version: please read this first about how to upgrade.

If you find something that DataNucleus Access Platform can't handle you can always extend it using its plugin mechanism for one of its defined interfaces. Just look for the icon.
2.1 AccessPlatform : What's New in 3.3

DataNucleus AccessPlatform version 3.3 extends the 3.2 capabilities, particularly for JPA. Below are some of the new features you can find in DataNucleus AccessPlatform 3.3.

- JPA: support for JPA2.1 FROM "ON" clauses in string-based and Criteria queries
- JPA: support for JPA2.1 UPDATE/DELETE Criteria queries
- JPA: support for JPA2.1 ForeignKey, and Index
- JPA: support for JPA2.1 JPQL "FUNCTION"
- JPA: support for JPA2.1 EntityGraphs
- JPA: support for JPA 2.1 create/drop/load scripts
- JPA: runs against a JPA 2.1 API jar
- JPA: support for multifield JPQL FROM join clause syntax
- JDO: support for create/drop/load scripts
- JDOQL/JPQL: support for ordering directives for NULL placement
- RDBMS: support for fetch plan handling in queries when the candidate uses complete-table inheritance
- RDBMS: support for persistence of File members, using streamable mechanism
- RDBMS: support for bulk-fetch of collection members that are in the FetchPlan of a candidate in a query
- RDBMS: initial support for NuoDB
- MongoDB: support for evaluation of querying order clause in-datastore
- Neo4j: support for persisting maps of persistable objects
# 3 Upgrade Migration

## 3.1 AccessPlatform: Migration between versions

This version of DataNucleus AccessPlatform builds on the 3.2 releases, and adds JPA 2.1. Some internal APIs could be changed from 3.1/3.2. All releases are checked regularly against the JDO/JPA TCKs, meaning that DataNucleus is always stable in terms of functionality. Occasionally, due to unknown bugs, or due to new functionality being introduced we need to change some aspects of DataNucleus. As a result sometimes users will have to make some changes to move between versions of DataNucleus. We aim to keep this to a minimum.

### 3.1.1 Migration from 3.3.6 to 3.3.7

Migrating will require the following changes
- Persistence property `datanucleus.jpa.findTypeConversion` is now removed and replaced with `datanucleus.findObject.typeConversion`, defaulting to `true`.

### 3.1.2 Migration from 3.3.5 to 3.3.6

Migrating will require the following changes
- The `spatial` and `awtgeom` plugins have been merged, to be `datanucleus-geospatial`.

### 3.1.3 Migration from 3.3.4 to 3.3.5

Migrating will require the following changes
- RDBMS: where you have a query that has a collection member in the FetchPlan it previously would have been ignored. Now it is used to attempt a bulk-fetch of the collection. Since this is new functionality there may be cases where the syntax is not optimal; remove the collection field from the query FetchPlan to get the previous behaviour.

### 3.1.4 Migration from 3.3.3 to 3.3.4

Migrating will require the following changes
- RDBMS: default mapping for Boolean/boolean java types is now JDBC type `BOOLEAN` for H2 database; previously this was unspecified so most likely fell back to CHAR for that database. Specify the jdbc-type explicitly if you want to have CHAR.

### 3.1.5 Migration from 3.3.2 to 3.3.3

Migrating from AccessPlatform 3.3.2 to 3.3.3 will require the following changes
- `datanucleus-googlecollections` plugin is now renamed to `datanucleus-guava`.

### 3.1.6 Migration from 3.3.1 to 3.3.2

Migrating will require no changes except to internal API(s).
3.1.7 Migration from 3.3.0.RELEASE to 3.3.1
Migrating will require no changes except to internal API(s).

3.1.8 Migration from 3.3.0.M1 to 3.3.0.RELEASE
Migrating will require the following changes
- DataNucleus @FetchGroup extension annotation for JPA is now dropped and people should use the official JPA 2.1 @NamedEntityGraph annotation instead (or XML equivalent of course)

3.1.9 Migration from 3.2.3 to 3.3.0.M1
Migrating will require the following changes
- Now requires a compliant JPA 2.1 API jar. An official JPA 2.1 API jar is not yet available, but as a stopgap there is a Eclipse javax.persistence v2.1.0 jar. If using the Maven plugin with JPA, note that you also require v3.3.0.m1 of that plugin
- DataNucleus @Index extension annotation for JPA is now dropped and people should use the official JPA 2.1 @Index annotation instead (or XML equivalent of course)

3.1.10 Migration from 3.2.8 to 3.2.9
Migrating will require the following changes
- RDBMS : where you have a query that has a collection member in the FetchPlan it previously would have been ignored. Now it is used to attempt a bulk-fetch of the collection. Since this is new functionality there may be cases where the syntax is not optimal; remove the collection field from the query FetchPlan to get the previous behaviour.

3.1.11 Migration from 3.2.7 to 3.2.8
Migrating will require the following changes
- RDBMS : default mapping for Boolean/boolean java types is now JDBC type BOOLEAN for H2 database; previously this was unspecified so most likely fell back to CHAR for that database. Specify the jdbc-type explicitly if you want to have CHAR

3.1.12 Migration from 3.2.6 to 3.2.7
Migrating from AccessPlatform 3.2.6 to 3.2.7 will require the following changes
- datanucleus-googlecollections plugin is now renamed to datanucleus-guava

3.1.13 Migration from 3.2.2 to 3.2.3
Migrating will require the following changes
- The persistence property datanucleus.metadata.validate is renamed to datanucleus.metadata.xml.validate to better describe its usage. The original name is still
supported but you are advised to move to this new naming as the old one can be removed in a future release.

3.1.14 Migration from 3.2.1 to 3.2.2
Migrating will require no changes.

3.1.15 Migration from 3.2.0.RELEASE to 3.2.1
Migrating will require the following changes

- The persistence property `datanucleus.attachSameDatastore` defaults to `true` with `datanucleus-core` version 3.2.1 and later. Set it to `false` if you require replicating objects into other datastores.
- The JDOQL method `Date.getDay` is now deprecated and `Date.getDate` should be used instead (day of the month). `Date.getDay` is likely to be converted to return the day of the week in a later release, so fixing any use of this now makes sense.
- PreparedStatement pooling is turned now turned OFF by default due to the fact that DBCP has a bug where it isn't closing ResultSets correctly when this is enabled.

3.1.16 Migration from 3.2.0.M4 to 3.2.0.RELEASE
Migrating will require no changes.

3.1.17 Migration from 3.2.0.M3 to 3.2.0.M4
Migrating will require the following changes.

- The RDBMS persistence property `datanucleus.rdbms.sqlParamValuesInBrackets` is now removed, and replaced by `datanucleus.rdbms.statementLogging` (see the docs).
- The persistence property `datanucleus.rdbms.useUpdateLock` is now removed (was deprecated many releases back). Use standard JDO/JPA locking mechanisms instead.
- Any user-defined RDBMS mapping plugins will need updating to match some minor type changes to the “datanucleus-rdbms” plugin API.

3.1.18 Migration from 3.2.0.M2 to 3.2.0.M3
Migrating will require no changes.

3.1.19 Migration from 3.2.0.M1 to 3.2.0.M2
Migrating will require the following changes.

- The Maven plugin has been renamed to `datanucleus-maven-plugin` from `maven-datanucleus-plugin` to match Maven3 naming policies.
- You no longer require to include `asm.jar` since version 4.1 of ASM is now repackaged into `datanucleus-core.jar`
• Added persistence property "datanucleus.useImplementationCreator" to allow turning off the persistent interface implementation creator.
• All java type mappings used by the RDBMS plugin are now moved from org.datanucleus.store.mapped.mapping in the core plugin, to org.datanucleus.store.rdbms.mapping.java in the RDBMS plugin. Related classes only for "mapped" datastores are also now in the RDBMS plugin

3.1.20 Migration from 3.1.x to 3.2.0.M1
Migrating will require the following changes.
• The Enhancer plugin is now merged into "datanucleus-core". Note also that the "pre-compilation" enhancement process is now discontinued.
• The Enhancer Ant task is now moved to org.datanucleus.enhancer.EnhancerTask
• Various DataNucleus internal classes have been refactored. Please refer to the DataNucleus Wiki for details of upgrading DataNucleus internal API calls
• Many "simple" Java field types now default to persistent (all supported types are now set to default persistent). Additionally many "simple" types default to being in the DFG whereas they used not to (i.e you had to enable the persistence of them, e.g java.sql.Date)

3.1.21 Migration from 3.1.1 to 3.1.2
Migrating will require no changes.

3.1.22 Migration from 3.1.0.RELEASE to 3.1.1
Migrating will require no changes.

3.1.23 Migration from 3.1.0.M5 to 3.1.0.RELEASE
Migrating will require the following changes.
• You no longer are required to specify the persistence property datanucleus.rdbms.stringDefaultLength as 255 for JDO; this is its new default

3.1.24 Migration from 3.1.0.M4 to 3.1.0.M5
Migrating will require no changes.

3.1.25 Migration from 3.1.0.M3 to 3.1.0.M4
Migrating will require the following changes.
• The enhancer (v3.1) is now upgraded and requires ASM v4.0+. You can continue to use the v3.0 enhancer with ASM v3.x but that will not work completely with JDK1.7
• The RDBMS plugin now requires JDK1.6+ to run. Use v3.0 if you are still using JDK1.5
3.1.26 Migration from 3.1.0.M2 to 3.1.0.M3

Migrating will require the following changes.

- Persistence property `datanucleus.managedRuntime` replaced by `datanucleus.jmxType` defining the JMX server to use.
- Persistence property `datanucleus.datastoreTransactionDelayOperations` is removed and replaced by `datanucleus.flush.mode` with values of MANUAL and AUTO. MANUAL means that operations will only go to the datastore on flush/commit, whereas AUTO will send them immediately.
- The persistence property `datanucleus.nontx.atomic` previously only included persists and deletes. It now also encompasses field updates. Bear this in mind when considering behaviour.
- The value strategy chosen when "native"(JDO)/"auto"(JPA) is specified has changed. It will now take "identity"/"sequence"/"increment" when numeric-based (first that is supported for that datastore) and "uuid-hex" when string-based. For RDBMS, use persistence property `datanucleus.rdbms.useLegacyNativeValueStrategy` as `true` if wanting the old process.

3.1.27 Migration from 3.1.0.M1 to 3.1.0.M2

Migrating will require the following changes.

- "javax.cache" is now split into "jcache" (old API) and "javax.cache" (standard API) and the standard API is now supported in `datanucleus-core`
- `datanucleus-management` plugin is now merged into `datanucleus-core`

3.1.28 Migration from 3.0.x to 3.1.0.M1

Migrating will require the following changes.

- Excel, ODF, MongoDB and HBase plugins now respect JDO/JPA table/column naming strategies. Make sure that you set the table/column names explicitly if requiring some other naming that was default with v3.0 and earlier plugins.
- If you have any "type" plugins using the ObjectStringConverter or ObjectLongConverter interface please rewrite them to use the new TypeConverter interface (minimal changes).

3.1.29 Migration from 3.0.3 to 3.0.4

Migrating will require the following changes.

- Move java.awt geometric type support into `datanucleus-awtgeom` plugin

3.1.30 Migration from 3.0.2 to 3.0.3

Migrating will require no changes.

3.1.31 Migration from 3.0.1 to 3.0.2

Migrating will require the following changes.
• HBase: Default behaviour was to use Java serialisation to get the bytes of the PK of objects. This is changed to now use HBase Bytes.toBytes resulting in cleaner PK ROW ID. To get the old behaviour set the persistence property `datanucleus.hbase.serialisePK`.

• HBase: default behaviour used to be to persist primitive wrapper fields as serialized. They are now persisted as serialised if specified in metadata, otherwise using HBase Bytes handler.

3.1.32 Migration from 3.0.0 M6 to 3.0.0 RELEASE
Migrating will require no changes.

3.1.33 Migration from 3.0.0 M5 to 3.0.0 M6
Migrating will require the following changes.

• The plugin attribute "override" utilised by "java_type", "store_mapping" and "rdbms_mapping" is now removed, and users should make use of the attribute "priority" (specify a number and the higher the number the higher the priority that plugin extension gets).

• JPA usage now defaults to use "datanucleus.RetainValues". This means that when an object leaves a transaction it will not move to HOLLOW state, but instead to PERSISTENT NONTRANSACTIONAL and has its field values intact.

• If using an identity string translator, note that this is now a IdentityStringTranslator and the persistence property is now "datanucleus.identityStringTranslatorType".

3.1.34 Migration from 3.0.0 M4 to 3.0.0 M5
Migrating should require no changes.

3.1.35 Migration from 3.0.0 M3 to 3.0.0 M4
Migrating will require the following changes.

• Maven2 plugin option "outputFile" is renamed to "ddlFile" for consistency with all docs/tools.

3.1.36 Migration from 3.0.0 M2 to 3.0.0 M3
Migrating will require the following changes.

• Anyone using "memcache" cache provider should rename it to "spymemcached". This renaming is to clarify which implementation of "memcached" is actually being used. Similarly the persistence properties are now spelt "memcached" instead of "memcache". Also the former property `datanucleus.cache.level2.memcached.keyprefix` is dropped and users should use `datanucleus.cache.level2.cacheName` instead.

• HBase: previously all primitives were stored serialised. Set the metadata 'serialized' flag on the field/property to continue doing that.

• Queries are no longer run in a separate thread (which was the previous way of supporting query cancellation, now reworked for RDBMS to use SQL error codes).

• Persistence properties for schema validation `datanucleus.validateXXX` now default to false.
3.1.37 Migration from 3.0.0 M1 to 3.0.0 M2

Migrating will require the following changes.

- The connection password decryption interface has been repackaged/renamed to `org.datanucleus.store.encryption.ConnectionEncryptionProvider` so if you were providing your own decryption of passwords then rebuild to this.
- If using your own DataNucleus plugins, make sure you specify the persistence property `datanucleus.plugin.allowUserBundles` as `true` since the default is now to just use official DataNucleus plugins.
- The identifier naming strategy `datanucleus` has been renamed to `datanucleus1` to make it clearer that it was used as the default for DataNucleus v1.x but no longer.

3.1.38 Migration from 2.2.x to 3.0.0 M1

Migrating will require the following changes.

- JDO API has been moved into its own plugin "datanucleus-api-jdo" and you will need this if using the JDO API. JDO classes have been repackaged to `org.datanucleus.api.jdo` and this is of particular importance for your PMF class (`org.datanucleus.api.jdo.JDOPersistenceManagerFactory`).
- "datanucleus-jpa" jar has been repackaged as "datanucleus-api-jpa" and the classes within repackaged to "org.datanucleus.api.jpa". In particular your JPA persistence provider class should reference this new package name (`org.datanucleus.api.jpa.PersistenceProviderImpl`).
- "datanucleus-rest" jar has been repackaged as "datanucleus-api-rest".
- SchemaTool (and its Ant task) has been moved in package to `org.datanucleus.store.schema`.
- HBase: generation of "family name" has changed when previously specifying a column name without a colon; previously used that as family name and qualifier name, but now uses the table name as the family name in that situation.
- HBase: previously all relationships were stored serialised. Set the metadata 'serialized' flag on the field/property to continue doing that.

3.1.39 Migration from 2.2.0 RELEASE to 2.2.1

Migrating will require the following changes.

- JDO 3.1 changes the return type of JDOQL "AVG" to be double or BigDecimal depending on the type being averaged (previously just returned the same type as the averaged type).

3.1.40 Migration from 2.2.0 Milestone3 to 2.2.0 RELEASE

Migrating will require the following changes.

- `datanucleus-connectionpool` is no longer provided/needed, and is included within `datanucleus-rdbms`. In addition, if using JDK1.6 you can use a builtin DBCP connection pool. You still need to include the relevant connection pool (e.g DBCP) in your CLASSPATH if using JDK1.5.
- If you experience different behaviour with delete of objects with Excel or ODF, this is because they now support cascade-delete.
• Major changes have been made to the use of the L2 cache (so that fields are used from there rather than from the datastore wherever possible) and also to Managed Relations. Please report any problems

3.1.41 Migration from 2.2.0 Milestone2 to 2.2.0 Milestone3
Migrating will require the following changes.

• Persistence property `datanucleus.attachPolicy` was removed since no longer needed - the default attach handler copes with all situations.
• Much improved support for collections/arrays/maps containing nulls is now present to better match the Java spec for types. If any problems come up, make use of the "allow-nulls" extension metadata
• JPA Criteria query annotation processor is now in its own plugin jar known as `datanucleus-jpa-query`
• JDO Typesafe query annotation processor is now in its own plugin jar known as `datanucleus-jdo-query`

3.1.42 Migration from 2.2.0 Milestone1 to 2.2.0 Milestone2
Migrating will require the following changes.

• NucleusJDOHelper methods for getting dirty/loaded fields have been improved. Check the docs for the new method names.
• JDO3.1 sequence changes allow specification of "allocationSize" and "initialValue". These default to 50 and 1 respectively. Set them for your sequences as required. The persistence properties now become only fallback values

3.1.43 Migration from 2.1.x to 2.2.0 Milestone1
Migrating will require the following changes.

• Legacy JDOQL implementation for RDBMS is now dropped. Use AccessPlatform 2.1 if you require it

3.1.44 Migration from 2.1.2 to 2.1.3
Migrating will require the following changes.

• Persistence property `datanucleus.attachPolicy` is now removed, and the default handling should work fine

3.1.45 Migration from 2.1.1 to 2.1.2
Migrating will require the following changes.

• The metadata extension `index` that is used to specify a column position (in table) was previously required under "field" for Excel/ODF plugins. It should be under "column" now
3.1.46 Migration from 2.1.0 RELEASE to 2.1.1

Migrating will require the following changes.

- Default allocation size for `increment` and `sequence` value strategies have been changed for JDO usage to 10 and 10 respectively (from 5 and 1). You can configure the global defaults via persistence properties.

3.1.47 Migration from 2.1.0 Milestone3 to 2.1.0 RELEASE

Migrating will require the following changes.

- Move to using JDO3 jar instead of JDO 2.3 "ee"
- Dropped support for class-level metadata extension "cacheable"; use standardised `cacheable` attribute (or annotation) instead.

3.1.48 Migration from 2.1.0 Milestone2 to 2.1.0 Milestone3

Migrating will require no changes.

3.1.49 Migration from 2.1.0 Milestone1 to 2.1.0 Milestone2

Migrating will require the following changes.

- JPQL "CASE" statements are now supported
- JPA2 static metamodel is now supported, and so can be used with criteria queries alongside the string-based field specification method
- Runtime enhancement is now turned off by default even when you use JDK1.6+ and have the enhancer/core jars in the CLASSPATH. Specify the compiler argument `processor` to enable it (see docs)

3.1.50 Migration from 2.0.x to 2.1.0 Milestone1

Migrating will require the following changes.

- The JDOQL implementation used for RDBMS is now the rewritten "generic" implementation. To use the old implementation, set the JDOQL implementation as "JDOQL-Legacy"
- Use of JPA should be run against the JPA2 "final" jar (or its Apache Geronimo specs equivalent)
- Heavy refactoring has been done internally so if relying on DataNucleus APIs you should check against SVN for changes. In particular, plugins should be using ObjectProvider instead of StateManager, and ExecutionContext in place of ObjectManager.
4 Getting Started

4.1 AccessPlatform: Getting Started

DataNucleus AccessPlatform implements the JDO and JPA specifications. These specifications define how Java classes can be persisted to a datastore and how they can be queried. By choosing AccessPlatform you can select which of these APIs you feel most comfortable with. Time for you to get started and use AccessPlatform!

4.1.1 What is required?

1. Decide which datastore your project will use, and then download DataNucleus AccessPlatform
2. Depending on which ZIP you downloaded above, and what add-ons you’ll be using you may also need to download some dependencies

You now have the necessary components to start investigating use of DataNucleus.

4.1.2 Starting up

Decide which persistence API you want to use. If you’re not familiar with these APIs then the next thing to do is to learn about JDO and JPA, or alternatively REST. You need to understand the basic concepts involved. There is plenty of reading on the internet, starting with the JDO or JPA specifications of course.

The best thing to do after some reading is to try the JDO Tutorial (for RDBMS, HBase, MongoDB, Neo4j, ODF, Excel) or try the JPA Tutorial (for RDBMS, HBase, MongoDB, Neo4j, ODF, Excel). These explain the basic steps of applying JDO/JPA (and DataNucleus) to your own application. The source code from the tutorials is available for download. Please download it and start up your development environment with the Tutorial classes and files. Once you have completed the Tutorial you’re in a position to apply DataNucleus JDO/JPA to your own application, and start benefiting from what it offers.

4.1.3 Key Points

There are some key points to bear in mind when starting using JDO/JPA for Java persistence.

- To persist objects of classes you firstly need to define which classes are persistable, and how they are persisted. Start under the JDO Class Mapping and JPA Class Mapping sections
- Use of JDO or JPA requires a datastore-controlling factory: `PersistenceManagerFactory` for JDO, `EntityManagerFactory` for JPA. You can define many properties to define the capabilities of this.
- The persistence of objects is controlled by an API. Look under JDO API and JPA API for more details.
- During the persistence process objects are in different lifecycle states (JDO, JPA) and you ought to be aware of what they are.
- You retrieve objects either by their identity, or using a query. With JDO you can use JDOQL, SQL or JPQL query languages. With JPA you can use JPQL or SQL query languages.
## 5 Dependencies

### 5.1 AccessPlatform : Dependencies

DataNucleus AccessPlatform utilises some third party software to provide some of its functionality. Dependent on how you intend to use this product you may have to also download some of these third party software packages. You can see below the dependencies and when they are required.

<table>
<thead>
<tr>
<th>Software</th>
<th>Description</th>
<th>Version</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Essential Dependencies</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JDO API</td>
<td>JDO API definition, developed by Apache JDO</td>
<td>3.0 or later</td>
<td>Required. Use v3.1-rc1 or 3.0.1 depending on whether you require JDO 3.1 or 3.0</td>
</tr>
<tr>
<td>JPA API</td>
<td>JPA API definition</td>
<td>2.1</td>
<td>Required if you are using the JPA API or JPA annotations. Note that the JPA &quot;Expert Group&quot; are seemingly too lazy to upload this into a freely downloadable location so you have to find one provided by other groups. Sadly there doesn't seem to be an Apache-licensed one currently, so you'll have to use the one under &quot;org.eclipse.persistence&quot;</td>
</tr>
<tr>
<td><strong>Datastore Dependencies</strong></td>
<td>(choose your datastore driver)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NeoDatis</td>
<td>NeoDatis object database</td>
<td>1.9.30</td>
<td>Required if you are using a NeoDatis datastore</td>
</tr>
<tr>
<td>jaxb-api</td>
<td>JAXB API</td>
<td>2.1</td>
<td>Required if you are using an XML datastore</td>
</tr>
<tr>
<td>jaxb-impl</td>
<td>JAXB Reference Implementation</td>
<td>2.x</td>
<td>Required if you are using an XML datastore</td>
</tr>
<tr>
<td>JDBC Driver</td>
<td>JDBC Driver for your chosen RDBMS</td>
<td></td>
<td>Required if you want to use an RDBMS datastore. Obtain from your RDBMS vendor</td>
</tr>
<tr>
<td>Apache POI</td>
<td>Apache library for writing to Microsoft documents</td>
<td>3.5+</td>
<td>Required if you want to use Excel (XLS/OOXML) documents</td>
</tr>
<tr>
<td>ODFDOM</td>
<td>ODF Toolkit for Java</td>
<td>0.8.7</td>
<td>Required if you want to use an ODF document for persistence. Required by ODFDOM</td>
</tr>
<tr>
<td>Xerces</td>
<td>Xerces XML parser</td>
<td>2.8+</td>
<td></td>
</tr>
<tr>
<td>Component</td>
<td>Description</td>
<td>Version</td>
<td>Required if you want to persist to...</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>------------------------------------------</td>
<td>---------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>Apache HBase</td>
<td>HBase</td>
<td>0.94+</td>
<td>to HBase datastores</td>
</tr>
<tr>
<td>HADOOP Core</td>
<td>HADOOP Core</td>
<td>1.0+</td>
<td>to HBase datastores</td>
</tr>
<tr>
<td>Apache ZooKeeper</td>
<td>Apache Zookeeper</td>
<td>3.4+</td>
<td>to HBase datastores</td>
</tr>
<tr>
<td>MongoDB Java driver</td>
<td>MongoDB Java driver</td>
<td>2.4+</td>
<td>to MongoDB datastores</td>
</tr>
<tr>
<td>Neo4j driver</td>
<td>Neo4j driver</td>
<td>1.9.4</td>
<td>to Neo4j datastores</td>
</tr>
<tr>
<td><strong>Feature Dependencies</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Log4j</td>
<td>Log4J logging library.</td>
<td>1.2+</td>
<td>to log using Log4J.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DataNucleus supports Log4J or JDK1.4 logging</td>
</tr>
<tr>
<td>mx4j</td>
<td>MX4J management library</td>
<td>3.0+</td>
<td>to use JMX with DataNucleus via MX4J</td>
</tr>
<tr>
<td>mx4j-tools</td>
<td>MX4J tools</td>
<td>1.2+</td>
<td>to use JMX with DataNucleus via MX4J</td>
</tr>
<tr>
<td>JodaTime</td>
<td>JodaTime</td>
<td>1.6+</td>
<td>to persist JodaTime java types</td>
</tr>
<tr>
<td>javax.time</td>
<td>JSR Time Library</td>
<td>0.6+</td>
<td>to persist javax.time types</td>
</tr>
<tr>
<td>GoogleCollections</td>
<td>GoogleCollections</td>
<td>1.0</td>
<td>to persist Google Collections java types, or are using BoneCP connection pool for RDBMS</td>
</tr>
<tr>
<td>EHCache</td>
<td>EHCache caching product</td>
<td>1.0+</td>
<td>to use EHCache for level 2 caching</td>
</tr>
<tr>
<td>OSCache</td>
<td>OSCache caching product</td>
<td>2.1</td>
<td>to use OSCache for level 2 caching</td>
</tr>
<tr>
<td>SwarmCache</td>
<td>SwarmCache caching product</td>
<td>1.0RC2</td>
<td>to use SwarmCache for level 2 caching</td>
</tr>
<tr>
<td>C3P0</td>
<td>C3P0 RDBMS connection pooling library</td>
<td>0.9.0+</td>
<td>you are using an RDBMS datastore and want to use C3P0 for connection pooling</td>
</tr>
<tr>
<td>Library</td>
<td>Description</td>
<td>Version</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>proxool</td>
<td>Proxool RDBMS connection pooling library</td>
<td>0.9.0RC3</td>
<td>Required if you are using an RDBMS datastore and want to use Proxool for connection pooling</td>
</tr>
<tr>
<td>commons-log</td>
<td>Apache commons logging library</td>
<td>1.0+</td>
<td>Required if you are using an RDBMS datastore and want to use Proxool for connection pooling</td>
</tr>
<tr>
<td>bonecp</td>
<td>BoneCP RDBMS connection pooling library</td>
<td>0.6.5</td>
<td>Required if you are using an RDBMS datastore and want to use BoneCP for connection pooling</td>
</tr>
<tr>
<td>SLF4J</td>
<td>SLF4J logging library</td>
<td>1.5.6</td>
<td>Required if you are using BoneCP for connection pooling</td>
</tr>
<tr>
<td>sdoapi</td>
<td>Oracle Spatial library</td>
<td>1.2+</td>
<td>Required if you want to persist Oracle spatial types</td>
</tr>
<tr>
<td>jta</td>
<td>JTA transaction API</td>
<td>1.0+</td>
<td>Required if you want to use JTA transactions</td>
</tr>
<tr>
<td>cache-api</td>
<td>Cache API</td>
<td>0.61+</td>
<td>Required if you want to use javax.cache L2 caching</td>
</tr>
<tr>
<td>validation-api</td>
<td>Bean validation API</td>
<td>1.0+</td>
<td>Required if you want to use bean validation (you also require a bean validation implementation)</td>
</tr>
</tbody>
</table>
6 Persistence API Choice

6.1 Persistence API: JDO or JPA?

There are two standard API's for persistence in Java - Java Data Objects (JDO) and Java Persistence API (JPA). JDO is designed for all datastores, and JPA is designed for RDBMS datastores only. DataNucleus supports both, fully, and also provides support for a REST API. When choosing the persistence API to use in your application you should bear the following factors in mind:

- **Target datastore**: JDO is designed for all datastores, whereas JPA is just designed around RDBMS and explicitly uses RDBMS/SQL terminology. If using RDBMS then you have the choice. If using, for example, a NoSQL store then JDO makes much more sense.

- **Datastore interoperability**: are you likely to change your datastore type at some point in the future? If so you likely ought to use JDO due to its design.

- **API**: both APIs are very similar. JDO provides more options and control though for basic persistence and retrieval there are differences only in the namings.

- **ORM**: JDO has a more complete ORM definition, as shown on Apache JDO ORM Guide.

- **Experience**: do your developers know a particular API already? As mentioned the API's themselves are very similar, though the metadata definition is different. Remember that you can use JPA metadata with the JDO API, and vice-versa.

- **Querying**: do you need a flexible query language that is object-oriented and extensible? JDOQL provides this and the implementation in DataNucleus allows extensions. If you just want SQL then you can use JDO or JPA since both provide this.

- **Fetch Control**: do you need full control of what is fetched, and when? JDO provides fetch groups, whereas JPA2.1 now provides EntityGraphs (A subset of fetch groups). Use JDO if full fetch groups is an important factor for your design, otherwise either.

- **API experience**: you may be more likely to find people with experience in JPA, but your developers may already have experience with one API.

There is a further comparison of JDO and JPA on technical grounds over at Apache JDO.

6.2 Persistence API FAQ

To supplement the factors above to bear in mind when choosing your persistence API, there has been much FUD on the web about JDO and JPA, largely perpetrated by RDBMS vendors, and we provide a FAQ that corrects many of these points so you can base your decision on what is best for you.

**Q: Which specification was the original?**

JDO was the first Java persistence specification, starting in 1999, and the JDO 1.0 specification being published in April 2002. This provided the persistence API, and was standardised as JSR012. In May 2006 JDO2 was released. This provided an update to the persistence API as well as a complete definition of ORM, standardised as JSR243. Later in May 2006 JPA1 was released. This provided a persistence API, and a limited definition of ORM, concentrating only on RDBMS, and was standardised as JSR220.

**Q: Why did JPA come about when we already had a specification for Java persistence in JDO?**

Politics. RDBMS vendors apparently didn't like the idea of having a technology that allowed users to leverage a single API, and easily swap to a different type of datastore. Much pressure was applied to SUN to provide a different specification, and even to try to say that JPA was to supercede JDO. The JCP is dominated by large organisations and SUN capitulated. They even published a "FAQ" to try to justify their decision.

**Q: Is JDO dead?**
No. As part of SUN's capitulation above, they donated JDO to Apache to develop the technology further. There have been the following revisions to the JDO specification:

- JDO2.1 adding on support for annotations, enums, and some JPA concepts.
- JDO2.2 adding on support for dynamic fetch groups, transaction isolation and cache control.
- JDO3.0 adding on MetaData/Enhancer APIs as well as query timeout/cancel support etc.

In addition, JDO3.1 is reaching its conclusion, adding on support for more JDOQL methods, as well as control over position of a column, size of a sequence etc.

**Q: Will JPA replace JDO?**

It is very hard to see that happening since JPA provides nothing to cater for persistence of Java objects to non-RDBMS datastores (LDAP, ODBMS, XML, ODF, Excel etc). It doesn't even provide a complete definition of ORM, so cannot yet compete with JDO's ORM handling. Even in JPA2 (final in late 2009) there are still basic ORM concepts that are not handled by JPA yet JDO standardises them. JDO is still being developed, and while users require this technology then it will continue to exist. DataNucleus will continue to support both APIs since there is a need for both in modern enterprise applications despite what Oracle, IBM, et al try to impose on you.

**Q: What differences are there between how JDO is developed and how JPA is developed?**

JPA is developed in isolation by an "expert group" though in JPA2.1 they have added a mailing list so you can see their discussion (not that they'll reply to input necessarily). JDO is developed in public by anybody interested in the technology. The tests to verify compliance with JPA are only available after signing non-disclosure agreements with SUN and this process can take up to 3 months just to get the test suite (if ever). The tests to verify compliance with JDO are freely downloadable and can be run by users or developers. This means that anybody can check whether an implementation is compliant with JDO, whereas the same is not true of JPA. DataNucleus runs the JDO3.x and JPA1 TCKs at frequent intervals and we publish the results on our website. DataNucleus has been prevented from accessing the JPA2 TCK (by Oracle and the JCP, documented in our blog).

**Q: Why should I use JDO when JPA is supported by "large organisations"?**

By "large organisations" you presumably mean commercial organisations like Oracle, IBM, RedHat. And they have their own vested interest in RDBMS technologies, or in selling application servers. You should make your own decisions rather than just follow down the path you are shepherded in by any commercial organisation. Your application will be supported by you not by them. The technology you use should be the best for the job and what you feel most comfortable with. If you feel more comfortable with JPA and it provides all that your application needs then use it. Similarly if JDO provides what you need then you use that. For this reason DataNucleus provides support for both specifications.
7 Development Process

7.1 DataNucleus AccessPlatform Development Process

DataNucleus attempts to make the whole process of persisting data a transparent process. The idea revolves around the developer having a series of Java classes that need persisting. With DataNucleus, the developer defines the persistence of these classes using Metadata (defined in XML, annotations or by API), and byte-code “enhances” these classes. DataNucleus also provides RDBMS SchemaTool that allows for schema generation/validation before running your application, to make sure that all is correctly mapped. Finally you provide persistence code (to manage the persistence of your objects), and queries (to retrieve your persisted data). DataNucleus Access Platform implements all JDO specifications (1.0, 2.0, 2.1, 2.2, 3.0) and also all JPA specifications (1.0, 2.0, 2.1). The following diagram shows the process for DataNucleus AccessPlatform (several parts of the diagram are clickable giving more details).
8 Compatibility

8.1 AccessPlatform : Compatibility

There are two aspects to compatibility that we discuss here. The compatibility between DataNucleus plugins, and the compatibility of DataNucleus with third party software.

8.1.1 Plugin Compatibility

If you download one of the DataNucleus AccessPlatform distribution zip files you get a consistent set of DataNucleus plugins. Alternatively you can inspect the Maven "POM" files in Maven Central repository to see the dependency requirements. For the record the latest released versions of the following plugins are all consistent.

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>datanucleus-core</td>
<td>3.2.12</td>
</tr>
<tr>
<td>datanucleus-api-jdo</td>
<td>3.2.8</td>
</tr>
<tr>
<td>datanucleus-api-jpa</td>
<td>3.3.7</td>
</tr>
<tr>
<td>datanucleus-api-rest</td>
<td>3.2.2</td>
</tr>
<tr>
<td>datanucleus-excel</td>
<td>3.2.0-release</td>
</tr>
<tr>
<td>datanucleus-hbase</td>
<td>3.2.6</td>
</tr>
<tr>
<td>datanucleus-json</td>
<td>3.2.1</td>
</tr>
<tr>
<td>datanucleus-ldap</td>
<td>3.2.2</td>
</tr>
<tr>
<td>datanucleus-mongodb</td>
<td>3.2.8</td>
</tr>
<tr>
<td>datanucleus-neo4j</td>
<td>3.2.3</td>
</tr>
<tr>
<td>datanucleus-neodatis</td>
<td>3.2.2</td>
</tr>
<tr>
<td>datanucleus-odf</td>
<td>3.2.1</td>
</tr>
<tr>
<td>datanucleus-rdbms</td>
<td>3.2.11</td>
</tr>
<tr>
<td>datanucleus-xml</td>
<td>3.2.1</td>
</tr>
<tr>
<td>datanucleus-cache</td>
<td>3.1.3</td>
</tr>
<tr>
<td>datanucleus-geospatial</td>
<td>3.2.6</td>
</tr>
<tr>
<td>datanucleus-jodatime</td>
<td>3.2.1</td>
</tr>
<tr>
<td>datanucleus-guava</td>
<td>3.2.3</td>
</tr>
<tr>
<td>datanucleus-jdo-query</td>
<td>3.2.2</td>
</tr>
<tr>
<td>datanucleus-jpa-query</td>
<td>3.2.3</td>
</tr>
</tbody>
</table>

8.1.2 Third Party Compatibility

We aim to make DataNucleus AccessPlatform as compatible with related software as possible. Here we give an overview of known compatibilities/problems

<table>
<thead>
<tr>
<th>Software</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraniteDS</td>
<td>Fully compatible from GraniteDS 2.0+</td>
</tr>
<tr>
<td>Scala</td>
<td>Fully compatible</td>
</tr>
<tr>
<td><strong>Play Framework</strong></td>
<td>Fully compatible with version 2.0 or later. <em>Version 1 of Play had a hardcoded Hibernate implementation which was obviously a bad idea when the whole point of having a persistence standard (JPA) is to allow portability.</em></td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>GWT</strong></td>
<td>Current versions of GWT (2+) ought to be able to serialise/deserialise any detached JDO/JPA objects. Earlier GWT versions had a problem with a bytecode enhancement field of type Object[] and there was a project <strong>GILEAD</strong> that attempted to handle this for various persistence solutions (and in version 1.3 will have specific support for DataNucleus, already in Gilead SVN). Also look at <a href="#">this</a> and <a href="#">this</a>.</td>
</tr>
<tr>
<td><strong>iReport v5</strong></td>
<td>Fully compatible, but you could consider removing the following iReport-5.0.0/ireport/modules/ext/commons-dbcp-1.2.2.jar and iReport-5.0.0/ireport/modules/ext/<em>hive</em> as they have been found by some to cause conflicts.</td>
</tr>
<tr>
<td><strong>SpringFramework</strong></td>
<td>Fully compatible since DataNucleus is a compliant JDO/JPA implementation</td>
</tr>
</tbody>
</table>
9 Services

9.1 DataNucleus Services: Free and Commercial

With any software, there are times when you need assistance to make full use of it. Here at DataNucleus we want you to make the best of the software that we provide. For that reason we provide both commercial and free support facilities.

9.1.1 DataNucleus: Commercial Support

Where you or your company require timely support when you need it without having to wait for somebody to respond on a forum we provide commercial support. You could, for example, have a preference for email support, or maybe phone support. If this is the case then we can discuss what would be possible. Please refer to our Support services for details.

9.1.2 DataNucleus: Free Support

DataNucleus provides its own online forums providing a place to discuss issues you are having. We don't guarantee to provide answers on this forum. It is simply a place where you could get some level of support when people have time. This may not be adequate for some people hence why we provide the commercial version above.

1. Check the Documentation before anything else. The answer is usually there, either in a tutorial/example, or in one of the many guides.

2. Look in the DataNucleus Log. This usually contains a lot of information that may answer the issue. You can always configure the log to give more output.

3. Try a recent build to see if your version is out of date and the expected result is achieved with the latest nightly builds.

4. Go to the Online Forum and ask. Always try to give as clear a description of the problem as possible, together with your input data, and any associated log output. Please be aware that we have very little time for this type of support and contributors to DataNucleus are more likely to get any available free support.

9.1.3 DataNucleus: Commercial Consulting/Training

The DataNucleus experts are available for Consulting for cases where you need somebody intrinsically familiar with the DataNucleus system available to resolve any particular implementation issues.

Similarly, if your company would like to build up your own experience in DataNucleus and would like a kick start for this process the DataNucleus experts are available to provide Training.

One further possible use for DataNucleus consulting is to provide a plugin for a datastore that we don't currently support. Do you have a datastore that you'd like to be able to persist to using JDO or JPA? We can help you achieve this. Contact us to discuss.
## 10 ORM Relationships

### 10.1 ORM Relationships

There are 2 prevalent persistence specifications in the Java ORM world. JDO2 provides the most complete definition, whilst JPA is the most recent. In this guide we show the different types of ORM relation commonly used, and mark against it which specification supports it. This list is not yet complete but will be added to to provide a comprehensive list of relationship type and where you can find it.

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Relation</th>
<th>JDO2</th>
<th>JPA2</th>
<th>DataNucleus</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>1-1 Unidirectional</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>PC</td>
<td>1-1 Bidirectional</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>PC</td>
<td>1-1 serialised</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>PC</td>
<td>1-1 CompoundIdentity</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>PC</td>
<td>1-N CompoundIdentity</td>
<td>☑</td>
<td></td>
<td>☑</td>
</tr>
<tr>
<td>Interface</td>
<td>1-1 Unidirectional</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Interface</td>
<td>1-1 Bidirectional</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Interface</td>
<td>1-1 serialised</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Collection&lt;PC&gt;</td>
<td>1-N ForeignKey</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Collection&lt;PC&gt;</td>
<td>1-N JoinTable</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Collection&lt;PC&gt;</td>
<td>1-N JoinTable using shared JoinTable</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Collection&lt;PC&gt;</td>
<td>1-N JoinTable using shared ForeignKey</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Collection&lt;P&gt;</td>
<td>M-N JoinTable</td>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
<td><img src="image3" alt="Diagram" /></td>
</tr>
<tr>
<td>--------------</td>
<td>---------------</td>
<td>-------------------</td>
<td>-------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>Collection&lt;P&gt;</td>
<td>1-N CompoundIdentity Unidirectional</td>
<td><img src="image4" alt="Diagram" /></td>
<td><img src="image5" alt="Diagram" /></td>
<td><img src="image6" alt="Diagram" /></td>
</tr>
<tr>
<td>Collection&lt;P&gt;</td>
<td>1-N serialised Collection</td>
<td><img src="image7" alt="Diagram" /></td>
<td><img src="image8" alt="Diagram" /></td>
<td><img src="image9" alt="Diagram" /></td>
</tr>
<tr>
<td>Collection&lt;P&gt;</td>
<td>1-N JoinTable Collection of serialised elements</td>
<td><img src="image10" alt="Diagram" /></td>
<td><img src="image11" alt="Diagram" /></td>
<td><img src="image12" alt="Diagram" /></td>
</tr>
<tr>
<td>List&lt;P&gt;</td>
<td>1-N ForeignKey Unidirectional Indexed List</td>
<td><img src="image13" alt="Diagram" /></td>
<td><img src="image14" alt="Diagram" /></td>
<td><img src="image15" alt="Diagram" /></td>
</tr>
<tr>
<td>List&lt;P&gt;</td>
<td>1-N ForeignKey Bidirectional Indexed List</td>
<td><img src="image16" alt="Diagram" /></td>
<td><img src="image17" alt="Diagram" /></td>
<td><img src="image18" alt="Diagram" /></td>
</tr>
<tr>
<td>List&lt;P&gt;</td>
<td>1-N JoinTable Unidirectional Indexed List</td>
<td><img src="image19" alt="Diagram" /></td>
<td><img src="image20" alt="Diagram" /></td>
<td><img src="image21" alt="Diagram" /></td>
</tr>
<tr>
<td>List&lt;P&gt;</td>
<td>1-N JoinTable Bidirectional Indexed List</td>
<td><img src="image22" alt="Diagram" /></td>
<td><img src="image23" alt="Diagram" /></td>
<td><img src="image24" alt="Diagram" /></td>
</tr>
<tr>
<td>List&lt;Non-PC&gt;</td>
<td>1-N JoinTable Indexed List</td>
<td><img src="image25" alt="Diagram" /></td>
<td><img src="image26" alt="Diagram" /></td>
<td><img src="image27" alt="Diagram" /></td>
</tr>
<tr>
<td>List&lt;P&gt;</td>
<td>1-N ForeignKey Unidirectional Ordered List</td>
<td><img src="image28" alt="Diagram" /></td>
<td><img src="image29" alt="Diagram" /></td>
<td><img src="image30" alt="Diagram" /></td>
</tr>
<tr>
<td>List&lt;P&gt;</td>
<td>1-N ForeignKey Bidirectional Ordered List</td>
<td><img src="image31" alt="Diagram" /></td>
<td><img src="image32" alt="Diagram" /></td>
<td><img src="image33" alt="Diagram" /></td>
</tr>
<tr>
<td>List&lt;P&gt;</td>
<td>1-N JoinTable Unidirectional Ordered List</td>
<td><img src="image34" alt="Diagram" /></td>
<td><img src="image35" alt="Diagram" /></td>
<td><img src="image36" alt="Diagram" /></td>
</tr>
<tr>
<td>List&lt;P&gt;</td>
<td>1-N JoinTable Bidirectional Ordered List</td>
<td><img src="image37" alt="Diagram" /></td>
<td><img src="image38" alt="Diagram" /></td>
<td><img src="image39" alt="Diagram" /></td>
</tr>
<tr>
<td>Map&lt;P, PC&gt;</td>
<td>1-N JoinTable Map</td>
<td><img src="image40" alt="Diagram" /></td>
<td><img src="image41" alt="Diagram" /></td>
<td><img src="image42" alt="Diagram" /></td>
</tr>
<tr>
<td>Map&lt;Non-PC, PC&gt;</td>
<td>1-N JoinTable Map</td>
<td><img src="image43" alt="Diagram" /></td>
<td><img src="image44" alt="Diagram" /></td>
<td><img src="image45" alt="Diagram" /></td>
</tr>
<tr>
<td>Map&lt;P, Non-PC&gt;</td>
<td>1-N JoinTable Map</td>
<td><img src="image46" alt="Diagram" /></td>
<td><img src="image47" alt="Diagram" /></td>
<td><img src="image48" alt="Diagram" /></td>
</tr>
<tr>
<td>Map&lt;Non-PC, Non-PC&gt;</td>
<td>1-N JoinTable Map</td>
<td><img src="image49" alt="Diagram" /></td>
<td><img src="image50" alt="Diagram" /></td>
<td><img src="image51" alt="Diagram" /></td>
</tr>
<tr>
<td>Map&lt;Non-PC, PC&gt;</td>
<td>1-N ForeignKey Map Unidirectional (key stored in value)</td>
<td><img src="image52" alt="Diagram" /></td>
<td><img src="image53" alt="Diagram" /></td>
<td><img src="image54" alt="Diagram" /></td>
</tr>
<tr>
<td>Map&lt;Non-PC, PC&gt;</td>
<td>1-N ForeignKey Map Bidirectional (key stored in value)</td>
<td><img src="image55" alt="Diagram" /></td>
<td><img src="image56" alt="Diagram" /></td>
<td><img src="image57" alt="Diagram" /></td>
</tr>
</tbody>
</table>
| Map<PC, Non-PC> | 1:N ForeignKey  
Map Unidirectional 
(value stored in key) |
| Map<PC, PC> | 1:N serialised Map |
| Map<PC, PC> | 1:N JoinTable Map 
of serialised keys/ 
values |
| PC[ ] | 1:N ForeignKey 
Unidirectional Array |
| PC[ ] | 1:N JoinTable 
Unidirectional Array |
| PC[ ] | 1:N serialised Array |
| Non-PC[ ] | 1:N JoinTable 
Unidirectional Array |
11 Persistence Properties

11.1 Persistence Properties

JDO and JPA with DataNucleus are highly configurable using persistence properties. When defining your `PersistenceManagerFactory` or `EntityManagerFactory` you have the opportunity to control many aspects of the persistence process. DataNucleus is perhaps more configurable than any other JDO/JPA implementation in this respect. This section defines the properties available for use. Please bear in mind that these properties are only for use with DataNucleus and will not work with other JDO/JPA implementations. **All persistence property names are case-insensitive**

- **Datastore Definition** - datastore properties
- **General** - general properties
- **Schema Control** - properties controlling the generation of the datastore schema.
- **Transactions and Locking** - properties controlling how transactions operate
- **Caching** - properties controlling the behaviour of the cache(s)
- **Bean Validation** - properties controlling bean validation at persist
- **Value Generation** - properties controlling the generation of object identities and field values
- **MetaData** - metadata properties
- **Auto-Start** - Auto-Start Mechanism properties
- **Query** - properties controlling the behaviour of queries
- **Datastore-Specific** - properties for particular datastores e.g RDBMS

11.1.1 Datastore Definition

<table>
<thead>
<tr>
<th>datanucleus.ConnectionURL</th>
<th>URL specifying the datastore to use for persistence. Note that this will define the type of datastore as well as the datastore itself. Please refer to the datastores guides for the URL appropriate for the type of datastore you're using.</th>
</tr>
</thead>
<tbody>
<tr>
<td>datanucleus.ConnectionUserName</td>
<td>Username to use for connecting to the DB</td>
</tr>
<tr>
<td>datanucleus.ConnectionPassword</td>
<td>Password to use for connecting to the DB. See <a href="#">datanucleus.ConnectionPasswordDecrypter</a> for a way of providing an encrypted password here</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>datanucleus.ConnectionDriverName</code></td>
<td>The name of the (JDBC) driver to use for the DB (for RDBMS only).</td>
</tr>
<tr>
<td><code>datanucleus.ConnectionFactory</code></td>
<td>Instance of a connection factory for transactional connections. This is an alternative to <code>datanucleus.ConnectionURL</code>. For RDBMS, it must be an instance of <code>javax.sql.DataSource</code>. See <a href="#">Data Sources</a>.</td>
</tr>
<tr>
<td><code>datanucleus.ConnectionFactory2</code></td>
<td>Instance of a connection factory for nontransactional connections. This is an alternative to <code>datanucleus.ConnectionURL</code>. For RDBMS, it must be an instance of <code>javax.sql.DataSource</code>. See <a href="#">Data Sources</a>.</td>
</tr>
<tr>
<td><code>datanucleus.ConnectionFactoryName</code></td>
<td>The JNDI name for a connection factory for transactional connections. For RDBMS, it must be a JNDI name that points to a <code>javax.sql.DataSource</code> object. See <a href="#">Data Sources</a>.</td>
</tr>
<tr>
<td><code>datanucleus.ConnectionFactory2Name</code></td>
<td>The JNDI name for a connection factory for nontransactional connections. For RDBMS, it must be a JNDI name that points to a <code>javax.sql.DataSource</code> object. See <a href="#">Data Sources</a>.</td>
</tr>
</tbody>
</table>
### datanucleus.ConnectionPasswordDecrypter

**Description**
Name of a class that implements org.datanucleus.store.connection.DecryptionProvider and should only be specified if the password is encrypted in the persistence properties.

**Range of Values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 11.1.2 General

#### datanucleus.IgnoreCache

**Description**
Whether to ignore the cache for queries. If the user sets this to true then the query will evaluate in the datastore, but the instances returned will be formed from the datastore; this means that if an instance has been modified and its datastore values match the query then the instance returned will not be the currently cached (updated) instance, instead an instance formed using the datastore values.

**Range of Values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>

#### datanucleus.Multithreaded

**Description**
Whether to run the PM/EM multithreaded. Note that this is a hint only to try to allow thread-safe operations on the PM/EM.

**Range of Values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>

#### datanucleus.NontransactionalRead

**Description**
Whether to allow nontransactional reads

**Range of Values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>

#### datanucleus.NontransactionalWrite

**Description**
Whether to allow nontransactional writes

**Range of Values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>datanucleus.RetainValues</td>
<td>Whether to suppress the clearing of values from persistent instances on transaction completion. With JDO this defaults to false, whereas for JPA it is true.</td>
</tr>
<tr>
<td>datanucleus.RestoreValues</td>
<td>Whether persistent object have transactional field values restored when transaction rollback occurs.</td>
</tr>
<tr>
<td>datanucleus.Mapping</td>
<td>Name for the ORM MetaData mapping files to use with this PMF. For example if this is set to &quot;mysql&quot; then the implementation looks for MetaData mapping files called &quot;{classname}-mysql.orm&quot; or &quot;package-mysql.orm&quot;. If this is not specified then the JDO implementation assumes that all is specified in the JDO MetaData file.</td>
</tr>
<tr>
<td>datanucleus.mapping.Catalog</td>
<td>Name of the catalog to use by default for all classes persisted using this PMF/EMF. This can be overridden in the MetaData where required, and is optional. DataNucleus will prefix all table names with this catalog name if the RDBMS supports specification of catalog names in DDL. <em>RDBMS datastores only</em></td>
</tr>
<tr>
<td>datanucleus.mapping.Schema</td>
<td>Name of the catalog to use by default for all classes persisted using this PMF/EMF. This can be overridden in the MetaData where required, and is optional. DataNucleus will prefix all table names with this catalog name if the RDBMS supports specification of catalog names in DDL. <em>RDBMS datastores only</em></td>
</tr>
</tbody>
</table>
### Description
Name of the schema to use by default for all classes persisted using this PMF/EMF. This can be overridden in the MetaData where required, and is optional. DataNucleus will prefix all table names with this schema name if the RDBMS supports specification of schema names in DDL. _RDBMS datastores only_

### datanucleus.tenantId
**Description**
String id to use as a discriminator on all persistable class tables to restrict data for the tenant using this application instance (aka multi-tenancy via discriminator). _RDBMS, MongoDB datastores only_

### datanucleus.DetachAllOnCommit
**Description**
Allows the user to select that when a transaction is committed all objects enlisted in that transaction will be automatically detached.

**Range of Values**
true | false

### datanucleus.detachAllOnRollback
**Description**
Allows the user to select that when a transaction is rolled back all objects enlisted in that transaction will be automatically detached.

**Range of Values**
true | false

### datanucleus.CopyOnAttach
**Description**
Whether, when attaching a detached object, we create an attached copy or simply migrate the detached object to attached state

**Range of Values**
true | false

### datanucleus.allowAttachOfTransient
**Description**
When you call EM.merge with a transient object (with PK fields set), if you enable this feature then it will first check for existence of an object in the datastore with the same identity and, if present, will merge into that object (rather than just trying to persist a new object).
### datanucleus.attachSameDatastore

**Description**
When attaching an object DataNucleus by default assumes that you're attaching to the same datastore as you detached from. DataNucleus does though allow you to attach to a different datastore (for things like replication). Set this to `false` if you want to attach to a different datastore to what you detached from.

**Range of Values**
true | false

### datanucleus.detachAsWrapped

**Description**
When detaching, any mutable second class objects (Collections, Maps, Dates etc) are typically detached as the basic form (so you can use them on client-side of your application). This property allows you to select to detach as wrapped objects. It only works with "detachAllOnCommit" situations (not with detachCopy) currently.

**Range of Values**
true | false

### datanucleus.DetachOnClose

**Description**
This allows the user to specify whether, when a PM/EM is closed, that all objects in the L1 cache are automatically detached. Users are recommended to use the `datanucleus.DetachAllOnCommit` wherever possible. This will not work in JCA mode.

**Range of Values**
false | true

### datanucleus.detachmentFields

**Description**
When detaching you can control what happens to loaded/unloaded fields of the FetchPlan. The default for JDO is to load any unloaded fields of the current FetchPlan before detaching. You can also unload any loaded fields that are not in the current FetchPlan (so you only get the fields you require) as well as a combination of both options.

**Range of Values**
load-fields | unload-fields | load-unload-fields

### datanucleus.maxFetchDepth
### datanucleus.maxFetchDepth

**Description**
Specifies the default maximum fetch depth to use for fetching operations. The JDO spec defines a default of 1, meaning that only the first level of related objects will be fetched by default. The JPA spec doesn’t provide fetch group control, just a “default fetch group” type concept, consequently the default there is -1 currently.

**Range of Values**
-1 | 1 | positive integer (non-zero)

### datanucleus.detachedState

**Description**
Allows control over which mechanism to use to determine the fields to be detached. By default DataNucleus uses the defined “fetch-groups”. Obviously JPA1/JPA2 don’t have that (although it is an option with DataNucleus), so we also allow loaded which will detach just the currently loaded fields, and all which will detach all fields of the object (be careful with this option since it, when used with maxFetchDepth of -1 will detach a whole object graph!)

**Range of Values**
fetch-groups | all | loaded

### datanucleus.TransactionType

**Description**
Type of transaction to use. If running under JavaSE the default is RESOURCE_LOCAL, and if running under JavaEE the default is JTA.

**Range of Values**
RESOURCE_LOCAL | JTA

### datanucleus.ServerTimeZoneID

**Description**
Id of the TimeZone under which the datastore server is running. If this is not specified or is set to null it is assumed that the datastore server is running in the same timezone as the JVM under which DataNucleus is running.

**Range of Values**

### datanucleus.PersistenceUnitName

**Description**
Name of a persistence-unit to be found in a persistence.xml file (under META-INF) that defines the persistence properties to use and the classes to use within the persistence process.

**Range of Values**

© 2015, DataNucleus • ALL RIGHTS RESERVED.
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>datanucleus.PersistenceUnitLoadClasses</td>
<td>Used when we have specified the persistence-unit name for a PMF/EMF and where we want the datastore &quot;tables&quot; for all classes of that persistence-unit loading up into the StoreManager. Defaults to false since some databases are slow so such an operation would slow down the startup process.</td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.persistenceXmlFilename</td>
<td>URL name of the persistence.xml file that should be used instead of using &quot;META-INF/persistence.xml&quot;.</td>
<td></td>
</tr>
<tr>
<td>datanucleus.datastoreReadTimeout</td>
<td>The timeout to apply to all reads (millisecs). e.g by query or by PM.getObjectById(). Only applies if the underlying datastore supports it</td>
<td>0</td>
</tr>
<tr>
<td>datanucleus.datastoreWriteTimeout</td>
<td>The timeout to apply to all writes (millisecs). e.g by makePersistent, or by an update. Only applies if the underlying datastore supports it</td>
<td>0</td>
</tr>
<tr>
<td>datanucleus.singletonPMFForName</td>
<td>Whether to only allow a singleton PMF for a particular name (the name can be either the name of the PMF in jdoconfig.xml, or the name of the persistence-unit). If a subsequent request is made for a PMF with a name that already exists then a warning will be logged and the original PMF returned.</td>
<td>true</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Range of Values</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>-----------------</td>
</tr>
<tr>
<td><code>datanucleus.allowListenerUpdateAfterInit</code></td>
<td>Whether you want to be able to add/remove listeners on the JDO PMF after it is marked as not configurable (when the first PM is created). The default matches the JDO spec, not allowing changes to the listeners in use.</td>
<td>true</td>
</tr>
<tr>
<td><code>datanucleus.storeManagerType</code></td>
<td>Type of the StoreManager to use for this PMF/EMF. This has typical values of &quot;rdbms&quot;, &quot;mongodb&quot;. If it isn't specified then it falls back to trying to find the StoreManager from the connection URL. The associated DataNucleus plugin has to be in the CLASSPATH when selecting this. When using data sources (as usually done in a JavaEE container), DataNucleus cannot find out the correct type automatically and this option must be set.</td>
<td>rdbms</td>
</tr>
<tr>
<td><code>datanucleus.jmxType</code></td>
<td>Which JMX server to use when hooking into JMX. Please refer to the Monitoring Guide</td>
<td>default</td>
</tr>
<tr>
<td><code>datanucleus.deletionPolicy</code></td>
<td>Allows the user to decide the policy when deleting objects. The default is &quot;JDO2&quot; which firstly checks if the field is dependent and if so deletes dependents, and then for others will null any foreign keys out. The problem with this option is that it takes no account of whether the user has also defined <code>&lt;foreign-key&gt;</code> elements, so we provide a &quot;DataNucleus&quot; mode that does the dependent field part first and then if a FK element is defined will leave it to the FK in the datastore to perform any actions, and otherwise does the nulling.</td>
<td></td>
</tr>
</tbody>
</table>
### Range of Values

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDO2</td>
<td>DataNucleus</td>
</tr>
</tbody>
</table>

#### datanucleus.identityStringTranslatorType

**Description**

You can allow identities input to `pm getObjectById(id)` be translated into valid JDO ids if there is a suitable translator. See [Identity String Translator Plugin](#).

**Range of Values**

- *datanucleus.identityStringTranslatorType*

#### datanucleus.identityKeyTranslatorType

**Description**

You can allow identities input to `pm getObjectById(cls, key)` be translated into valid JDO ids if there is a suitable key translator. See [Identity Key Translator Plugin](#).

**Range of Values**

- *datanucleus.identityKeyTranslatorType*

#### datanucleus.datastoreIdentityType

**Description**

Which "datastore-identity" class plugin to use to represent datastore identities. Refer to [Datastore Identity extensions](#) for details.

**Range of Values**

- *datanucleus.datastoreIdentityType*

#### datanucleus.executionContext.maxIdle

**Description**

Specifies the maximum number of ExecutionContext objects that are pooled ready for use

**Range of Values**

- 20 | integer value greater than 0

#### datanucleus.executionContext.reaperThread

**Description**

Whether to start a reaper thread that continually monitors the pool of ExecutionContext objects and frees them off after they have surpassed their expiration period

**Range of Values**

- *datanucleus.executionContext.reaperThread*

#### datanucleus.objectProvider.className

**Description**

Class name for the ObjectProvider to use when managing object state. From v3.2.2

**Range of Values**

- *datanucleus.objectProvider.className*
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>datanucleus.useImplementationCreator</td>
<td>Whether to allow use of the implementation-creator (feature of JDO to dynamically create implementations of persistent interfaces). Defaults to true for JDO, and false for JPA.</td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.manageRelationships</td>
<td>This allows the user control over whether DataNucleus will try to manage bidirectional relations, correcting the input objects so that all relations are consistent. This process runs when flush()/commit() is called. JDO defaults to true and JPA defaults to false. You can set it to false if you always set both sides of a relation when persisting/updating.</td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.manageRelationshipsChecks</td>
<td>This allows the user control over whether DataNucleus will make consistency checks on bidirectional relations. If “datanucleus.managedRelationships” is not selected then no checks are performed. If a consistency check fails at flush()/commit() then a JDOUserException is thrown. You can set it to false if you want to omit all consistency checks.</td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.persistenceByReachabilityAtCommit</td>
<td>Whether to run the “persistence-by-reachability” algorithm at commit() time. This means that objects that were reachable at a call to makePersistent() but that are no longer persistent will be removed from persistence. For performance improvements, consider turning this off.</td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.classLoaderResolverName</td>
<td>Name of a ClassLoaderResolver to use in class loading. DataNucleus provides a default that loosely follows the JDO specification for class loading. This property allows the user to override this with their own class better suited to their own loading requirements.</td>
<td></td>
</tr>
</tbody>
</table>
### datanucleus.primaryClassLoader

**Description**
Sets a primary classloader for situations where a primary classloader is not accessible. This ClassLoader is used when the class is not found in the default ClassLoader search path. As example, when the database driver is loaded by a different ClassLoader not in the ClassLoader search path for JDO or JPA specifications.

**Range of Values**
instance of java.lang.ClassLoader

### datanucleus.plugin.pluginRegistryClassName

**Description**
Name of a class that acts as registry of plug-ins. When used outside of OSGi this will be auto set to org.datanucleus.plugin.NonManagedPluginRegistry. If used within an Eclipse OSGi environment it will auto set this to org.datanucleus.plugin.EclipsePluginRegistry. If you are within a non-Eclipse OSGi environment you can set this to org.datanucleus.plugin.OSGiPluginRegistry

**Range of Values**
{fully-qualified class name}

### datanucleus.plugin.pluginRegistryBundleCheck

**Description**
Defines what happens when plugin bundles are found and are duplicated

**Range of Values**
EXCEPTION | LOG | NONE

### datanucleus.plugin.allowUserBundles

**Description**
Defines whether user-provided bundles providing DataNucleus extensions will be registered. This is only respected if used in a non-Eclipse OSGi environment.

**Range of Values**
true | false

### datanucleus.plugin.validatePlugins

**Description**
Defines whether a validation step should be performed checking for plugin dependencies etc. This is only respected if used in a non-Eclipse OSGi environment.

**Range of Values**
false | true

### datanucleus.localisation.language

**Description**
Language to use for logging (e.g en, es)
### 11.1.3 Schema Control

**datanucleus.autoCreateSchema**

Description: Whether to automatically generate any tables and constraints that don't exist. Please refer to the [Schema Guide](#) for more details.

Range of Values: `true` | `false`

**datanucleus.autoCreateTables**

Description: Whether to automatically generate any tables that don't exist. Please refer to the [Schema Guide](#) for more details.

Range of Values: `true` | `false`

**datanucleus.autoCreateColumns**

Description: Whether to automatically generate any columns that don't exist. Please refer to the [Schema Guide](#) for more details.
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>datanucleus.autoCreateConstraints</td>
<td>Whether to automatically generate any constraints that don't exist. Please refer to the Schema Guide for more details.</td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.autoCreateWarnOnError</td>
<td>Whether to only log a warning when errors occur during the auto-creation/validation process. Please use with care since if the schema is incorrect errors will likely come up later and this will postpone those error checks til later, when it may be too late!!</td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.validateSchema</td>
<td>Alias for defining datanucleus.validateTables, datanucleus.validateColumns and datanucleus.validateConstraints as all true. Please refer to the Schema Guide for more details.</td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.validateTables</td>
<td>Whether to validate tables against the persistence definition. Please refer to the Schema Guide for more details.</td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.validateColumns</td>
<td>Whether to validate columns against the persistence definition. This refers to the column detail structure and NOT to whether the column exists or not. Please refer to the Schema Guide for more details.</td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.validateConstraints</td>
<td>Whether to validate table constraints against the persistence definition. Please refer to the Schema Guide for more details.</td>
<td>true</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Range of Values</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>datanucleus.readOnlyDatastore</td>
<td>Whether the datastore is read-only or not (fixed in structure and contents).</td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.readOnlyDatastoreAction</td>
<td>What happens when a datastore is read-only and an object is attempted to be persisted.</td>
<td>EXCEPTION</td>
</tr>
<tr>
<td>datanucleus.fixedDatastore</td>
<td>Whether the datastore is fixed in structure or not.</td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.generateSchema.database.mode</td>
<td>Whether to perform any schema generation to the database at startup. Will process the schema for all classes that have metadata loaded at startup (i.e the classes specified in a persistence-unit).</td>
<td>create</td>
</tr>
<tr>
<td>datanucleus.generateSchema.scripts.mode</td>
<td>Whether to perform any schema generation into scripts at startup. Will process the schema for all classes that have metadata loaded at startup (i.e the classes specified in a persistence-unit).</td>
<td>create</td>
</tr>
<tr>
<td>datanucleus.generateSchema.scripts.create.tar</td>
<td>Name of the script file to write to if doing a &quot;create&quot; with the target as &quot;scripts&quot;</td>
<td>datanucleus-schema-create.ddl</td>
</tr>
<tr>
<td>datanucleus.generateSchema.scripts.drop.tar</td>
<td>Name of the script file to write to if doing a &quot;drop&quot; with the target as &quot;scripts&quot;</td>
<td></td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Range of Values</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>----------------------------------------------------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td><code>datanucleus.generateSchema.scripts.create.source</code></td>
<td>Name of a script file to run to create tables. Can be absolute filename, or URL string</td>
<td><code>{filename}</code></td>
</tr>
<tr>
<td><code>datanucleus.generateSchema.scripts.drop.source</code></td>
<td>Name of a script file to run to drop tables. Can be absolute filename, or URL string</td>
<td><code>{filename}</code></td>
</tr>
<tr>
<td><code>datanucleus.generateSchema.scripts.load</code></td>
<td>Name of a script file to run to load data into the schema. Can be absolute filename, or URL string</td>
<td><code>{filename}</code></td>
</tr>
<tr>
<td><code>datanucleus.identifierFactory</code></td>
<td>Name of the identifier factory to use when generating table/column names etc. See also the <a href="#">JDO RDBMS Identifier Guide</a>.</td>
<td>datanucleus1</td>
</tr>
<tr>
<td><code>datanucleus.identifier.case</code></td>
<td>Which case to use in generated table and column names. See also the <a href="#">JDO RDBMS Identifier Guide</a>.</td>
<td><code>UpperCase</code></td>
</tr>
<tr>
<td><code>datanucleus.identifier.wordSeparator</code></td>
<td>Separator character(s) to use between words in generated identifiers. Defaults to &quot;_&quot; (underscore)</td>
<td></td>
</tr>
<tr>
<td><code>datanucleus.identifier.tablePrefix</code></td>
<td>Prefix to be prepended to all generated table names (if the identifier factory supports it)</td>
<td></td>
</tr>
<tr>
<td><code>datanucleus.identifier.tableSuffix</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Range of Values</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>datanucleus.defaultInheritanceStrategy</td>
<td>Suffix to be appended to all generated table names (if the identifier factory supports it)</td>
<td></td>
</tr>
<tr>
<td>datanucleus.store.allowReferencesWithNoImpl</td>
<td>Whether we permit a reference field (1-1 relation) or collection of references where there are no defined implementations of the reference. False means that an exception will be thrown</td>
<td>true</td>
</tr>
</tbody>
</table>

### 11.1.4 Transactions and Locking

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>datanucleus.transactionIsolation</td>
<td>Select the default transaction isolation level for ALL PM/EM factories. Some databases do not support all isolation levels, refer to your database documentation. Please refer to the transaction guides for JDO and JPA</td>
<td>read-uncommitted</td>
</tr>
<tr>
<td>datanucleus.SerializeRead</td>
<td>With datastore transactions you can apply locking to objects as they are read from the datastore. This setting applies as the default for all PM/EMs obtained. You can also specify this on a per-operation or per-query basis (which is often better to avoid deadlocks etc)</td>
<td>true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>datanucleus.jtaLocator</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### datanucleus.jtaJndiLocation

**Description**

Selects the locator to use when using JTA transactions so that DataNucleus can find the JTA TransactionManager. If this isn't specified and using JTA transactions DataNucleus will search all available locators which could have a performance impact. See [JTA Locator extension](#). If specifying "custom_jndi" please also specify "datanucleus.jtaJndiLocation".

**Range of Values**

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>jboss</td>
</tr>
</tbody>
</table>

### datanucleus.datastoreTransactionFlushLimit

**Description**

For use when using datastore transactions and is the limit on number of dirty objects before a flush to the datastore will be performed.

**Range of values**

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

### datanucleus.flush.mode

**Description**

Sets when persistence operations are flushed to the datastore. **MANUAL** means that operations will be sent only on flush()/commit(). **AUTO** means that operations will be sent immediately.

**Range of values**

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MANUAL</td>
</tr>
</tbody>
</table>

### datanucleus.flush.optimised

**Description**

Whether to use an "optimised" flush process, changing the order of persists for referential integrity (as used by RDBMS typically), or whether to just build a list of deletes, inserts and updates and do them in batches. RDBMS defaults to true, whereas other datastores default to false (due to not having referential integrity, so gaining from batching).

**Range of values**

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
</tr>
<tr>
<td>Property</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>datanucleus.connectionPoolingType</td>
</tr>
<tr>
<td>datanucleus.connectionPoolingType.nontx</td>
</tr>
<tr>
<td>datanucleus.connection.nontx.releaseAfterUse</td>
</tr>
<tr>
<td>datanucleus.connection.resourceType</td>
</tr>
<tr>
<td>datanucleus.connection.resourceType2</td>
</tr>
</tbody>
</table>
### 11.1.5 Caching

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>datanucleus.cache.collections</code></td>
<td>SCO collections can be used in 2 modes in DataNucleus. You can allow DataNucleus to cache the collections contents, or you can tell DataNucleus to access the datastore for every access of the SCO collection. The default is to use the cached collection.</td>
<td>true</td>
</tr>
<tr>
<td><code>datanucleus.cache.collections.lazy</code></td>
<td>When using cached collections/maps, the elements/keys/values can be loaded when the object is initialised, or can be loaded when accessed (lazy loading). The default is to use lazy loading when the field is not in the current fetch group, and to not use lazy loading when the field is in the current fetch group.</td>
<td>true</td>
</tr>
<tr>
<td><code>datanucleus.cache.level1.type</code></td>
<td>Name of the type of Level 1 cache to use. Defines the backing map. See also Cache docs for JDO, and for JPA</td>
<td>soft</td>
</tr>
<tr>
<td><code>datanucleus.cache.level2.type</code></td>
<td>Name of the type of Level 2 Cache to use. Can be used to interface with external caching products. Use &quot;none&quot; to turn off L2 caching. See also Cache docs for JDO, and for JPA</td>
<td>none</td>
</tr>
</tbody>
</table>

©2015, DataNucleus • ALL RIGHTS RESERVED.
### datanucleus.cache.level2.storeMode

**Description:**
The mode of operation of the L2 cache, deciding which entities are cached. The default (UNSPECIFIED) is the same as DISABLE_SELECTIVE. See also Cache docs for JDO, and for JPA.

**Range of Values:**
NONE | ALL | ENABLE_SELECTIVE | DISABLE_SELECTIVE | UNSPECIFIED

### datanucleus.cache.level2.retrieveMode

**Description:**
Whether to use the L2 cache for retrieving values (set to "bypass" to not retrieve from L2 cache within the context of the operation).

**Range of Values:**
use | bypass

### datanucleus.cache.level2.updateMode

**Description:**
When the objects in the L2 cache should be updated. Defaults to updating at commit AND when fields are read from a datastore object.

**Range of Values:**
commit-and-datastore-read | commit

### datanucleus.cache.level2.cacheName

**Description:**
Name of the cache. This is for use with plugins such as the Tangosol cache plugin for accessing the particular cache. Please refer to the Cache Guide for JDO or JPA.

**Range of Values:**
your cache name

### datanucleus.cache.level2.maxSize

**Description:**
Max size for the L2 cache (supported by weak, soft, coherence, ehcache, ehcache-classbased, javax.cache).

**Range of Values:**
-1 | integer value

### datanucleus.cache.level2.clearAtClose
### Persistence Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>datanucleus.cache.level2.clearOnClose</code></td>
<td>Whether the close of the L2 cache (when the PMF/EMF closes) should also clear out any objects from the underlying cache mechanism. By default it will clear objects out but if the user has configured an external cache product and wants to share objects across multiple PMF/EMFs then this can be set to false.</td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>datanucleus.cache.level2.batchSize</code></td>
<td>When objects are added to the L2 cache at commit they are typically batched. This property sets the max size of the batch.</td>
<td><code>100</code></td>
</tr>
<tr>
<td><code>datanucleus.cache.level2.timeout</code></td>
<td>Some caches (Cacheonix, javax.cache) allow specification of an expiration time for objects in the cache. This property is the timeout in milliseconds (will be unset meaning use cache default).</td>
<td><code>-1</code></td>
</tr>
<tr>
<td><code>datanucleus.cache.level2.readThrough</code></td>
<td>With javax.cache L2 caches you can configure the cache to allow read-through</td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>datanucleus.cache.level2.writeThrough</code></td>
<td>With javax.cache L2 caches you can configure the cache to allow write-through</td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>datanucleus.cache.level2.storeByValue</code></td>
<td>With javax.cache L2 caches you can configure the cache to store by value (as opposed to by reference)</td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>datanucleus.cache.level2.statisticsEnabled</code></td>
<td>With javax.cache L2 caches you can configure the cache to enable statistics gathering (accessible via JMX)</td>
<td></td>
</tr>
</tbody>
</table>
### Range of Values

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>datanucleus.cache.queryCompilation.type</code></td>
<td>Type of cache to use for caching of generic query compilations</td>
<td>none</td>
</tr>
<tr>
<td><code>datanucleus.cache.queryCompilationDatastore.type</code></td>
<td>Type of cache to use for caching of datastore query compilations</td>
<td>none</td>
</tr>
<tr>
<td><code>datanucleus.cache.queryResults.type</code></td>
<td>Type of cache to use for caching query results.</td>
<td>none</td>
</tr>
<tr>
<td><code>datanucleus.cache.queryResults.cacheName</code></td>
<td>Name of cache for caching the query results.</td>
<td>datanucleus-query</td>
</tr>
<tr>
<td><code>datanucleus.cache.queryResults.maxSize</code></td>
<td>Max size for the query results cache (supported by weak, soft, strong)</td>
<td>-1</td>
</tr>
</tbody>
</table>

### 11.1.6 Validation

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>datanucleus.validation.mode</code></td>
<td>Determines whether the automatic lifecycle event validation is in effect. Defaults to <code>auto</code> for JPA and <code>none</code> for JDO</td>
<td>auto</td>
</tr>
<tr>
<td><code>datanucleus.validation.group.pre-persist</code></td>
<td>The classes to validation on pre-persist callback</td>
<td></td>
</tr>
</tbody>
</table>
### Persistence Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>datanucleus.validation.group.pre-update</td>
<td>The classes to validation on pre-update callback</td>
<td></td>
</tr>
<tr>
<td>datanucleus.validation.group.pre-remove</td>
<td>The classes to validation on pre-remove callback</td>
<td></td>
</tr>
<tr>
<td>datanucleus.validation.factory</td>
<td>The validation factory to use in validation</td>
<td></td>
</tr>
</tbody>
</table>

#### 11.1.7 Value Generation

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>datanucleus.valuegeneration.transactionAttribute</td>
<td>Whether to use the PM connection or open a new connection. Only used by value generators that require a connection to the datastore.</td>
<td>New</td>
</tr>
<tr>
<td>datanucleus.valuegeneration.transactionIsolation</td>
<td>Select the default transaction isolation level for identity generation. Must have <code>datanucleus.valuegeneration.transactionAttribute</code> set to New. Some databases do not support all isolation levels, refer to your database documentation. Please refer to the transaction guides for JDO and JPA.</td>
<td>read-uncommitted</td>
</tr>
<tr>
<td>datanucleus.valuegeneration.sequence.allocationSize</td>
<td>If using JDO3.0 still and not specifying the size of your sequence, this acts as the default allocation size.</td>
<td>10</td>
</tr>
</tbody>
</table>

©2015, DataNucleus • ALL RIGHTS RESERVED.
11.1.8 MetaData

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>datanucleus.metadata.jdoFileExtension</td>
<td>Suffix for JDO MetaData files. Provides the ability to override the default suffix and also to have one PMF with one suffix and another with a different suffix, hence allowing differing persistence of the same classes using different PMF’s.</td>
<td>jdo</td>
</tr>
<tr>
<td>datanucleus.metadata.ormFileExtension</td>
<td>Suffix for ORM MetaData files. Provides the ability to override the default suffix and also to have one PMF with one suffix and another with a different suffix, hence allowing differing persistence of the same classes using different PMF’s.</td>
<td>orm</td>
</tr>
<tr>
<td>datanucleus.metadata.jdoqueryFileExtension</td>
<td>Suffix for JDO Query MetaData files. Provides the ability to override the default suffix and also to have one PMF with one suffix and another with a different suffix, hence allowing differing persistence of the same classes using different PMF’s.</td>
<td>jdoquery</td>
</tr>
<tr>
<td>datanucleus.metadata.alwaysDetachable</td>
<td>Whether to treat all classes as detachable irrespective of input metadata. See also &quot;alwaysDetachable&quot; enhancer option.</td>
<td>false</td>
</tr>
<tr>
<td>datanucleus.metadata.xml.validate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### datanucleus.metadata.xml.namespaceAware

**Description**
Whether to allow for XML namespaces in metadata files. The vast majority of sane people should not need this at all, but it's enabled by default to allow for those that do (since v3.2.3)

**Range of values**
true | false

### datanucleus.metadata.allowXML

**Description**
Whether to allow XML metadata. Turn this off if not using any, for performance. From v3.0.4 onwards

**Range of values**
true | false

### datanucleus.metadata.allowAnnotations

**Description**
Whether to allow annotations metadata. Turn this off if not using any, for performance. From v3.0.4 onwards

**Range of values**
true | false

### datanucleus.metadata.allowLoadAtRuntime

**Description**
Whether to allow load of metadata at runtime. This is intended for the situation where you are handling persistence of a persistence-unit and only want the classes explicitly specified in the persistence-unit.

**Range of values**
true | false

### datanucleus.metadata.autoregistration

**Description**
Whether to use the JDO auto-registration of metadata. Turned on by default

**Range of values**
true | false

### datanucleus.metadata.supportORM

**Description**
Whether to support "orm" mapping files. By default we use what the datastore plugin supports. This can be used to turn it off when the datastore supports it but we don't plan on using it (for performance)
### 11.1.9 Auto-Start

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>datanucleus.autoStartMechanism</code></td>
<td>How to initialise DataNucleus at startup. This allows DataNucleus to read in from some source the classes that it was persisting for this data store the previous time. &quot;XML&quot; stores the information in an XML file for this purpose. &quot;SchemaTable&quot; (only for RDBMS) stores a table in the RDBMS for this purpose. &quot;Classes&quot; looks at the property <code>datanucleus.autoStartClassNames</code> for a list of classes. &quot;MetaData&quot; looks at the property <code>datanucleus.autoStartMetaDataFiles</code> for a list of metadata files. The other option is &quot;None&quot; (start from scratch each time). Please refer to the Auto-Start Mechanism Guide for more details. The default is &quot;None&quot;.</td>
<td>None</td>
</tr>
<tr>
<td><code>datanucleus.autoStartMechanismMode</code></td>
<td>The mode of operation of the auto start mode. Currently there are 3 values. &quot;Quiet&quot; means that at startup if any errors are encountered, they are fixed quietly. &quot;Ignored&quot; means that at startup if any errors are encountered they are just ignored. &quot;Checked&quot; means that at startup if any errors are encountered they are thrown as exceptions.</td>
<td>Checked</td>
</tr>
<tr>
<td><code>datanucleus.autoStartMechanismXmlFile</code></td>
<td>Filename used for the XML file for AutoStart when using &quot;XML&quot; Auto-Start Mechanism</td>
<td></td>
</tr>
<tr>
<td><code>datanucleus.autoStartClassNames</code></td>
<td>This property specifies a list of classes (comma-separated) that are loaded at startup when using the &quot;Classes&quot; Auto-Start Mechanism.</td>
<td></td>
</tr>
<tr>
<td><code>datanucleus.autoStartMetaDataFiles</code></td>
<td>This property specifies a list of metadata files (comma-separated) that are loaded at startup when using the &quot;MetaData&quot; Auto-Start Mechanism.</td>
<td></td>
</tr>
</tbody>
</table>
## 11.1.10 Query control

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>datanucleus.query.flushBeforeExecution</td>
<td>This property can enforce a flush to the datastore of any outstanding changes just before executing all queries. If using optimistic transactions any updates are typically held back until flush/commit and so the query would otherwise not take them into account.</td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.query.useFetchPlan</td>
<td>Whether to use the FetchPlan when executing a JDOQL query. The default is to use it which means that the relevant fields of the object will be retrieved. This allows the option of just retrieving the identity columns.</td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.query.compileOptimised</td>
<td>The generic query compilation process has a simple &quot;optimiser&quot; to try to iron out potential problems in users queries. It isn’t very advanced yet, but currently will detect and try to fix a query clause like &quot;var == this&quot; (which is pointless). This will be extended in the future to handle other common situations</td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.query.jdoql.allowAll</td>
<td>javax.jdo.query.JDOQL queries are allowed by JDO only to run SELECT queries. This extension permits to bypass this limitation so that DataNucleus extension bulk &quot;update&quot; and bulk &quot;delete&quot; can be run.</td>
<td>false</td>
</tr>
<tr>
<td>datanucleus.query.sql.allowAll</td>
<td>javax.jdo.query.SQL queries are allowed by JDO2 only to run SELECT queries. This extension permits to bypass this limitation (so for example can execute stored procedures).</td>
<td>false</td>
</tr>
<tr>
<td>datanucleus.query.checkUnusedParameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>Whether to check for unused input parameters and throw an exception if found. The JDO and JPA specs require this check and is a good guide to having misnamed a parameter name in the query for example.</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Range of Values</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

### 11.1.11 Datastore Specific

Properties below here are for particular datastores only.

<table>
<thead>
<tr>
<th>datanucleus.rdbms.datastoreAdapterClassName</th>
<th>Description</th>
<th>This property allows you to supply the class name of the adapter to use for your datastore. The default is not to specify this property and DataNucleus will autodetect the datastore type and use its own internal datastore adapter classes. This allows you to override the default behaviour where there maybe is some issue with the default adapter class. <strong>Applicable for RDBMS only</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Range of Values</td>
<td>(valid class name on the CLASSPATH)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>datanucleus.rdbms.useLegacyNativeValueStrategy</th>
<th>Description</th>
<th>This property changes the process for deciding the value strategy to use when the user has selected &quot;native&quot;(JDO)/&quot;auto&quot;(JPA) to be like it was with version 3.0 and earlier, so using &quot;increment&quot; and &quot;uuid-hex&quot;. <strong>Applicable for RDBMS only</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Range of Values</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>datanucleus.rdbms.statementBatchLimit</th>
<th>Description</th>
<th>Maximum number of statements that can be batched. The default is 50 and also applies to delete of objects. Please refer to the <a href="#">Statement Batching guide</a>. <strong>Applicable for RDBMS only</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Range of Values</td>
<td>integer value (0 = no batching)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>datanucleus.rdbms.checkExistTablesOrViews</th>
<th>Description</th>
<th>Whether to check if the table/view exists. If false, it disables the automatic generation of tables that don't exist. <strong>Applicable for RDBMS only</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Range of Values</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>datanucleus.rdbms.initializeColumnInfo</th>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Range of Values</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Property</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>datanucleus.rdbms.classAdditionMaxRetries</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>The maximum number of retries when trying to find a class to persist or when validating a class. <strong>Applicable for RDBMS only</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Range of Values</strong></td>
<td>3</td>
<td>A positive integer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>datanucleus.rdbms.constraintCreateMode</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Range of Values</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>datanucleus.rdbms.uniqueConstraints.mapInverse</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Range of values</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>datanucleus.rdbms.discriminatorPerSubclassTable</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Range of values</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>datanucleus.rdbms.stringDefaultLength</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Range of Values</strong></td>
</tr>
<tr>
<td>Property</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td><strong>11 Persistence Properties</strong></td>
</tr>
<tr>
<td><strong>datanucleus.rdbms.persistEmptyStringAsNull</strong></td>
</tr>
<tr>
<td><strong>datanucleus.rdbms.query.fetchDirection</strong></td>
</tr>
<tr>
<td><strong>datanucleus.rdbms.query.resultSetType</strong></td>
</tr>
<tr>
<td><strong>datanucleus.rdbms.query.resultSetConcurrency</strong></td>
</tr>
<tr>
<td><strong>datanucleus.rdbms.oracleNlsSortOrder</strong></td>
</tr>
</tbody>
</table>
### datanucleus.rdbms.schemaTable.tableName

**Description**
Name of the table to use when using auto-start mechanism of "SchemaTable" Please refer to the JDO Auto-Start guide

**Range of Values**
NUCLEUS_TABLES | Valid table name

---

### datanucleus.rdbms.connectionProviderName

**Description**
Name of the connection provider to use to allow failover Please refer to the Failover guide

**Range of Values**
PriorityList | Name of a provider

---

### datanucleus.rdbms.connectionProviderFailOnError

**Description**
Whether to fail if an error occurs, or try to continue and log warnings

**Range of Values**
true | false

---

### datanucleus.rdbms.dynamicSchemaUpdates

**Description**
Whether to allow dynamic updates to the schema. This means that upon each insert/update the types of objects will be tested and any previously unknown implementations of interfaces will be added to the existing schema.

**Range of Values**
true | false

---

### datanucleus.rdbms.omitDatabaseMetaDataGetColumns

**Description**
Whether to bypass all calls to DatabaseMetaData.getColumns(). This JDBC method is called to get schema information, but on some JDBC drivers (e.g Derby) it can take an inordinate amount of time. Setting this to true means that your datastore schema has to be correct and no checks will be performed.

**Range of Values**
true | false

---

### datanucleus.rdbms.sqlTableNamingStrategy

**Description**
Name of the plugin to use for defining the names of the aliases of tables in SQL statements.

**Range of Values**
alpha-scheme | t-scheme
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>datanucleus.rdbms.tableColumnOrder</td>
<td>How we should order the columns in a table. The default is to put the fields of the owning class first, followed by superclasses, then subclasses. An alternative is to start from the base superclass first, working down to the owner, then the subclasses.</td>
<td>owner-first</td>
</tr>
<tr>
<td>datanucleus.rdbms.allowColumnReuse</td>
<td>This property allows you to reuse columns for more than 1 field of a class. It is false by default to protect the user from erroneously typing in a column name. Additionally, if a column is reused, the user ought to think about how to determine which field is written to that column ... all reuse ought to imply the same value in those fields so it doesn't matter which field is written there, or retrieved from there.</td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.rdbms.statementLogging</td>
<td>How to log SQL statements. The default is to log the statement and replace any parameters with the value provided in angle brackets. Alternatively you can log the statement with any parameters replaced by just the values (no brackets). The final option is to log the raw JDBC statement (with ? for parameters).</td>
<td>values-in-brackets</td>
</tr>
<tr>
<td>datanucleus.rdbms.fetchUnloadedAutomatically</td>
<td>If enabled will, upon a request to load a field, check for any unloaded fields that are non-relation fields or 1-1/N-1 fields and will load them in the same SQL call.</td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.rdbms.adapter.informixUseSerialI</td>
<td>Whether we are using SERIAL for identity columns (instead of SERIAL8).</td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.cloud.storage.bucket</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Range of Values</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>peristence.properties</td>
<td>This is a mandatory property that allows you to supply the bucket name to store your data. <strong>Applicable for Google Storage, and AmazonS3</strong></td>
<td>Any valid string</td>
</tr>
<tr>
<td>datanucleus.hbase.enforceUniquenessInApplication</td>
<td>Setting this property to true means that when a new object is persisted (and its identity is assigned), no check will be made as to whether it exists in the datastore and that the user takes responsibility for such checks. <strong>Applicable for HBase</strong></td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.cassandra.compression</td>
<td>Type of compression to use for the Cassandra cluster. <strong>Applicable for Cassandra only</strong></td>
<td>none</td>
</tr>
<tr>
<td>datanucleus.cassandra.metrics</td>
<td>Whether metrics are enabled for the Cassandra cluster. <strong>Applicable for Cassandra only</strong></td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.cassandra.ssl</td>
<td>Whether SSL is enabled for the Cassandra cluster. <strong>Applicable for Cassandra only</strong></td>
<td>true</td>
</tr>
<tr>
<td>datanucleus.cassandra.socket.readTimeoutMillis</td>
<td>Socket read timeout for the Cassandra cluster. <strong>Applicable for Cassandra only</strong></td>
<td></td>
</tr>
<tr>
<td>datanucleus.cassandra.socket.connectTimeoutMillis</td>
<td>Socket connect timeout for the Cassandra cluster. <strong>Applicable for Cassandra only</strong></td>
<td></td>
</tr>
</tbody>
</table>
12 Security

12.1 Java Security Manager

The Java Security Manager can be used with DataNucleus to provide a security platform to sensitive applications.

To use the Security Manager, specify the `java.security.manager` and `java.security.policy` arguments when starting the JVM. e.g.

```
java -Djava.security.manager -Djava.security.policy=/etc/apps/security/security.policy ...
```

Note that when you use `-Djava.security.policy==...` (double equals sign) you override the default JVM security policy files, while if you use `-Djava.security.policy=...` (single equals sign), you append the security policy file to any existing ones.

The following is a sample security policy file to be used with DataNucleus.
grant codeBase "file:${~/}jdo2-api-2.0.jar" {

    // jdo API needs datetime (timezone class needs the following)
    permission java.util.PropertyPermission "user.country", "read";
    permission java.util.PropertyPermission "user.variant", "read";
    permission java.util.PropertyPermission "user.timezone", "read,write";
    permission java.util.PropertyPermission "java.home", "read";
}
grant codeBase "file:${~/}datanucleus*.jar" {

    // jdo
    permission javax.jdo.spi.JDOPermission "getMetadata";
    permission javax.jdo.spi.JDOPermission "setStateManager";

    // DataNucleus needs to get classloader of classes
    permission java.lang.RuntimePermission "getClassLoader";

    // DataNucleus needs to detect the java and os version
    permission java.util.PropertyPermission "java.version", "read";
    permission java.util.PropertyPermission "os.name", "read";

    // DataNucleus reads these system properties
    permission java.util.PropertyPermission "datanucleus.*", "read";
    permission java.util.PropertyPermission "javax.jdo.*", "read";

    // DataNucleus runtime enhancement (needs read access to all jars/classes in classpath,
    // so use <<ALL FILES>> to facilitate config)
    permission java.lang.RuntimePermission "createClassLoader";
    permission java.io.FilePermission "<<ALL FILES>>", "read";

    // DataNucleus needs to read manifest files (read permission to location of MANIFEST.MF files)
    permission java.io.FilePermission "${user.dir}/${~}", "read";
    permission java.io.FilePermission "<<ALL FILES>>", "read";

    // DataNucleus uses reflection!!!
    permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
    permission java.lang.RuntimePermission "accessDeclaredMembers";
}
grant codeBase "file:${~/}datanucleus-hbase*.jar" {

    // HBASE does not run in a doPrivileged, so we do...
    permission java.net.SocketPermission "*", "connect,resolve";
}
13 Logging

13.1 DataNucleus Logging
DataNucleus can be configured to log significant amounts of information regarding its process. This information can be very useful in tracking the persistence process, and particularly if you have problems. DataNucleus will log as follows:

- **Log4J** - if you have Log4J in the CLASSPATH, Apache Log4J will be used
- **java.util.logging** - if you don't have Log4J in the CLASSPATH, then java.util.logging will be used

DataNucleus logs messages to various categories (in Log4J and java.util.logging these correspond to a "Logger"), allowing you to filter the logged messages by these categories - so if you are only interested in a particular category you can effectively turn the others off. DataNucleus's log is written by default in English. If your JDK is running in a Spanish locale then your log will be written in Spanish. **If you have time to translate our log messages into other languages, please contact one of the developers via the Online Forum.**

13.1.1 Logging Categories
DataNucleus uses a series of categories, and logs all messages to these categories. Currently DataNucleus uses the following

- **DataNucleus.Persistence** - All messages relating to the persistence process
- **DataNucleus.Transaction** - All messages relating to transactions
- **DataNucleus.Connection** - All messages relating to Connections.
- **DataNucleus.Query** - All messages relating to queries
- **DataNucleus.Cache** - All messages relating to the DataNucleus Cache
- **DataNucleus.MetaData** - All messages relating to MetaData
- **DataNucleus.Datastore** - All general datastore messages
- **DataNucleus.Datastore.Schema** - All schema related datastore log messages
- **DataNucleus.DatastorePersist** - All datastore persistence messages
- **DataNucleus.DatastoreRetrieve** - All datastore retrieval messages
- **DataNucleus.DatastoreNative** - Log of all 'native' statements sent to the datastore
- **DataNucleus.General** - All general operational messages
- **DataNucleus.Lifecycle** - All messages relating to object lifecycle changes
- **DataNucleus.ValueGeneration** - All messages relating to value generation
- **DataNucleus.Enhancer** - All messages from the DataNucleus Enhancer.
- **DataNucleus.SchemaTool** - All messages from DataNucleus SchemaTool
- **DataNucleus.JDO** - All messages general to JDO
- **DataNucleus.JPA** - All messages general to JPA
- **DataNucleus.JCA** - All messages relating to Connector JCA.
- **DataNucleus.IDE** - Messages from the DataNucleus IDE.

13.1.2 Using Log4J
Log4J allows logging messages at various severity levels. The levels used by Log4J, and by DataNucleus's use of Log4J are **DEBUG, INFO, WARN, ERROR, FATAL**. Each message is logged at a particular level to a category (as described above). The other setting is **OFF** which turns
off a logging category. This is very useful in a production situation where maximum performance is required.

To enable the DataNucleus log, you need to provide a Log4J configuration file when starting up your application. This may be done for you if you are running within a JavaEE application server (check your manual for details). If you are starting your application yourself, you would set a JVM parameter as

```
-Dlog4j.configuration=file:log4j.properties
```

where log4j.properties is the name of your Log4J configuration file. Please note the "file:" prefix to the file since a URL is expected. [When using java.util.logging you need to specify the system property "java.util.logging.config.file"]

The Log4J configuration file is very simple in nature, and you typically define where the log goes to (e.g to a file), and which logging level messages you want to see. Here’s an example

```
# Define the destination and format of our logging
log4j.appender.A1.File=datanucleus.log
log4j.appender.A1.layout.ConversionPattern=%d{HH:mm:ss,SSS} (%t) %-5p [%c] - %m%n

# DataNucleus Categories
log4j.category.DataNucleus.JDO=INFO, A1
log4j.category.DataNucleus.Cache=INFO, A1
log4j.category.DataNucleus.MetaData=INFO, A1
log4j.category.DataNucleus.MetaData=INFO, A1
log4j.category.DataNucleus.Transaction=INFO, A1
log4j.category.DataNucleus.Datastore=DEBUG, A1
log4j.category.DataNucleus.ValueGeneration=DEBUG, A1
log4j.category.DataNucleus.Enhancer=INFO, A1
log4j.category.DataNucleus.SchemaTool=INFO, A1
```

In this example, I am directing my log to a file (datanucleus.log). I have defined a particular "pattern" for the messages that appear in the log (to contain the date, level, category, and the message itself). In addition I have assigned a level "threshold" for each of the DataNucleus categories. So in this case I want to see all messages down to DEBUG level for the DataNucleus RDBMS persister.

**Performance Tip**: Turning OFF the logging, or at least down to ERROR level provides a significant improvement in performance. With Log4J you do this via

```
log4j.category.DataNucleus=OFF
```

### 13.1.3 Using java.util.logging

`java.util.logging` allows logging messages at various severity levels. The levels used by `java.util.logging`, and by DataNucleus's internally are **fine, info, warn, severe**. Each message is logged at a particular level to a **category** (as described above).

By default, the `java.util.logging` configuration is taken from a properties file `<JRE_DIRECTORY>/lib/logging.properties`. Modify this file and configure the categories to be logged, or use the
java.util.logging.config.file system property to specify a properties file (in java.util.Properties format) where the logging configuration will be read from. Here is an example:

```java
handlers=java.util.logging.FileHandler, java.util.logging.ConsoleHandler
DataNucleus.General.level=fine
DataNucleus.JDO.level=fine

# --- ConsoleHandler ---
# Override of global logging level
java.util.logging.ConsoleHandler.level=SEVERE
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter

# --- FileHandler ---
# Override of global logging level
java.util.logging.FileHandler.level=SEVERE

# Naming style for the output file:
java.util.logging.FileHandler.pattern=datanucleus.log

# Limiting size of output file in bytes:
java.util.logging.FileHandler.limit=50000

# Number of output files to cycle through, by appending an integer to the base file name:
java.util.logging.FileHandler.count=1

# Style of output (Simple or XML):
java.util.logging.FileHandler.formatter=java.util.logging.SimpleFormatter
```

Please read the javadocs for java.util.logging for additional details on its configuration.

**13.1.4 Sample Log Output**

Here is a sample of the type of information you may see in the DataNucleus log when using Log4J.
So you see the time of the log message, the level of the message (DEBUG, INFO, etc), the category (DataNucleus.Datastore, etc), and the message itself. So, for example, if I had set the DataNucleus.Datastore.Schema to DEBUG and all other categories to INFO I would see *all* DDL statements sent to the database and very little else.

13.1.5 HOWTO : Log with log4j and DataNucleus under OSGi

This guide was provided by Marco Lopes, when using DataNucleus v2.2. All the bundles which use log4j should have org.apache.log4j in their Import-Package attribute! (use: org.apache.log4j;resolution:=optional if you don't want to be stuck with log4j whenever you use an edited bundle in your project!).

Method 1

- Create a new "Fragment Project". Call it whatever you want (ex: log4j-fragment)
- You have to define a "Plugin-ID", that's the plugin where DN will run
- Edit the MANIFEST
- Under RUNTIME add log4j JAR to the Classpath
- Under Export-Packages add org.apache.log4j
- Save MANIFEST
- PASTE the log4j PROPERTIES file into the SRC FOLDER of the Project

Method 2

- Get an "OSGI Compliant" log4j bundle (you can get it from the SpringSource Enterprise Bundle Repository at [http://ebr.springsource.com/repository/app/])
- Open the Bundle JAR with WINRAR (others might work)
- PASTE the log4j PROPERTIES file into the ROOT of the bundle
• Exit. Winrar will ask to UPDATE the JAR. Say YES.
• Add the updated OSGI compliant Log4j bundle to your Plugin Project Dependencies (Required-Plugins)

Each method has its own advantages. Use method 1 if you need to EDIT the log4j properties file ON-THE-RUN. The disadvantage: it can only "target" one project at a time (but very easy to edit the MANIFEST and select a new Host Plugin!). Use method 2 if you want to have log4j support in every project with only one file. The disadvantage: it's not very practical to edit the log4j PROPERTIES file (not because of the bundle EDIT, but because you have to restart eclipse in order for the new bundle to be recognized).
14 Datastore

14.1 Datastores

The DataNucleus AccessPlatform is designed for flexibility to operate with any type of datastore. We already support a very wide range of datastores and this will only increase in the future. In this section you can find the specifics for particular supported datastores over and above what was already addressed for JDO and JPA persistence.

- **Relational**
  - **RDBMS**: tried and tested since the 1970s, relational databases form an integral component of many systems. They incorporate optimised querying mechanisms, yet also can suffer from object-relational impedance mismatch in some situations. They also require an extra level of configuration to map from objects across to relational tables/columns.

- **Documents**
  - **Open Document Format (ODF)**: ODF is an international standard document format, and its spreadsheets provide a widely used form for publishing of data, making it available to other groups.
  - **Excel (XLS)**: Excel spreadsheets provide a widely used format allowing publishing of data, making it available to other groups (XLS format).
  - **Excel (OOXML)**: Excel spreadsheets provide a widely used format allowing publishing of data, making it available to other groups (OOXML format).
  - **XML**: XML defines a document format and, as such, is a key data transfer medium.

- **Map-Based**
  - **HBase**: HBase is a map-based datastore originated within Hadoop, following the model of BigTable.
  - **Cassandra**: Cassandra is a distributed robust clustered datastore. This plugin is a development work-in-progress needing volunteers, and will be released for DataNucleus AccessPlatform v4.0
  - **Google AppEngine/Datastore**: Google AppEngine provides persistence to its own BigTable-based Datastore using a DataNucleus plugin, and its v3.0 release is designed to work with DataNucleus 3.2. At the time of writing, Google still hasn’t released v3.0 of their plugin and continues to release v1.0 as the default plugin; use that and everything is your problem. There is a built v3.0 of their plugin in the DataNucleus Maven repo

- **Document-Based**
  - **MongoDB**: plugin providing persistence to the MongoDB NoSQL datastore
  - **OrientDB**: plugin providing persistence to the OrientDB datastore. Works with DataNucleus 2.x

- **Graph-Based**
  - **Neo4j**: plugin providing persistence to the Neo4j graph store

- **Web-Based**
- **JSON**: another format of document for exchange, in this case with particular reference to web contexts.
- **Amazon S3**: Amazon Simple Storage Service.
- **Google Storage**: Google Storage.

**Object-Based**

- **NeoDatis**: an open source object datastore. This provides fast persistence of large object graphs, without the necessity of any object-relational mapping.
- **DB4O**: an open source object datastore. DataNucleus (upto and including v3.0) provided a plugin for db4o. This is now discontinued. Uses DataNucleus v3.0 if you wish to use this, or look in our SVN for the plugin (no longer developed, so may not run against latest code)

**Other**

- **LDAP**: an internet standard datastore for indexed data that is not changing significantly.
- **VMForce**:

  plugin providing persistence to the Force.com datastore. Works with DataNucleus 2.x. You can get the source code for this from here

If you have a requirement for persistence to some other datastore, then it would likely be easily provided by creation of a DataNucleus StoreManager. Please contact us via the forum so that you can provide this and contribute it back to the community.
### 15 Supported Features

#### 15.1 Datastore Feature Support

Whilst we aim to ultimately support all API features on all supported datastores, this isn't currently possible. See below for a summary of what feature is supported on which datastore.

<table>
<thead>
<tr>
<th>Feature</th>
<th>RDBMS</th>
<th>ODF</th>
<th>Excel</th>
<th>OOXML</th>
<th>XML</th>
<th>HBase</th>
<th>MongoDB</th>
<th>GAE</th>
<th>JSON</th>
<th>AmazonS3</th>
<th>G.Storage</th>
<th>LDAP</th>
<th>Neo4j</th>
<th>NeoDatis</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Feature</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dataset Identity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application Identity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nondurable Identity</td>
<td>[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compound Identity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Notification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACID Transactions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Versioning objects</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optimistic Checks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetch Plan control</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Native Connection access (JDO)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encryption of data</td>
<td>[7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Backed object wrapper</td>
<td>[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feature</td>
<td>JDO</td>
<td>JPA</td>
<td>Comparison</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>-----------</td>
<td>-----------</td>
<td>------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Persist</td>
<td>!</td>
<td>!</td>
<td>!</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>!</td>
<td>!</td>
<td>!</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>!</td>
<td>!</td>
<td>!</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>New fields</td>
<td>![]</td>
<td>![]</td>
<td>![]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value Generation</td>
<td>!</td>
<td>!</td>
<td>!</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>native</td>
<td>![]</td>
<td>![]</td>
<td>![]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>auto(JDBC)</td>
<td>![]</td>
<td>![]</td>
<td>![]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>increment</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>table(JDBC)</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>identity</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JPA</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sequencer</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JPA</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uuid</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uuid-hex</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uuid-string</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>persist</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>max</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indexes</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unique Keys</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Foreign Keys</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Primary Keys</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inheritance</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2015, DataNucleus • ALL RIGHTS RESERVED.
### Supported Features

<table>
<thead>
<tr>
<th>Supported Feature</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java.io</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java.util</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arrays</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interfac</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type Converter</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type Converter Auto-apply</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Queries</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JDOQL Evalu at   in mem or</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JDOQL Evalu at   in data st</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JDOQL of candid. interfac</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JDOQL Polymq queries</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JPQL Evalu at   in mem or</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JPQL Evalu at   in data st</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stored Proced</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

©2015, DataNucleus • ALL RIGHTS RESERVED.
[1] represents partial implementation.

[2] - when a collection/map/array is "backed" it can put individual elements in the datastore at once rather than writing everything, and additionally can control how the elements are retrieved


[4] means that datastore doesn't explicitly support inheritance but "complete-table" is the nearest to what happens.


[7] Using Cumulus4j plugin for DataNucleus
16 RDBMS

16.1 RDBMS Datastores

MySQL
MariaDB
Microsoft SQL Server
ORACLE
Sybase
HSQL database engine
H2
PostgreSQL
PostGIS

© 2015, DataNucleus • ALL RIGHTS RESERVED.
DataNucleus supports persisting objects to RDBMS datastores (using the datanucleus-rdbms plugin). It supports the vast majority of RDBMS products available today. DataNucleus communicates with the RDBMS datastore using JDBC. RDBMS systems accept varying standards of SQL and so
DataNucleus will support particular RDBMS/JDBC combinations only, though clearly we try to support as many as possible.

The jars required to use DataNucleus RDBMS persistence are datanucleus-core, datanucleus-api-jdo, datanucleus-api-jpa, datanucleus-rdbms and JDBC driver.

There are tutorials available for use of DataNucleus with RDBMS for JDO and for JPA

By default when you create a PersistenceManagerFactory or EntityManagerFactory to connect to a particular datastore DataNucleus will automatically detect the datastore adapter to use and will use its own internal adapter for that type of datastore. If you find that either DataNucleus has incorrectly detected the adapter to use, you can override the default behaviour using the persistence property datanucleus.rdbms.datastoreAdapterClassName.

The following RDBMS have support built in to DataNucleus. Click on the one of interest to see details of any provisos for its support, as well as the JDBC connection information

- MySQL/MariaDB
- PostgreSQL Database
- PostgreSQL+PostGIS Database
- HSQL DB
- H2 Database
- SQLite
- Apache Derby
- Microsoft SQLServer
- Sybase
- Oracle
- IBM DB2
- IBM Informix
- Firebird
- NuoDB
- SAPDB/MaxDB
- Virtuoso
- Pointbase
- Oracle TimesTen
- McKoi database

Note that if your RDBMS is not listed or currently supported you can easily write your own Datastore Adapter for it raise an issue in DataNucleus JIRA when you have it working and attach a patch to contribute it. Similarly if you are using an adapter that has some problem on your case you could use the same plugin mechanism to override the non-working feature.

16.1.1 DB2

To specify DB2 as your datastore, you will need something like the following specifying (where "mydb1" is the name of the database)
With DB2 Express-C v9.7 you need to have db2jcc.jar and db2jcc_license_cu.jar in the CLASSPATH.

16.1.2 MySQL

MySQL and its more developed drop in replacement MariaDB are supported as an RDBMS datastore by DataNucleus with the following provisos

- **INNODB** tables must be used since it is the only table type that allows foreign keys etc at the moment. You can however define what type your table uses by setting the `<class>` extension "mysql-engine-type" to be MyISAM or whatever for the class being persisted.
- JDOQL.isEmpty()/contains() will not work in MySQL 4.0 (or earlier) since the query uses EXISTS and that is only available from MySQL 4.1.
- MySQL on Windows MUST specify `datanucleus.identifier.case` as "LowerCase" since the MySQL server stores all identifiers in lowercase BUT the mysql-connector-java JDBC driver has a bug (in versions up to and including 3.1.10) where it claims that the MySQL server stores things in mixed case when it doesn't.
- MySQL 3.* will not work reliably with inheritance cases since DataNucleus requires UNION and this doesn't exist in MySQL 3.*
- MySQL before version 4.1 will not work correctly on JDOQL Collection.size(), Map.size() operations since this requires subqueries, which are not supported before MySQL 4.1.
- If you receive an error "Incorrect arguments to mysql_stmt_execute" then this is a bug in MySQL and you need to update your JDBC URL to append "?useServerPrepStmts=false".
- MySQL throws away the milliseconds on a Date and so cannot be used reliably for Optimistic locking using strategy "date-time" (use "version" instead)
- You can specify "BLOB", "CLOB" JDBC types when using MySQL with DataNucleus but you must turn validation of columns OFF. This is because these types are not supported by the MySQL JDBC driver and it returns them as LONGVARBINARY/LONGVARCHAR when querying the column type

To specify MySQL as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```java
datanucleus.ConnectionDriverName=com.mysql.jdbc.Driver
datanucleus.ConnectionURL=jdbc:mysql://'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

16.1.3 MS SQL Server

Microsoft SQL Server is supported as an RDBMS datastore by DataNucleus with the following proviso
• MS SQL 2000 does not keep accuracy on `datetime` datatypes. This is an MS SQL 2000 issue. In order to keep the accuracy when storing `java.util.Date` java types, use `int` datatype.

To specify MS SQL as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

**Microsoft SqlServer 2005 JDBC Driver (Recommended)**

```java
datanucleus.ConnectionDriverName=com.microsoft.sqlserver.jdbc.SQLServerDriver
datanucleus.ConnectionURL=jdbc:sqlserver://'host':'port';DatabaseName='db-name'
;SelectMethod=cursor
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

**Microsoft SqlServer 2000 JDBC Driver**

```java
datanucleus.ConnectionDriverName=com.microsoft.jdbc.sqlserver.SQLServerDriver
datanucleus.ConnectionURL=jdbc:microsoft:sqlserver://'host':'port';DatabaseName='db-name'
;SelectMethod=cursor
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

### 16.1.4 Oracle

To specify **Oracle** as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc) ... you can also use 'oci' instead of 'thin' depending on your driver.

```java
datanucleus.ConnectionDriverName=oracle.jdbc.driver.OracleDriver
datanucleus.ConnectionURL=jdbc:oracle:thin:@'host':'port':'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

### 16.1.5 Sybase

To specify **Sybase** as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```java
datanucleus.ConnectionDriverName=com.sybase.jdbc2.jdbc.SybDriver
datanucleus.ConnectionURL=jdbc:sybase:Tds:'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```
16.1.6 HSQLDB

HSQLDB is supported as an RDBMS datastore by DataNucleus with the following proviso:

- Use of batched statements is disabled since HSQLDB has a bug where it throws exceptions "batch failed" (really informative). Still waiting for this to be fixed in HSQLDB.
- Use of JDOQL/JPQL subqueries cannot be used where you want to refer back to the parent query since HSQLDB up to and including version 1.8 don't support this.

To specify HSQL (1.x) as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc):

```java
datanucleus.ConnectionDriverName=org.hsqldb.jdbcDriver
datanucleus.ConnectionURL=jdbc:hsqldb:hsql://'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

Note that in HSQLDB v2.x the driver changes to `org.hsqldb.jdbc.JDBCDriver`.

16.1.7 H2

H2 is supported as an RDBMS datastore by DataNucleus.

To specify H2 as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc):

```java
datanucleus.ConnectionDriverName=org.h2.Driver
datanucleus.ConnectionURL=jdbc:h2:'db-name'
datanucleus.ConnectionUserName=sa
datanucleus.ConnectionPassword=
```

16.1.8 Informix

Informix is supported as an RDBMS datastore by DataNucleus.

To specify Informix as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc):

```java
datanucleus.ConnectionDriverName=com.informix.jdbc.IfxDriver
datanucleus.ConnectionURL=jdbc:informix-sqli://[ip|host]:port[/dbname]:
    INFORMIXSERVER=servername[name=value;name=value]...]
datanucleus.ConnectionUserName=informix
datanucleus.ConnectionPassword=password
```

e.g.
Note that some database logging options in Informix do not allow changing autoCommit dynamically. You need to rebuild the database to support it. To rebuild the database refer to Informix documentation, but as example, run $INFORMIXDIR\bin\dbaccess and execute the command "CREATE DATABASE mydb WITH BUFFERED LOG".

**INDEXOF**: Informix 11.x does not have a function to search a string in another string. DataNucleus defines a user defined function, DATANUCLEUS_STRPOS, which is automatically created on startup. The SQL for the UDF function is:

```sql
create function DATANUCLEUS_STRPOS(str char(40),search char(40),from smallint) returning smallint
    define i,pos,lenstr,lensearch smallint;
    let lensearch = length(search);
    let lenstr = length(str);
    if lenstr=0 or lensearch=0 then return 0; end if;
    let pos=-1;
    for i=1+from to lenstr
        if substr(str,i,lensearch)=search then
            let pos=i;
            exit for;
        end if;
    end for;
    return pos;
end function;
```

### 16.1.9 PostgreSQL

To specify PostgreSQL as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```sql
datanucleus.ConnectionDriverName=org.postgresql.Driver
datanucleus.ConnectionURL=jdbc:postgresql://host:port/db-name
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```
16.1.10 PostgreSQL with PostGIS extension

To specify PostGIS as your datastore, you will need to decide first which geometry library you want to use and then set the connection url accordingly.

For the PostGIS JDBC geometries you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```java
datanucleus.ConnectionDriverName=org.postgresql.Driver
datanucleus.ConnectionURL=jdbc:postgresql://'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

For Oracle's JGeometry you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```java
datanucleus.ConnectionDriverName=org.postgresql.Driver
datanucleus.ConnectionURL=jdbc:postgres_jgeom://'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

For the JTS (Java Topology Suite) geometries you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```java
datanucleus.ConnectionDriverName=org.postgresql.Driver
datanucleus.ConnectionURL=jdbc:postgres_jts://'host':'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

16.1.11 Apache Derby

To specify Apache Derby as your datastore, you will need something like the following specifying (replacing 'db-name' with filename of your database etc)

```java
datanucleus.ConnectionDriverName=org.apache.derby.jdbc.EmbeddedDriver
datanucleus.ConnectionURL=jdbc:derby:'db-name';create=true
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

Above settings are used together with the Apache Derby in embedded mode. The below settings are used in network mode, where the default port number is 1527.
ASCII: Derby 10.1 does not have a function to convert a char into ascii code. DataNucleus needs such function to converts chars to int values when performing queries converting chars to ints. DataNucleus defines a user defined function, DataNucleus_ASCII, which is automatically created on startup. The SQL for the UDF function is:

```
DROP FUNCTION NUCLEUS_ASCII;
CREATE FUNCTION NUCLEUS_ASCII(C CHAR(1)) RETURNS INTEGER
EXTERNAL NAME 'org.datanucleus.store.rdbms.adapter.DerbySQLFunction.ascii'
CALLED ON NULL INPUT
LANGUAGE JAVA PARAMETER STYLE JAVA;
```

String.matches(pattern): When pattern argument is a column, DataNucleus defines a function that allows Derby 10.1 to perform the matches function. The SQL for the UDF function is:

```
DROP FUNCTION NUCLEUS_MATCHES;
CREATE FUNCTION NUCLEUS_MATCHES(TEXT VARCHAR(8000), PATTERN VARCHAR(8000)) RETURNS INTEGER
EXTERNAL NAME 'org.datanucleus.store.rdbms.adapter.DerbySQLFunction.matches'
CALLED ON NULL INPUT
LANGUAGE JAVA PARAMETER STYLE JAVA;
```

16.1.12 Firebird

Firebird is supported as an RDBMS datastore by DataNucleus with the proviso that

- Auto-table creation is severely limited with Firebird. In Firebird, DDL statements are not auto-committed and are executed at the end of a transaction, after any DML statements. This makes "on the fly" table creation in the middle of a DML transaction not work. You must make sure that "autoStartMechanism" is NOT set to "SchemaTable" since this will use DML. You must also make sure that nobody else is connected to the database at the same time. Don't ask us why such limitations are in a RDBMS, but then it was you that chose to use it ;-)

To specify Firebird as your datastore, you will need something like the following specifying (replacing 'db-name' with filename of your database etc)

```
datanucleus.ConnectionDriverName=org.firebirdsql.jdbc.FBDriver
datanucleus.ConnectionURL=jdbc:firebirdsql://localhost/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```
16.1.13 NuoDB
To specify NuoDB as your datastore, you will need something like the following specifying (replacing 'db-name' with filename of your database etc)

```java
datanucleus.ConnectionDriverName=com.nuodb.jdbc.Driver
datanucleus.ConnectionURL=jdbc:com.nuodb://localhost/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
datanucleus.Schema={my-schema-name}
```

16.1.14 SAPDB/MaxDB
To specify SAPDB/MaxDB as your datastore, you will need something like the following specifying (replacing 'db-name' with filename of your database etc)

```java
datanucleus.ConnectionDriverName=com.sap.dbtech.jdbc.DriverSapDB
datanucleus.ConnectionURL=jdbc:sapdb://localhost/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

16.1.15 SQLite
SQLite is supported as an RDBMS datastore by DataNucleus with the proviso that

- When using sequences, you must set the persistence property `datanucleus.valuegeneration.transactionAttribute` to `UsePM`

To specify SQLite as your datastore, you will need something like the following specifying (replacing 'db-name' with filename of your database etc)

```java
datanucleus.ConnectionDriverName=org.sqlite.JDBC
datanucleus.ConnectionURL=jdbc:sqlite:'db-name'
datanucleus.ConnectionUserName=
datanucleus.ConnectionPassword=
```

16.1.16 Virtuoso
To specify Virtuoso as your datastore, you will need something like the following specifying (replacing 'db-name' with filename of your database etc)
16.1.17 Pointbase

To specify **Pointbase** as your datastore, you will need something like the following specifying (replacing 'db-name' with filename of your database etc)

```java
datanucleus.ConnectionDriverName=com.pointbase.jdbc.jdbcUniversalDriver
datanucleus.ConnectionURL=jdbc:pointbase://127.0.0.1/{dbname}
datanucleus.ConnectionUserName=
datanucleus.ConnectionPassword=
```

16.1.18 McKoi

**McKoi** is supported as an RDBMS datastore by DataNucleus with the following proviso

- McKoi doesn't provide full information to allow correct validation of tables/constraints.

To specify McKoi as your datastore, you will need something like the following specifying (replacing 'db-name' with name of your database etc)

```java
datanucleus.ConnectionDriverName=com.mckoi.JDBCDriver
datanucleus.ConnectionURL=jdbc:mckoi://host:'port'/'db-name'
datanucleus.ConnectionUserName='user-name'
datanucleus.ConnectionPassword='password'
```

16.1.19 JDBC Driver parameters

If you need to pass additional parameters to the JDBC driver you can append these to the end of the `datanucleus.ConnectionURL`. For example,

```java
```
17 Java Types (Spatial)

17.1 RDBMS: Spatial Types Support

DataNucleus supports by default a large number of Java types. DataNucleus Spatial supports the storage and query of a number of different spatial data types, like points, polygons or lines. Spatial types like these are used to store geographic information like locations, rivers, cities, roads, etc. The datanucleus-geospatial plugin allows using spatial and traditional types simultaneously in persistent objects making DataNucleus a single interface to read and manipulate any business data.

The table below shows the currently supported Spatial SCO Java types in DataNucleus.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>oracle.spatial.geospatial</td>
<td>[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>datanucleus-geospatial</td>
</tr>
<tr>
<td>org.postgis.LinearRing</td>
<td>[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>datanucleus-geospatial</td>
</tr>
</tbody>
</table>
Dirty check mechanism is limited to immutable mode, it means, if you change a field of one of these spatial objects, you must reassign it to the owner object field to make sure changes are propagated to the database.

The implementation of these spatial types follows the OGC Simple Feature specification, but adds further types where the datastores support them.

17.1.1 Mapping Scenarios

DataNucleus supports different combinations of geometry libraries and spatially enabled databases. These combinations are called mapping scenarios. Each of these scenarios has a different set of advantages (and drawbacks), some have restrictions that apply. The table below tries to give as much information as possible about the different scenarios.

One such mapping scenario, is to use the Java geometry types from JTS (Java Topology Suite) and PostGIS as datastore. The short name for this mapping scenario is **jts2postgis**. The following table lists all supported mapping scenarios.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle’s JGeometry</td>
<td>jgeom2mysql</td>
<td>jgeom2oracle</td>
<td>x</td>
</tr>
<tr>
<td>Java Topology Suite (JTS)</td>
<td>jts2mysql</td>
<td>jts2oracle</td>
<td>jts2postgis</td>
</tr>
<tr>
<td>PostGIS JDBC Geometries</td>
<td>pg2mysql</td>
<td>x</td>
<td>pg2postgis</td>
</tr>
</tbody>
</table>

- **[1]** - MySQL doesn't support 3-dimensional geometries. Trying to persist them anyway results in **undefined behaviour**, there may be an exception thrown or the z-ordinate might just get stripped.
- **[2]** - Oracle Spatial supports additional data types like circles and curves that are not defined in the OGC SF specification. Any attempt to read or persist one of those data types, if you're not using jgeom2oracle, will result in failure!
- **[3]** - PostGIS added support for curves in version 1.2.0, but at the moment the JDBC driver doesn't support them yet. Any attempt to read curves geometries will result in failure, for every mapping scenario!
• [4] - Both PostGIS and Oracle have a system to add user data to specific points of a geometry. In PostGIS these types are called measure types and the z-coordinate of every 2d-point can be used to store arbitrary (numeric) data of double precision associated with that point. In Oracle this user data is called LRS. DataNucleus-Spatial tries to handle these types as gracefully as possible. But the recommendation is to not use them, unless you have a mapping scenario that is known to support them, i.e. pg2postgis for PostGIS and jgeom2oracle for Oracle.

• [5] - PostGIS supports two additional types called box2d and box3d, that are not defined in OGC SF. There are only mappings available for these types in pg2postgis, any attempt to read or persist one of those data types in another mapping scenario will result in failure!

17.1.2 Spatial types

This table lists all the spatial Java types that are currently supported. The JDBC type is the same for every Java type in a given database. It’s SDO_GEOMETRY for Oracle, OTHER for PostGIS and BINARY for MySQL. When a type is supported by a database, the column type that is used for it, is listed after the icon. None of the types are proxied, this means that if you change an object field, you must reassign it to the owner object field to make sure changes are propagated to the database.

<table>
<thead>
<tr>
<th>Geometry Library</th>
<th>Java Type</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostGIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>JGeometry</td>
<td>oracle.spatial.geometry</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>JTS</td>
<td>com.vividsolutions.jts:</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>JTS</td>
<td>com.vividsolutions.jts:</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>JTS</td>
<td>com.vividsolutions.jts:</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>JTS</td>
<td>com.vividsolutions.jts:</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>JTS</td>
<td>com.vividsolutions.jts:</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>JTS</td>
<td>com.vividsolutions.jts:</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>JTS</td>
<td>com.vividsolutions.jts:</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>JTS</td>
<td>com.vividsolutions.jts:</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>PostGIS-JDBC</td>
<td>org.postgis.Geometry</td>
<td>✔️</td>
<td>✗</td>
<td>✔️</td>
</tr>
</tbody>
</table>
### 17.1.3 Metadata

DataNucleus-Spatial has defined some metadata extensions that can be used to give additional information about the geometry types in use. The position of these tags in the meta-data determines their scope. If you use them inside a `<field>`-tag the values are only used for that field specifically, if you use them inside the `<package>`-tag the values are in effect for all (geometry) fields of all classes inside that package, etc.
17.1.4 Querying Spatial types

DataNucleus-Spatial defines a set of functions that can be applied to spatial types in JDOQL queries. These functions follow the definitions in the OGC Simple Feature specification and are translated into appropriate SQL statements, provided the underlying database system implements the functions and the geometry object model accordingly. There are also some additional functions that are not defined OGC SF, most of them are database specific.

This set of more than eighty functions contains:

- Basic methods on geometry objects like `IsSimple()` and `Boundary()`.
- Methods for testing spatial relations between geometric objects like `Intersects()` and `Touches()`.
- Methods that support spatial analysis like `Union()` and `Difference()`.
- Methods to create geometries from WKB/WKT (Well Known Binary/Text) like `GeomFromText()` and `GeomFromWKB()`.

For a complete list of all supported functions and usage examples, please see JDOQL: Spatial Methods.
17.1.5 Dependencies
Depending on the mapping scenario you want to use, there is a different set of JARs that need to be in your classpath.
18 Datastore Types

18.1 RDBMS: Datastore Types

As we saw in the Types Guide, DataNucleus supports the persistence of a large range of Java field types. With RDBMS datastores, we have the notion of tables/columns in the datastore and so each Java type is mapped across to a column or a set of columns in a table. It is important to understand this mapping when mapping to an existing schema for example. In RDBMS datastores a java type is stored using JDBC types. DataNucleus supports the use of the vast majority of the available JDBC types.

18.1.1 JDBC types used when persisting Java types

When persisting a Java type in general it is persisted into a single column. For example a String will be persisted into a VARCHAR column by default. Some types (e.g. Color) have more information to store than we can conveniently persist into a single column and so use multiple columns. Other types (e.g. Collection) store their information in other ways, such as foreign keys.

This table shows the Java types we saw earlier and whether they can be queried using JDOQL queries, and what JDBC types can be used to store them in your RDBMS datastore. Not all RDBMS datastores support all of these options. While DataNucleus always tries to provide a complete list sometimes this is impossible due to limitations in the underlying JDBC driver.

<table>
<thead>
<tr>
<th>Java Type</th>
<th>Number of Columns</th>
<th>Queryable</th>
<th>JDBC Type(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1</td>
<td></td>
<td>BIT, CHAR (‘Y’, ‘N’), BOOLEAN, TINYINT, SMALLINT, NUMERIC</td>
</tr>
<tr>
<td>byte</td>
<td>1</td>
<td></td>
<td>TINYINT, SMALLINT, NUMERIC</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td></td>
<td>CHAR, INTEGER, NUMERIC</td>
</tr>
<tr>
<td>double</td>
<td>1</td>
<td></td>
<td>DOUBLE, DECIMAL, FLOAT</td>
</tr>
<tr>
<td>float</td>
<td>1</td>
<td></td>
<td>FLOAT, REAL, DOUBLE, DECIMAL</td>
</tr>
<tr>
<td>int</td>
<td>1</td>
<td></td>
<td>INTEGER, BIGINT, NUMERIC</td>
</tr>
<tr>
<td>Variable Type</td>
<td>Length</td>
<td>Data Types</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>--------</td>
<td>------------</td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>1</td>
<td>BIGINT, NUMERIC, DOUBLE, DECIMAL, INTEGER</td>
<td></td>
</tr>
<tr>
<td>short</td>
<td>1</td>
<td>SMALLINT, INTEGER, NUMERIC</td>
<td></td>
</tr>
<tr>
<td>boolean[]</td>
<td>1 [5]</td>
<td>LONGVARBINARY, BLOB</td>
<td></td>
</tr>
<tr>
<td>byte[]</td>
<td>1 [5]</td>
<td>LONGVARBINARY, BLOB</td>
<td></td>
</tr>
<tr>
<td>char[]</td>
<td>1 [5]</td>
<td>LONGVARBINARY, BLOB</td>
<td></td>
</tr>
<tr>
<td>double[]</td>
<td>1 [5]</td>
<td>LONGVARBINARY, BLOB</td>
<td></td>
</tr>
<tr>
<td>float[]</td>
<td>1 [5]</td>
<td>LONGVARBINARY, BLOB</td>
<td></td>
</tr>
<tr>
<td>int[]</td>
<td>1 [5]</td>
<td>LONGVARBINARY, BLOB</td>
<td></td>
</tr>
<tr>
<td>long[]</td>
<td>1 [5]</td>
<td>LONGVARBINARY, BLOB</td>
<td></td>
</tr>
<tr>
<td>short[]</td>
<td>1 [5]</td>
<td>LONGVARBINARY, BLOB</td>
<td></td>
</tr>
<tr>
<td>java.lang.Long</td>
<td>1 [5]</td>
<td>BIGINT, NUMERIC, DOUBLE, DECIMAL, INTEGER</td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>Count</td>
<td>Datatype</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>java.lang.Byte[]</td>
<td>1</td>
<td>LONGVARBINARY, BLOB</td>
<td></td>
</tr>
<tr>
<td>java.lang.Character[]</td>
<td>1</td>
<td>LONGVARBINARY, BLOB</td>
<td></td>
</tr>
<tr>
<td>java.lang.Double[]</td>
<td>1</td>
<td>LONGVARBINARY, BLOB</td>
<td></td>
</tr>
<tr>
<td>java.lang.Float[]</td>
<td>1</td>
<td>LONGVARBINARY, BLOB</td>
<td></td>
</tr>
<tr>
<td>java.lang.Integer[]</td>
<td>1</td>
<td>LONGVARBINARY, BLOB</td>
<td></td>
</tr>
<tr>
<td>java.lang.Long[]</td>
<td>1</td>
<td>LONGVARBINARY, BLOB</td>
<td></td>
</tr>
<tr>
<td>java.lang.Short[]</td>
<td>1</td>
<td>LONGVARBINARY, BLOB</td>
<td></td>
</tr>
<tr>
<td>java.lang.Number</td>
<td>1</td>
<td>LONGVARBINARY, BLOB</td>
<td></td>
</tr>
<tr>
<td>java.lang.Object</td>
<td>1</td>
<td>LONGVARBINARY, BLOB</td>
<td></td>
</tr>
<tr>
<td>java.lang.String [8]</td>
<td>1</td>
<td>VARCHAR, CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [6], UNIQUEIDENTIFIER [7], XMLTYPE [9]</td>
<td></td>
</tr>
<tr>
<td>java.lang.StringBuffer [8]</td>
<td>1</td>
<td>VARCHAR, CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [6], UNIQUEIDENTIFIER [7], XMLTYPE [9]</td>
<td></td>
</tr>
<tr>
<td>java.lang.String[]</td>
<td>1</td>
<td>LONGVARBINARY, BLOB</td>
<td></td>
</tr>
<tr>
<td>java.lang.Enum</td>
<td>1</td>
<td>LONGVARBINARY, BLOB, VARCHAR, INTEGER</td>
<td></td>
</tr>
<tr>
<td>java.lang.Enum[]</td>
<td>1</td>
<td>LONGVARBINARY, BLOB</td>
<td></td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>1</td>
<td>DECIMAL, NUMERIC</td>
<td></td>
</tr>
<tr>
<td>java.math.BigInteger</td>
<td>1</td>
<td>NUMERIC, DECIMAL</td>
<td></td>
</tr>
<tr>
<td>java.math.BigDecimal[]</td>
<td>1</td>
<td>LONGVARBINARY, BLOB</td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>Count</td>
<td>Variables</td>
<td></td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>-------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>java.math.BigInteger[]</td>
<td>1</td>
<td>[5]</td>
<td></td>
</tr>
<tr>
<td>java.sql.Date</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.sql.Time</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.sql.Timestamp</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.ArrayList</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.BitSet</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Calendar</td>
<td>1</td>
<td>[2]</td>
<td></td>
</tr>
<tr>
<td>java.util.Collection</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Currency</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Date</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Date[]</td>
<td>1</td>
<td>[5]</td>
<td></td>
</tr>
<tr>
<td>java.util.GregorianCalendar</td>
<td>1 or 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.HashMap</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.HashSet</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Hashtable</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.LinkedHashMap</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.LinkedHashSet</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.LinkedList</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.List</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Locale</td>
<td>[8]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Locale[]</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Locale[]</td>
<td>1</td>
<td>[5]</td>
<td></td>
</tr>
<tr>
<td>java.util.Map</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| Variables: | LONGVARBINARY, BLOB, DATE, TIMESTAMP, TIME, TIMESTAMP, VARCHAR, CHAR, TIMESTAMP, BIGINT, CHAR, VARCHAR, LONGVARCHAR, CLOB, BLOB, DATALINK, UNIQUEIDENTIFIER, XMLTYPE |</p>
<table>
<thead>
<tr>
<th>Class</th>
<th>Count</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.util.Properties</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>java.util.PriorityQueue</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>java.util.Queue</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>java.util.Set</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>java.util.SortedMap</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>java.util.SortedSet</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>java.util.Stack</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>java.util.TimeZone [8]</td>
<td>1</td>
<td>VARCHAR, CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [7], UNIQUEIDENTIFIER [8], XMLTYPE [10]</td>
</tr>
<tr>
<td>java.util.TreeMap</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>java.util.TreeSet</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>java.util.UUID [8]</td>
<td>1</td>
<td>VARCHAR, CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [7], UNIQUEIDENTIFIER [8], XMLTYPE [10]</td>
</tr>
<tr>
<td>java.util.Vector</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>java.awt.Color [1]</td>
<td>4</td>
<td>INTEGER</td>
</tr>
<tr>
<td>java.awt.Point [2]</td>
<td>2</td>
<td>INTEGER</td>
</tr>
<tr>
<td>java.awt.image.BufferedImage</td>
<td>1</td>
<td>LONGVARBINARY, BLOB</td>
</tr>
<tr>
<td>java.net.URI [8]</td>
<td>1</td>
<td>VARCHAR, CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [7], UNIQUEIDENTIFIER [8], XMLTYPE [10]</td>
</tr>
<tr>
<td>java.net.URL [8]</td>
<td>1</td>
<td>VARCHAR, CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [7], UNIQUEIDENTIFIER [8], XMLTYPE [10]</td>
</tr>
<tr>
<td>java.io.Serializable</td>
<td>1</td>
<td>LONGVARBINARY, BLOB</td>
</tr>
<tr>
<td>javax.jdo.spi.PersistenceCapable [embedded]</td>
<td>1</td>
<td>[embedded]</td>
</tr>
</tbody>
</table>
18 Datastore Types

- [4] - `java.awt.image.BufferedImage` is stored using JPG image format
- [5] - Array types are queryable if not serialised, but stored to many rows
- [6] - DATALINK JDBC type supported on DB2 only. Uses the SQL function DLURLCOMPLETEONLY to fetch from the datastore. You can override this using the select-function extension. See the JDO MetaData reference.
- [7] - UNIQUEIDENTIFIER JDBC type supported on MSSQL only.
- [8] - Oracle treats an empty string as the same as NULL. To workaround this limitation DataNucleus replaces the empty string with the character \u0001.
- [9] - XMLTYPE JDBC type supported on Oracle only, and is included in the "datanucleus-rdbms" plugin.

If you need to extend the provided DataNucleus capabilities in terms of its datastore types support you can utilise a plugin point.

18.1.2 Supported JDBC types

DataNucleus provides support for the majority of the JDBC types. The support is shown below.

<table>
<thead>
<tr>
<th>JDBC Type</th>
<th>Supported</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>BIGINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BINARY</td>
<td></td>
<td>Only for spatial types on MySQL</td>
</tr>
<tr>
<td>BIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLOB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOOLEAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLOB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATALINK</td>
<td></td>
<td>Only on DB2</td>
</tr>
<tr>
<td>DATE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Datastore Types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DISTINCT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JAVA_OBJECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LONGVARBINARY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LONGVARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OTHER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMALLINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STRUCT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TINYINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VARBINARY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VARCHAR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Only for spatial types on PostgreSQL with PostGIS extension
- Only for spatial types on Oracle
19 Failover

19.1 RDBMS : Failover

In the majority of production situations it is desirable to have a level of failover between the underlying datastores used for persistence. You have at least 2 options available to you here. These are shown below

19.1.1 Sequoia

Sequoia is a transparent middleware solution offering clustering, load balancing and failover services for any database. Sequoia is the continuation of the C-JDBC project. The database is distributed and replicated among several nodes and Sequoia balances the queries among these nodes. Sequoia handles node and network failures with transparent failover. It also provides support for hot recovery, online maintenance operations and online upgrades.

Sequoia can be used with DataNucleus by just providing the Sequoia datastore URLs as input to DataNucleus. There is a problem outstanding in Sequoia itself in that its JDBC driver doesn’t provide DataNucleus with the correct major/minor versions of the underlying datastore. Until Sequoia fix this issue, use of Sequoia will be unreliable

19.1.2 DataNucleus Failover capability

DataNucleus has the capability to switch to between DataSources upon failure of one while obtaining a datastore connection. The failover mechanism is useful for applications with multiple database nodes when the data is actually replicated/synchronized by the underlying database. There are 2 things to be aware of before utilising this functionality.

• DataNucleus doesn’t replicate changes to all database nodes, and for this reason, this feature is suggested to be used only for reading objects or if the database is capable to replicate the changes to all nodes.
• If a connection breaks while in use the failover mechanism will not handle it, thus the user application must take care of restarting the transaction and execute the operations.

Several failover algorithm are allowed to be used, one at a time, as for example round-robin, ordered list or random. The default algorithm, ordered list, is described below and is provided by DataNucleus. You can also implement and plug your own algorithm. See Connection Provider.

To use failover, each datastore connection must be provided through DataSources. The datanucleus.ConnectionFactoryName property must be declared with a list of JNDI names pointing to DataSources, in the form of <JNDINAME>,<JNDINAME>. See the example:

```
datanucleus.ConnectionFactoryName=JNDINAME1,JNDINAME2
```

At least one least one JNDI name must be declared.

The Ordered List Algorithm (default) allows you to switch to slave DataSources upon failure of a master DataSource while obtaining a datastore connection. This is shown below.
Each time DataNucleus needs to obtain a connection to the datastore, it takes the first DataSource, the Master, and tries, on failure to obtain the connection goes to the next on the list until it obtains a connection to the datastore or the end of the list is reached.

The first JNDI name in the `datanucleus.ConnectionFactoryName` property is the Master DataSource and the following JNDI names are the Slave DataSources.
20 Queries

20.1 RDBMS : Queries

Using an RDBMS datastore DataNucleus allows you to query the objects in the datastore using the following:
- **JDOQL** - language based around the objects that are persisted and using Java-type syntax
- **SQL** - language found on almost all RDBMSs.
- **JPQL** - language defined in the JPA1 specification for JPA persistence which closely mirrors SQL.

When using queries with RDBMS there are some specific situations where it can be useful to benefit from special treatment. These are listed here.

20.1.1 Result Set : Type

The `java.sql.ResultSet` defines three possible result set types:
- **forward-only**: the result set is navigable forwards only.
- **scroll-sensitive**: the result set is scrollable in both directions and is sensitive to changes in the datastore.
- **scroll-insensitive**: the result set is scrollable in both directions and is insensitive to changes in the datastore.

DataNucleus allows specification of this type as a query extension `datanucleus.rdbms.query.resultSetType`.

To do this on a per query basis for JDO you would do:

```java
query.addExtension("datanucleus.rdbms.query.resultSetType", "scroll-insensitive");
```

To do this on a per query basis for JPA you would do:

```java
query.setHint("datanucleus.rdbms.query.resultSetType", "scroll-insensitive");
```

The default is `forward-only`. The benefit of the other two is that the result set will be scrollable and hence objects will only be read into memory when accessed. So if you have a large result set you should set this to one of the scrollable values.

20.1.2 Result Set : Caching of Results

When using a "scrollable" result set (see above for `datanucleus.rdbms.query.resultSetType`) by default the query result will cache the rows that have been read. You can control this caching to optimise it for your memory requirements. You can set the query extension `datanucleus.query.resultCacheType` and it has the following possible values:
- **weak**: use a weak hashmap for caching (default)
• *soft*: use a soft reference map for caching
• *hard*: use a HashMap for caching (objects not garbage collected)
• *none*: no caching (hence uses least memory)

To set this on a per query basis for JDO you would do

```java
query.addExtension("datanucleus.query.resultCacheType", "weak");
```

To do this on a per query basis for JPA you would do

```java
query.setHint("datanucleus.query.resultCacheType", "weak");
```

### 20.1.3 Large Result Sets: Size

If you have a large result set you clearly don't want to instantiate all objects since this would hit the memory footprint of your application. To get the number of results many JDBC drivers will load all rows of the result set. This is to be avoided so DataNucleus provides control over the mechanism for getting the size of results. The persistence property `datanucleus.query.resultSizeMethod` has a default of *last* (which means navigate to the last object - hence hitting the JDBC driver problem). If you set this to *count* then it will use a simple "count()" query to get the size.

To do this on a per query basis for JDO you would do

```java
query.addExtension("datanucleus.query.resultSizeMethod", "count");
```

To do this on a per query basis for JPA you would do

```java
query.setHint("datanucleus.query.resultSizeMethod", "count");
```

### 20.1.4 Large Result Sets: Loading Results at Commit()

When a transaction is committed by default all remaining results for a query are loaded so that the query is usable thereafter. With a large result set you clearly don't want this to happen. So in this case you should set the extension `datanucleus.query.loadResultsAtCommit` to false.

To do this on a per query basis for JDO you would do

```java
query.addExtension("datanucleus.query.loadResultsAtCommit", "false");
```

To do this on a per query basis for JPA you would do

```java
query.setHint("datanucleus.query.loadResultsAtCommit", "false");
```
20.1.5 Result Set : Control

DataNucleus provides a useful extension allowing control over the ResultSet's that are created by queries. You have at your convenience some properties that give you the power to control whether the result set is read only, whether it can be read forward only, the direction of fetching etc.

To do this on a per query basis for JDO you would do

```java
query.addExtension("datanucleus.rdbms.query.fetchDirection", "forward");
query.addExtension("datanucleus.rdbms.query.resultSetConcurrency", "read-only");
```

To do this on a per query basis for JPA you would do

```java
query.setHint("datanucleus.rdbms.query.fetchDirection", "forward");
query.setHint("datanucleus.rdbms.query.resultSetConcurrency", "read-only");
```

Alternatively you can specify these as persistence properties so that they apply to all queries for that PMF/EMF. Again, the properties are

- **datanucleus.rdbms.query.fetchDirection** - controls the direction that the ResultSet is navigated. By default this is forwards only. Use this property to change that.
- **datanucleus.rdbms.query.resultSetConcurrency** - controls whether the ResultSet is read only or updateable.

Bear in mind that not all RDBMS support all of the possible values for these options. That said, they do add a degree of control that is often useful.

20.1.6 JDOQL : SQL Generation

When using the method contains on a collection (or containsKey, containsValue on a map) this will either add an EXISTS subquery (if there is a NOT or OR present in the query) or will add an INNER JOIN across to the element table. Let's take an example

```sql
SELECT FROM org.datanucleus.samples.A
WHERE (elements.contains(b1) && b1.name == 'Jones')
VARIABLES org.datanucleus.samples.B b1
```

Note that we add the contains first that binds the variable "b1" to the element table, and then add the condition on the variable. The order is important here. If we instead had put the condition on the variable first we would have had to do a CROSS JOIN to the variable table and then try to repair the situation and change it to INNER JOIN if possible. In this case the generated SQL will be like

```sql
SELECT `A0`.`ID`
FROM `A` `A0`
INNER JOIN `B` `B0` ON `A0`.ID = `B`.ELEMENT
WHERE `B0`.NAME = 'Jones'
```
20.1.7 JDOQL : Use of variables and joining

In all situations we aim for DataNucleus JDOQL implementation to work out the right way of linking a variable into the query, whether this is via a join (INNER, LEFT OUTER), or via a subquery. As you can imagine this can be complicated to work out the optimum for all situations so that in mine we allow (for a limited number of situations) the option of specifying the join type. This is achieved by setting the query extension `datanucleus.query.jdoql.{varName}.join` to the required type. For 1-1 relations this would be either "INNERJOIN" or "LEFTOUTERJOIN", and for 1-N relations this would be either "INNERJOIN" or "SUBQUERY".

Please, if you find a situation where the optimum join type is not chosen then report it in JIRA for project "NUCRDBMS" as priority "Minor" so it can be registered for future work.

20.1.8 JPQL : SQL Generation

With a JPQL query running on an RDBMS the query is compiled into SQL. Here we give a few examples of what SQL is generated. You can of course try this for yourself observing the content of the DataNucleus log.

In JPQL you specify a candidate class and its alias (identifier). In addition you can specify joins with their respective alias. The DataNucleus implementation of JPQL will preserve these aliases in the generated SQL.

```
JPQL:
SELECT Object(P) FROM mydomain.Person P INNER JOIN P.bestFriend AS B

SQL:
SELECT P.ID
FROM PERSON P INNER JOIN PERSON B ON B.ID = P.BESTFRIEND_ID
```

With the JPQL `MEMBER OF` syntax this is typically converted into an EXISTS query.

```
JPQL:
SELECT DISTINCT Object(p) FROM mydomain.Person p WHERE :param MEMBER OF p.friends

SQL:
SELECT DISTINCT P.ID FROM PERSON P
WHERE EXISTS {
    SELECT 1 FROM PERSON_FRIENDS P_FRIENDS, PERSON P_FRIENDS_1
    WHERE P_FRIENDS.PERSON_ID = P.ID
    AND P_FRIENDS_1.GLOBAL_ID = P_FRIENDS.FRIEND_ID
    AND 101 = P_FRIENDS_1.ID
}
```
21 JDOQL : Spatial Methods

21.1 RDBMS : JDOQL Spatial Methods

When querying spatial data you can make use of a set of spatial methods on the various Java geometry types. The list contains all of the functions detailed in Section 3.2 of the OGC Simple Features specification. Additionally DataNucleus provides some commonly required methods like bounding box test and datastore specific functions. The following tables list all available functions as well as information about which RDBMS implement them. An entry in the "Result" column indicates, whether the function may be used in the result part of a JDOQL query.

**Functions for Constructing a Geometry Value given its Well-known Text Representation (OGC SF 3.2.6)**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial.geomFromText(String, Integer)</td>
<td>Construct a Geometry value given its well-known textual representation.</td>
<td>OGC SF</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
</tr>
<tr>
<td>Spatial.pointFromText(String, Integer)</td>
<td>Construct a Point.</td>
<td>OGC SF</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
</tr>
<tr>
<td>Spatial.lineFromText(String, Integer)</td>
<td>Construct a LineString.</td>
<td>OGC SF</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
</tr>
<tr>
<td>Spatial.polyFromText(String, Integer)</td>
<td>Construct a Polygon.</td>
<td>OGC SF</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
</tr>
<tr>
<td>Spatial.mPointFromText(String, Integer)</td>
<td>Construct a MultiPoint.</td>
<td>OGC SF</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
</tr>
<tr>
<td>Spatial.mLineFromText(String, Integer)</td>
<td>Construct a MultiLineString.</td>
<td>OGC SF</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
</tr>
<tr>
<td>Spatial.mPolyFromText(String, Integer)</td>
<td>Construct a MultiPolygon.</td>
<td>OGC SF</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
</tr>
<tr>
<td>Spatial.geomCollFromText(String, Integer)</td>
<td>Construct a GeometryCollection.</td>
<td>OGC SF</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
</tr>
</tbody>
</table>

[1] These functions can't be used in the return part because it's not possible to determine the return type from the parameters.

**Functions for Constructing a Geometry Value given its Well-known Binary Representation (OGC SF 3.2.7)**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial.geomCollFromText(String, Integer)</td>
<td>Construct a GeometryCollection.</td>
<td>OGC SF</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
<td>![x]</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
<td>Specification</td>
<td>Result</td>
<td>PostGIS</td>
<td>MySQL</td>
<td>Oracle Spatial</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>---------------</td>
<td>----------------------</td>
<td>---------</td>
<td>-------</td>
<td>----------------</td>
</tr>
<tr>
<td>Spatial.geomFm</td>
<td>Construct a Geometry value given its well-known binary representation.</td>
<td>OGC SF</td>
<td>![Red]</td>
<td>![Green]</td>
<td>![Green]</td>
<td>![Green]</td>
</tr>
<tr>
<td>Spatial.pointFm</td>
<td>Construct a Point.</td>
<td>OGC SF</td>
<td>![Red]</td>
<td>![Green]</td>
<td>![Green]</td>
<td>![Green]</td>
</tr>
<tr>
<td>Spatial.lineFm</td>
<td>Construct a LineString.</td>
<td>OGC SF</td>
<td>![Red]</td>
<td>![Green]</td>
<td>![Green]</td>
<td>![Green]</td>
</tr>
<tr>
<td>Spatial.polyFm</td>
<td>Construct a Polygon.</td>
<td>OGC SF</td>
<td>![Red]</td>
<td>![Green]</td>
<td>![Green]</td>
<td>![Green]</td>
</tr>
<tr>
<td>Spatial.mPointFm</td>
<td>Construct a MultiPoint.</td>
<td>OGC SF</td>
<td>![Red]</td>
<td>![Green]</td>
<td>![Green]</td>
<td>![Green]</td>
</tr>
<tr>
<td>Spatial.mLineFm</td>
<td>Construct a MultiLineString.</td>
<td>OGC SF</td>
<td>![Red]</td>
<td>![Green]</td>
<td>![Green]</td>
<td>![Green]</td>
</tr>
<tr>
<td>Spatial.mPolyFm</td>
<td>Construct a MultiPolygon.</td>
<td>OGC SF</td>
<td>![Red]</td>
<td>![Green]</td>
<td>![Green]</td>
<td>![Green]</td>
</tr>
<tr>
<td>Spatial.geomCc</td>
<td>Construct a GeometryCollection.</td>
<td>OGC SF</td>
<td>![Red]</td>
<td>![Green]</td>
<td>![Green]</td>
<td>![Green]</td>
</tr>
</tbody>
</table>

[1] These functions can't be used in the return part because it's not possible to determine the return type from the parameters.

**Functions on Type Geometry**

(OGC SF 3.2.10)
### Spatial.srid(Geometry)
Returns the Spatial Reference System ID for this Geometry.

**OGC SF**

### Spatial.isEmpty(Geometry)
TRUE if this Geometry corresponds to the empty set.

**OGC SF**

### Spatial.isSimple(Geometry)
TRUE if this Geometry is simple, as defined in the Geometry Model.

**OGC SF**

### Spatial.boundary(Geometry)
Returns a Geometry that is the combinatorial boundary of the Geometry.

**OGC SF**

### Spatial.envelope(Geometry)
Returns the rectangle bounding Geometry as a Polygon.

**OGC SF**

---

[1] Oracle does not allow boolean expressions in the SELECT-list.

**Functions on Type Point**
(OGC SF 3.2.11)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Specification Result</th>
<th>PostGIS</th>
<th>MySQL</th>
<th>Oracle</th>
<th>Spatial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial.x(Point)</td>
<td>Returns the x-coordinate of the Point as a Double.</td>
<td>OGC SF</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Spatial.y(Point)</td>
<td>Returns the y-coordinate of the Point as a Double.</td>
<td>OGC SF</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

**Functions on Type Curve**
(OGC SF 3.2.12)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Specification Result</th>
<th>PostGIS</th>
<th>MySQL</th>
<th>Oracle</th>
<th>Spatial</th>
</tr>
</thead>
</table>

© 2015, DataNucleus • ALL RIGHTS RESERVED.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Specification Result</th>
<th>PostGIS</th>
<th>MySQL</th>
<th>Oracle</th>
<th>Spatial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial.startPoi</td>
<td>Returns the first point of the Curve.</td>
<td>OGC SF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spatial.endPoi</td>
<td>Returns the last point of the Curve.</td>
<td>OGC SF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spatial.isRing(C)</td>
<td>Returns TRUE if Curve is closed and simple.</td>
<td>[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[1] Oracle does not allow boolean expressions in the SELECT-list.

Functions on Type Curve and Type MultiCurve
(OGC SF 3.2.12, 3.2.17)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Specification Result</th>
<th>PostGIS</th>
<th>MySQL</th>
<th>Oracle</th>
<th>Spatial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial.isClosed(C)</td>
<td>Returns TRUE if Curve is closed, i.e., if StartPoint(Curve) = EndPoint(Curve).</td>
<td>[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spatial.length(C)</td>
<td>Returns the length of the Curve.</td>
<td>OGC SF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[1] Oracle does not allow boolean expressions in the SELECT-list.

Functions on Type LineString
(OGC SF 3.2.13)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Specification Result</th>
<th>PostGIS</th>
<th>MySQL</th>
<th>Oracle</th>
<th>Spatial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial.numPoints</td>
<td>Returns the number of points in the LineString.</td>
<td>OGC SF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spatial.pointN(l)</td>
<td>Returns Point n.</td>
<td>OGC SF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Functions on Type Surface and Type MultiSurface
(OGC SF 3.2.14, 3.2.18)
Spatial.centroid Returns the centroid(MultiSurface) centroid of Surface, which may lie outside of it. OGC SF

Spatial.pointOnSurface Returns pointOnSurface a Point guaranteed to lie on the surface. OGC SF

Spatial.area Surface Returns area(MultiSurface) the area of Surface. OGC SF

[1] MySQL does not implement these functions.

**Functions on Type Polygon**
(OGC SF 3.2.15)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Specification</th>
<th>Result</th>
<th>PostGIS</th>
<th>MySQL</th>
<th>Oracle</th>
<th>Spatial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial.exteriorRing</td>
<td>Returns the exterior ring of Polygon.</td>
<td>OGC SF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spatial.numInteriorRing</td>
<td>Returns the number of interior rings.</td>
<td>OGC SF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spatial.interiorRingN</td>
<td>Returns the nth interior ring.</td>
<td>OGC SF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Functions on TypeGeomCollection**
(OGC SF 3.2.16)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Specification</th>
<th>Result</th>
<th>PostGIS</th>
<th>MySQL</th>
<th>Oracle</th>
<th>Spatial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial.numGeometries</td>
<td>Returns the number of geometries in the collection.</td>
<td>OGC SF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spatial.geometryN</td>
<td>Returns the nth geometry in the collection.</td>
<td>OGC SF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Functions that test Spatial Relationships**
(OGC SF 3.2.19)
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>OGC SF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial.equals(Geometry)</td>
<td>TRUE if the two geometries are spatially equal.</td>
<td>[2]</td>
</tr>
<tr>
<td>Spatial.disjoint(Geometry)</td>
<td>TRUE if the two geometries are spatially disjoint.</td>
<td>[2]</td>
</tr>
<tr>
<td>Spatial.touches(Geometry)</td>
<td>TRUE if the first Geometry spatially touches the other Geometry.</td>
<td>[2]</td>
</tr>
<tr>
<td>Spatial.within(Geometry)</td>
<td>TRUE if first Geometry is completely contained in second Geometry.</td>
<td>[2]</td>
</tr>
<tr>
<td>Spatial.overlaps(Geometry)</td>
<td>TRUE if first Geometries is spatially overlapping the other Geometry.</td>
<td>[2]</td>
</tr>
<tr>
<td>Spatial.crosses(Geometry)</td>
<td>TRUE if first Geometry crosses the other Geometry.</td>
<td>[3]</td>
</tr>
<tr>
<td>Spatial.intersect(Geometry)</td>
<td>TRUE if first Geometry spatially intersects the other Geometry.</td>
<td>[2]</td>
</tr>
<tr>
<td>Spatial.contains(Geometry)</td>
<td>TRUE if second Geometry is completely contained in first Geometry.</td>
<td>[2]</td>
</tr>
<tr>
<td>Spatial.relate(Geometry, String)</td>
<td>TRUE if the spatial relationship specified by the patternMatrix holds.</td>
<td>[3]</td>
</tr>
</tbody>
</table>
[1] Oracle does not allow boolean expressions in the SELECT-list.
[2] MySQL does not implement these functions according to the specification. They return the same result as the corresponding MBR-based functions.
[3] MySQL does not implement these functions.

Function on Distance Relationships
(OGC SF 3.2.20)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Specification Result</th>
<th>PostGIS</th>
<th>MySQL</th>
<th>Oracle Spatial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial.distance(Geometry)</td>
<td>Returns the distance between the two geometries.</td>
<td>OGC SF</td>
<td></td>
<td>[-]</td>
<td></td>
</tr>
</tbody>
</table>

[1] MySQL does not implement this function.

Functions that implement Spatial Operators
(OGC SF 3.2.21)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Specification Result</th>
<th>PostGIS</th>
<th>MySQL</th>
<th>Oracle Spatial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial.intersection(Geometry)</td>
<td>Returns a Geometry that is the set intersection of the two geometries.</td>
<td>OGC SF</td>
<td></td>
<td>[-]</td>
<td></td>
</tr>
<tr>
<td>Spatial.difference(Geometry)</td>
<td>Returns a Geometry that is the closure of the set difference of the two geometries.</td>
<td>OGC SF</td>
<td></td>
<td>[-]</td>
<td></td>
</tr>
<tr>
<td>Spatial.union(Geometry)</td>
<td>Returns a Geometry that is the set union of the two geometries.</td>
<td>OGC SF</td>
<td></td>
<td>[-]</td>
<td></td>
</tr>
<tr>
<td>Spatial.symDiff(Geometry)</td>
<td>Returns a Geometry that is the closure of the set symmetric difference of the two geometries.</td>
<td>OGC SF</td>
<td></td>
<td>[-]</td>
<td></td>
</tr>
</tbody>
</table>
### Spatial Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>OGC SF</th>
<th>PostGIS</th>
<th>MySQL</th>
<th>Oracle Spatial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial.buffer(G, Double)</td>
<td>Returns a Geometry defined by buffering a distance around the Geometry.</td>
<td>![ ]</td>
<td>![ ]</td>
<td>![ ]</td>
<td>![ ]</td>
</tr>
<tr>
<td>Spatial.convexHull(Geometry)</td>
<td>Returns a Geometry that is the convex hull of the Geometry.</td>
<td>![ ]</td>
<td>![ ]</td>
<td>![ ]</td>
<td>![ ]</td>
</tr>
</tbody>
</table>

[1] These functions are currently not implemented in MySQL. They may appear in future releases.

Test whether the bounding box of one geometry intersects the bounding box of another

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Result</th>
<th>PostGIS</th>
<th>MySQL</th>
<th>Oracle Spatial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial.bboxTest(Geometry, G)</td>
<td>Returns TRUE if the bounding box of the first Geometry overlaps second Geometry's bounding box</td>
<td>![ ]</td>
<td>![ ]</td>
<td>![ ]</td>
<td>![ ]</td>
</tr>
</tbody>
</table>

[1] Oracle does not allow boolean expressions in the SELECT-list.

### PostGIS Spatial Operators

These functions are only supported on PostGIS.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostGIS.bboxOverlapsLeft(G, G)</td>
<td>The PostGIS (&lt;) operator returns TRUE if the bounding box of the first Geometry overlaps or is to the left of second Geometry's bounding box</td>
<td>![ ]</td>
</tr>
<tr>
<td>PostGIS.bboxOverlapsRight(G, G)</td>
<td>The PostGIS (&gt;) operator returns TRUE if the bounding box of the first Geometry overlaps or is to the right of second Geometry's bounding box</td>
<td>![ ]</td>
</tr>
<tr>
<td>PostGIS.bboxLeft(G, G)</td>
<td>The PostGIS (&lt;=) operator returns TRUE if the bounding box of the first Geometry overlaps or is strictly to the left of second Geometry's bounding box</td>
<td>![ ]</td>
</tr>
</tbody>
</table>
PostGIS.bboxRight(Geometry, Geometry)  The PostGIS >> operator returns TRUE if the bounding box of the first Geometry overlaps or is strictly to the right of second Geometry's bounding box

PostGIS.bboxOverlapsBelow(Geometry, Geometry)  The PostGIS &<@ operator returns TRUE if the bounding box of the first Geometry overlaps or is below second Geometry's bounding box

PostGIS.bboxOverlapsAbove(Geometry, Geometry)  The PostGIS |>> operator returns TRUE if the bounding box of the first Geometry overlaps or is above second Geometry's bounding box

PostGIS.bboxBelow(Geometry, Geometry)  The PostGIS <<| operator returns TRUE if the bounding box of the first Geometry is strictly below second Geometry's bounding box

PostGIS.bboxAbove(Geometry, Geometry)  The PostGIS |>> operator returns TRUE if the bounding box of the first Geometry is strictly above second Geometry's bounding box

PostGIS.sameAs(Geometry, Geometry)  The PostGIS ~= operator returns TRUE if the two geometries are vertex-by-vertex equal.

PostGIS.bboxWithin(Geometry, Geometry)  The PostGIS @ operator returns TRUE if the bounding box of the first Geometry overlaps or is completely contained by second Geometry's bounding box

PostGIS.bboxContains(Geometry, Geometry)  The PostGIS - operator returns TRUE if the bounding box of the first Geometry completely contains second Geometry's bounding box

**MySQL specific Functions for Testing Spatial Relationships between Minimal Bounding Boxes**

These functions are only supported on MySQL.

<table>
<thead>
<tr>
<th>Method</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL.mbrEqual(Geometry, Geometry)</td>
<td></td>
</tr>
<tr>
<td>MySQL.mbrDisjoint(Geometry, Geometry)</td>
<td></td>
</tr>
<tr>
<td>MySQL.mbrIntersects(Geometry, Geometry)</td>
<td></td>
</tr>
<tr>
<td>MySQL.mbrTouches(Geometry, Geometry)</td>
<td></td>
</tr>
<tr>
<td>MySQL.mbrWithin(Geometry, Geometry)</td>
<td></td>
</tr>
<tr>
<td>MySQL.mbrContains(Geometry, Geometry)</td>
<td></td>
</tr>
</tbody>
</table>
### Oracle specific Functions for Constructing SDO_GEOMETRY types

These functions are only supported on Oracle Spatial.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle.sdo_geometry(Integer gtype, Integer srid, SDO_POINT point, SDO_ELEM_INFO_ARRAY elem_info, SDO_ORDINATE_ARRAY ordinates)</td>
<td>Creates a SDO_GEOMETRY geometry from the passed geometry type, srid, point, element infos and ordinates.</td>
</tr>
<tr>
<td>Oracle.sdo_point_type(Double x, Double y, Double z)</td>
<td>Creates a SDO_POINT geometry from the passed ordinates.</td>
</tr>
<tr>
<td>Oracle.sdo_elem_info_array(String numbers)</td>
<td>Creates a SDO_ELEM_INFO_ARRAY from the passed comma-separated integers.</td>
</tr>
<tr>
<td>Oracle.sdoordinate_array(String ordinates)</td>
<td>Creates a SDO_ORDINATE_ARRAY from the passed comma-separated doubles.</td>
</tr>
</tbody>
</table>

### Examples

The following sections provide some examples of what can be done using spatial methods in JDOQL queries. In the examples we use a class from the test suite. Here’s the source code for reference:

```java
package org.datanucleus.samples.pggeometry;

import org.postgis.LineString;

public class SampleLineString {
    private long id;
    private String name;
    private LineString geom;

    public SampleLineString(long id, String name, LineString lineString) {
        this.id = id;
        this.name = name;
        this.geom = lineString;
    }

    public long getId() {
        return id;
    }
    ...
}
```
### 21.1.1 Example 1 - Spatial Function in the Filter of a Query

This example shows how to use spatial functions in the filter of a query. The query returns a list of `SampleLineString` objects whose line string has a length less than the given limit.

```java
Double limit = new Double(100.0);
Query query = pm.newQuery(SampleLineString.class, "geom != null && Spatial.length(geom) < :limit");
List list = (List) query.execute(limit);
```

### 21.1.2 Example 2 - Spatial Function in the Result Part of a Query

This time we use a spatial function in the result part of a query. The query returns the length of the line string from the selected `SampleLineString`.

```java
query = pm.newQuery(SampleLineString.class, "id == :id");
query.setResult("Spatial.pointN(geom, 2)";
query.setUnique(true);
Geometry point = (Geometry) query.execute(new Long(1001));
```

### 21.1.3 Example 3 - Nested Functions

You may want to use nested functions in your query. This example shows how to do that. The query returns a list of `SampleLineString` objects, whose end point spatially equals a given point.

```java
Point point = new Point("SRID=4326;POINT(110 45)";
Query query = pm.newQuery(SampleLineString.class, "geom != null && Spatial.equals(Spatial.endPoint(geom), :point)");
List list = (List) query.execute(point);
```
22 Statement Batching

22.1 RDBMS : Statement Batching

When changes are required to be made to an underlying RDBMS datastore, statements are sent via JDBC. A statement is, in general, a single SQL command, and is then executed. In some circumstances the statements due to be sent to the datastore are the same JDBC statement several times. In this case the statement can be batched. This means that a statement is created for the SQL, and it is passed to the datastore with multiple sets of values before being executed. When it is executed the SQL is executed for each of the sets of values. DataNucleus allows statement batching under certain circumstances.

The maximum number of statements that can be included in a batch can be set via a persistence property datanucleus.rdbms.statementBatchLimit. This defaults to 50. If you set it to -1 then there is no maximum limit imposed. Setting it to 0 means that batching is turned off.

It should be noted that while batching sounds essential, it is only of any possible use when the exact same SQL is required to be executed more than 1 times in a row. If a different SQL needs executing between 2 such statements then no batching is possible anyway. Let’s take an example

```
INSERT INTO MYTABLE VALUES(?,?,?,?)
INSERT INTO MYTABLE VALUES(?,?,?,?)
SELECT ID, NAME FROM MYOTHERTABLE WHERE VALUE=?
INSERT INTO MYTABLE VALUES(?,?,?,?)
SELECT ID, NAME FROM MYOTHERTABLE WHERE VALUE=?
```

In this example the first two statements can be batched together since they are identical and nothing else separates them. All subsequent statements cannot be batched since no two identical statements follow each other.

The statements that DataNucleus currently allows for batching are

- Insert of objects. This is not enabled when objects being inserted are using identity value generation strategy
- Delete of objects
- Insert of container elements/keys/values
- Delete of container elements/keys/values

Please note that if using MySQL, you should also specify the connection URL with the argument rewriteBatchedStatements=true since MySQL won't actually batch without this
23 Views

23.1 RDBMS: Views

DataNucleus supports persisting objects to RDBMS datastores, persisting to Tables. The majority of RDBMS also provide support for Views, providing the equivalent of a read-only SELECT across various tables. DataNucleus also provides support for querying such Views. This provides more flexibility to the user where they have data and need to display it in their application. Support for Views is described below.

When you want to access data according to a View, you are required to provide a class that will accept the values from the View when queried, and Meta-Data for the class that defines the View and how it maps onto the provided class. Let's take an example. We have a View SALEABLE_PRODUCT in our database as follows, defined based on data in a PRODUCT table.

```
CREATE VIEW SALEABLE_PRODUCT (ID, NAME, PRICE, CURRENCY) AS
    SELECT ID, NAME, CURRENT_PRICE AS PRICE, CURRENCY FROM PRODUCT
    WHERE PRODUCT.STATUS_ID = 1
```

So we define a class to receive the values from this View.

```
package org.datanucleus.samples.views;
public class SaleableProduct
{
    String id;
    String name;
    double price;
    String currency;

    public String getId()
    {
        return id;
    }

    public String getName()
    {
        return name;
    }

    public double getPrice()
    {
        return price;
    }

    public String getCurrency()
    {
        return currency;
    }
}
```
and then we define how this class is mapped to the View

```xml
<?xml version="1.0"?>
<!DOCTYPE jdo SYSTEM "file:/javax/jdo/jdo.dtd">
<jdo>
  <package name="org.datanucleus.samples.views">
    <class name="SaleableProduct" identity-type="nondurable" table="SALEABLE_PRODUCT">
      <field name="id"/>
      <field name="name"/>
      <field name="price"/>
      <field name="currency"/>

      <!-- This is the "generic" SQL92 version of the view. -->
      <extension vendor-name="datanucleus" key="view-definition" value="
        CREATE VIEW SALEABLE_PRODUCT
        {
          (this.id),
          (this.name),
          (this.price),
          (this.currency)
        } AS
        SELECT ID, NAME, CURRENT_PRICE AS PRICE, CURRENCY FROM PRODUCT
        WHERE PRODUCT.STATUS_ID = 1"/>
    </class>
  </package>
</jdo>
```

Please note the following

- We've defined our class as using "nondurable" identity. This is an important step since rows of the View typically don't operate in the same way as rows of a Table, not mapping onto a persisted updateable object as such.
- We've specified the "table", which in this case is the view name - otherwise DataNucleus would create a name for the view based on the class name.
- We've defined a DataNucleus extension view-definition that defines the view for this class. If the view doesn't already exist it doesn't matter since DataNucleus (when used with autoCreateSchema) will execute this construction definition.
- The view-definition can contain macros utilising the names of the fields in the class, and hence borrowing their column names (if we had defined column names for the fields of the class).
- You can also utilise other classes in the macros, and include them via a DataNucleus MetaData extension view-imports (not shown here)
- If your View already exists you are still required to provide a view-definition even though DataNucleus will not be utilising it, since it also uses this attribute as the flag for whether it is a View or a Table - just make sure that you specify the "table" also in the MetaData.

We can now utilise this class within normal DataNucleus querying operation.
Hopefully that has given enough detail on how to create and access views from a DataNucleus-enabled application.
24 Datastore API

24.1 RDBMS: Datastore Schema API

JDO/JPA are APIs for persisting and retrieving objects to/from datastores. They don't provide a way of accessing the schema of the datastore itself (if it has one). In the case of RDBMS it is useful to be able to find out what columns there are in a table, or what data types are supported for example. DataNucleus Access Platform provides an API for this.

The first thing to do is get your hands on the DataNucleus StoreManager and from that the StoreSchemaHandler. You do this as follows:

```java
import org.datanucleus.api.jdo.JDOPersistenceManagerFactory;
import org.datanucleus.store.StoreManager;
import org.datanucleus.store.schema.StoreSchemaHandler;

[assumed to have "pmf"]
...

StoreManager storeMgr = ((JDOPersistenceManagerFactory)pmf).getStoreManager();
StoreSchemaHandler schemaHandler = storeMgr.getSchemaHandler();
```

So now we have the StoreSchemaHandler what can we do with it? Well start with the javadoc for the implementation that is used for RDBMS

```java
import org.datanucleus.store.rdbms.schema.RDBMSTypesInfo;

Connection conn = (Connection)pm.getDataStoreConnection().getNativeConnection();
RDBMSTypesInfo typesInfo = schemaHandler.getSchemaData(conn, "types");
```

As you can see from the javadocs for RDBMSTypesInfo

```java

```

we can access the JDBC types information via the "children". They are keyed by the JDBC type number of the JDBC type (see java.sql.Types). So we can just iterate it
24.1.2 Column information for a table

Here we have a table in the datastore and want to find the columns present. So we do this

```java
import org.datanucleus.store.rdbms.schema.RDBMSTableInfo;

Connection conn = (Connection)pm.getDataStoreConnection().getNativeConnection();
RDBMSTableInfo tableInfo = schemaHandler.getSchemaData(conn, "columns",
    new Object[] {catalogName, schemaName, tableName});
```

As you can see from the javadocs for `RDBMSTableInfo`

we can access the columns information via the "children".

```java
Iterator columnsIter = tableInfo.getChildren().iterator();
while (columnsIter.hasNext()) {
    RDBMSColumnInfo colInfo = (RDBMSColumnInfo)columnsIter.next();
    ...
}
```

24.1.3 Index information for a table

Here we have a table in the datastore and want to find the indices present. So we do this

```java
import org.datanucleus.store.rdbms.schema.RDBMSTableInfo;

Connection conn = (Connection)pm.getDataStoreConnection().getNativeConnection();
RDBMSTableIndexInfo tableInfo = schemaHandler.getSchemaData(conn, "indices",
    new Object[] {catalogName, schemaName, tableName});
```
As you can see from the javadocs for `RDBMSTableIndexInfo`

```java
Iterator indexIter = tableInfo.getChildren().iterator();
while (indexIter.hasNext())
{
    IndexInfo idxInfo = (IndexInfo)indexIter.next();
    ...
}
```

### 24.1.4 ForeignKey information for a table

Here we have a table in the datastore and want to find the FKs present. So we do this

```java
import org.datanucleus.store.rdbms.schema.RDBMSTableInfo;
Connection conn = (Connection)pm.getDataStoreConnection().getNativeConnection();
RDBMSTableFKInfo tableInfo = schemaHandler.getSchemaData(conn, "foreign-keys",
        new Object[] {catalogName, schemaName, tableName});
```

As you can see from the javadocs for `RDBMSTableFKInfo`

```java
Iterator fkIter = tableInfo.getChildren().iterator();
while (fkIter.hasNext())
{
    ForeignKeyInfo fkInfo = (ForeignKeyInfo)fkIter.next();
    ...
}
```

### 24.1.5 PrimaryKey information for a table

Here we have a table in the datastore and want to find the PK present. So we do this

```java
import org.datanucleus.store.rdbms.schema.RDBMSTableInfo;
Connection conn = (Connection)pm.getDataStoreConnection().getNativeConnection();
RDBMSTablePKInfo tableInfo = schemaHandler.getSchemaData(conn, "primary-keys",
        new Object[] {catalogName, schemaName, tableName});
```
As you can see from the javadocs for `RDBMSTablePKInfo` we can access the foreign-key information via the "children".

```java
Iterator pkIter = tableInfo.getChildren().iterator();
while (pkIter.hasNext())
{
    PrimaryKeyInfo pkinfo = (PrimaryKeyInfo)pkIter.next();
    ...
}
```
25 ODF

25.1 ODF Documents

DataNucleus supports persisting/retrieving objects to/from ODF documents (using the datanucleus-odf plugin, which makes use of the ODFDOM project). Simply specify your "connectionURL" as follows

```java
datanucleus.ConnectionURL=odf:file:myfile.ods
```

replacing "myfile.ods" with your filename, which can be absolute or relative. This connects to a file on your local machine. You then create your PMF/EMF as normal and use JDO/JPA as normal.

The jars required to use DataNucleus ODF persistence are datanucleus-core, datanucleus-api-jdo, datanucleus-api-jpa, datanucleus-odf and odftoolkit

There are tutorials available for use of DataNucleus with ODF for JDO and for JPA

Things to bear in mind with ODF usage :-

- Querying can be performed using JDOQL or JPQL. Any filtering/ordering will be performed in-memory
- Relations : A spreadsheet cannot store related objects directly, since each object is a row of a particular worksheet. DataNucleus gets around this by storing the String-form of the identity of the related object in the relation cell. See

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>35 Fred</td>
<td>[1,2]</td>
<td>[1]</td>
<td>1 Smith</td>
<td>TRUE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 Sarah</td>
<td></td>
<td></td>
<td>2 Green</td>
<td>TRUE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 Chris</td>
<td></td>
<td></td>
<td>3 Tomlinson</td>
<td>TRUE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

25.1.1 Worksheet Headers

A typical spreadsheet has many rows of data. It contains no names of columns tying the data back to the input object (field names). DataNucleus allows an extension specified at class level called include-column-headers (should be set to true). When the table is then created it will include an extra row (the first row) with the column names from the metadata (or field names if no column names were defined). For example
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Age</td>
<td>First Name</td>
<td>Friends</td>
<td>House</td>
<td>IQ</td>
<td>Last Name</td>
<td>Single?</td>
</tr>
<tr>
<td>2</td>
<td>35</td>
<td>Fred</td>
<td>[2,3]</td>
<td>[1]</td>
<td></td>
<td>Smith</td>
<td>TRUE</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>Sarah</td>
<td>[]</td>
<td></td>
<td></td>
<td>2</td>
<td>Green</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>Chris</td>
<td>[]</td>
<td></td>
<td></td>
<td>3</td>
<td>Carlson</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2015, DataNucleus • ALL RIGHTS RESERVED.
26 Excel (XLS)

26.1 Excel Documents

DataNucleus supports persisting/retrieving objects to/from Excel documents (using the datanucleus-excel plugin, which makes use of the Apache POI project). Simply specify your "connectionURL" as follows:

```
datanucleus.ConnectionURL=excel:file:myfile.xls
```

replacing "myfile.xls" with your filename, which can be absolute or relative. This connects to a file on your local machine. You then create your PMF/EMF as normal and use JDO/JPA as normal.

The jars required to use DataNucleus Excel persistence are datanucleus-core, datanucleus-api-jdo, datanucleus-api-jpa, datanucleus-excel and apache-poi

There are tutorials available for use of DataNucleus with Excel for JDO and for JPA

Things to bear in mind with Excel usage:

- Querying can be performed using JDOQL or JPQL. Any filtering/ordering will be performed in-memory.
- Relations: A spreadsheet cannot store related objects directly, since each object is a row of a particular worksheet. DataNucleus gets around this by storing the String-form of the identity of the related object in the relation cell.

26.1.1 References

Some references that may be of some use:

- A JDO With DataNucleus AccessPlatform using Excel and Eclipse Tutorial That Actually Works
27 Excel (OOXML)

27.1 Excel (OOXML) Documents

DataNucleus supports persisting/retrieving objects to/from OOXML documents (using the datanucleus-excel plugin) which makes use of the Apache POI project. Simply specify your "connectionURL" as follows

```
datanucleus.ConnectionURL=excel:file:myfile.xlsx
```

replacing "myfile.xlsx" with your filename, which can be absolute or relative. This connects to a file on your local machine. You then create your PMF/EMF as normal and use JDO/JPA as normal.

The jars required to use DataNucleus OOXML persistence are datanucleus-core, datanucleus-api-jdo, datanucleus-api-jpa, datanucleus-excel and apache-poi

There are tutorials available for use of DataNucleus with Excel for JDO and for JPA

Things to bear in mind with OOXML usage :-

- Querying can be performed using JDOQL or JPQL. Any filtering/ordering will be performed in-memory
- Relations: A spreadsheet cannot store related objects directly, since each object is a row of a particular worksheet. DataNucleus gets around this by storing the String-form of the identity of the related object in the relation cell.
28.1 XML Documents

DataNucleus supports persisting/retrieving objects to/from XML documents (using the **datanucleus-xml** plugin). Simply specify your "connectionURL" as follows:

```
datanucleus.ConnectionURL=xml:file:myfile.xml
```

replacing *myfile.xml* with your filename, which can be absolute or relative.

It makes use of JAXB, and the jars required to use DataNucleus XML persistence are **datanucleus-core**, **datanucleus-xml**, **datanucleus-api-jdo**, **datanucleus-api-jpa**, **datanucleus-xml** and **JAXB API**. If you wish to help out in this effort either by contributing or by sponsoring particular functionality please contact us via the [DataNucleus Forum](https://datanucleus.org/).  

Things to bear in mind with XML usage:-

- Indentation of XML : the persistence property **datanucleus.xml.indentSize** defaults to 4 but you can set it to the desired indent size
- Querying using JDOQL/JPQL will operate in-memory currently.
- Application identity is supported but can only have 1 PK field and must be a String. This is a limitation of JAXB
- Persistent properties are not supported, only persistent fields
- Out of the box it will use the JAXB reference implementation. You could, in principle, provide support for other JAXB implementations by implementing **org.datanucleus.store.xml.JAXBHandler** and then specify the persistence property **datanucleus.xml.jaxbHandlerClass** to the JAXBHandler implementation. If you do manage to write a JAXBHandler for other JAXB implementations please consider contributing it to the project

### 28.1.1 Mapping : XML Datastore Mapping

When persisting a Java object to an XML datastore clearly the user would like some control over the structure of the XML document. Here’s an example using JDO XML MetaData:

```xml
<jdo>
  <package name="org.datanucleus.samples.models.company">
    <class name="Person" detachable="true" schema="/myproduct/people" table="person">
      <field name="personNum">
        <extension vendor-name="datanucleus" key="XmlAttribute" value="true"/>
      </field>
      <field name="firstName" primary-key="true"/> <!-- PK since JAXB requires String -->
      <field name="lastName"/>
      <field name="bestFriend"/>
    </class>
  </package>
</jdo>
```

Things to note :
• **schema** on class is used to define the "XPath" to the root of the class in XML. You can also use the extension "xpath" to specify the same thing.
• **table** on class is used to define the name of the element for an object of the particular class.
• **column** on field is used to define the name of the element for a field of the particular class.
• **XmlAttribute** : when set to true denotes that this will appear in the XML file as an attribute of the overall element for the object
• When a field is primary-key it will gain a JAXB "XmlID" attribute.
• When a field is a relation to another object (and the field is not embedded) then it will gain a JAXB "XmlIDREF" attribute as a link to the other object.
• **Important : JAXB has a limitation for primary keys** : there can only be a single PK field, and it must be a String!

What is generated with the above is as follows

```xml
<?xml version="1.0" encoding="UTF-8"?>
<myproduct>
    <people>
        <person personNum="1">
            <firstName>Bugs</firstName>
            <lastName>Bunny</lastName>
            <bestFriend>My</bestFriend>
        </person>
    </people>
</myproduct>
```

Here's the same example using JDO Annotations

```java
@PersistenceCapable(schema="/myproduct/people", table="person")
public class Person {
    @XmlAttribute
    private long personNum;

    @PrimaryKey
    private String firstName;

    private String lastName;

    private Person bestFiend;

    @XmlElementWrapper(name="phone-numbers")
    @XmlElement(name="phone-number")
    @Element(types=String.class)
    private Map phoneNumbers = new HashMap();

    ...
```

Here's the same example using JPA Annotations (with DataNucleus @Extension/@Extensions annotations)
TODO Add this example
29 HBase

29.1 HBase Datastores

DataNucleus supports persisting/retrieving objects to/from HBase datastores (using the datanucleus-hbase plugin, which makes use of the HBase/Hadoop jars). Simply specify your "connectionURL" as follows

```java
datanucleus.ConnectionURL=hbase[:{server}:{port}]
datanucleus.ConnectionUserName=
datanucleus.ConnectionPassword=
```

If you just specify the URL as hbase then you have a local HBase datastore, otherwise it tries to connect to the datastore at {server}:{port}. Alternatively just put "hbase" as the URL and set the zookeeper details in "hbase-site.xml" as normal. Then you then create your PMF/EMF as normal and use JDO/JPA as normal.

The jars required to use DataNucleus HBase persistence are datanucleus-core, datanucleus-api-jdo/datanucleus-api-jpa, datanucleus-hbase and hbase, hadoop-core, zookeeper.

There are tutorials available for use of DataNucleus with HBase for JDO and for JPA

Things to bear in mind with HBase usage :-

- Querying can be performed using JDOQL or JPQL. Some components of a filter are handled in the datastore, and the remainder in-memory. Currently any expression of a field (in the same table), or a literal are handled in-datastore, as are the operators &&, ||, >, >=, <, <=, ==, and !=.
- The "row key" will be the PK field(s) when using "application-identity", and the generated id when using "datastore-identity"

29.1.1 Field/Column Naming

By default each field is mapped to a single column in the datastore, with the family name being the name of the table, and the column name using the name of the field as its basis (but following JDO/JPA naming strategies for the precise column name). You can override this as follows

```java
@Column(name="{familyName}:{qualifierName}")
String myField;
```

replacing {familyName} with the family name you want to use, and {qualifierName} with the column name (qualifier name in HBase terminology) you want to use. Alternatively if you don't want to override the default family name (the table name), then you just omit the "{familyName}" part and simply specify the column name.
29.1.2 MetaData Extensions

Some metadata extensions (@Extension) have been added to DataNucleus to support some of HBase particular table creation options. The supported attributes at Table creation for a column family are:

- **bloomFilter**: An advanced feature available in HBase is Bloom filters, allowing you to improve lookup times given you have a specific access pattern. Default is NONE. Possible values are: Row -> use the row key for the filter, RowKey -> use the row key and column key (family + qualifier) for the filter.

- **inMemory**: The in-memory flag defaults to false. Setting it to true is not a guarantee that all blocks of a family are loaded into memory nor that they stay there. It is an elevated priority, to keep them in memory as soon as they are loaded during a normal retrieval operation, and until the pressure on the heap (the memory available to the Java-based server processes) is too high, at which time they need to be discarded by force.

- **maxVersions**: Per family, you can specify how many versions of each value you want to keep. The default value is 3, but you may reduce it to 1, for example, in case you know for sure that you will never want to look at older values.

- **keepDeletedCells**: ColumnFamilies can optionally keep deleted cells. That means deleted cells can still be retrieved with Get or Scan operations, as long these operations have a time range specified that ends before the timestamp of any delete that would affect the cells. This allows for point in time queries even in the presence of deletes. Deleted cells are still subject to TTL and there will never be more than "maximum number of versions" deleted cells. A new "raw" scan options returns all deleted rows and the delete markers.

- **compression**: HBase has pluggable compression algorithm, default value is NONE. Possible values GZ, LZO, SNAPPY.

- **blockCacheEnabled**: As HBase reads entire blocks of data for efficient I/O usage, it retains these blocks in an in-memory cache so that subsequent reads do not need any disk operation. The default of true enables the block cache for every read operation. But if your use case only ever has sequential reads on a particular column family, it is advisable that you disable it from polluting the block cache by setting it to false.

- **timeToLive**: HBase supports predicate deletions on the number of versions kept for each value, but also on specific times. The time-to-live (or TTL) sets a threshold based on the timestamp of a value and the internal housekeeping is checking automatically if a value exceeds its TTL. If that is the case, it is dropped during major compactions.

To express these options, a format similar to a properties file is used such as:

```plaintext
hbase.columnFamily.[family name to apply property on].[attribute] = {value}
```

where:

- **attribute**: One of the above defined attributes (inMemory, bloomFilter,...)
- **family name to apply property on**: The column family affected.
- **value**: Associated value for this attribute.

An example that would apply to the "meta" column family, that would set the bloom filter option to ROWKEY, and the in memory flag to true would look like:
@PersistenceCapable
@Extensions({
    @Extension(vendorName = "datanucleus", key = "hbase.columnFamily.meta.bloomFilter", value = "ROWKEY"),
    @Extension(vendorName = "datanucleus", key = "hbase.columnFamily.meta.inMemory", value = "true")
})
public class MyClass
{
    @PrimaryKey
    private long id;

    // column family data, name of attribute blob
    @Column(name = "data:blob")
    private String blob;

    // column family meta, name of attribute firstName
    @Column(name = "meta:firstName")
    private String firstName;

    // column family meta, name of attribute lastName
    @Column(name = "meta:lastName")
    private String lastName;

    [ ... getter and setter ... ]

29.1.3 References
Below are some references using this support

• Apache Hadoop HBase plays nicely with JPA
• HBase with JPA and Spring Roo
• Value Generator plugin for HBase and DataNucleus
30 MongoDB

30.1 MongoDB Datastores

DataNucleus supports persisting/retrieving objects to/from MongoDB datastores (using the datanucleus-mongodb plugin, which utilises the Mongo Java driver). Simply specify your "connectionURL" as follows

```
  datanucleus.ConnectionURL=mongodb:[{server}][/{dbName}],[{server2}],[{server3}]
```

For example, to connect to a local server, with database called "myMongoDB"

```
  datanucleus.ConnectionURL=mongodb:/myMongoDB
```

If you just specify the URL as mongodb then you have a local MongoDB datastore called "DataNucleus", otherwise it tries to connect to the datastore {dbName} at {server}. The multiple {server} option allows you to run against MongoDB replica sets. You then create your PMF/EMF as normal and use JDO/JPA as normal.

The jars required to use DataNucleus MongoDB persistence are datanucleus-core, datanucleus-api-jdo, datanucleus-api-jpa, datanucleus-mongodb and mongo-java-driver

There are tutorials available for use of DataNucleus with MongoDB for JDO and for JPA

Things to bear in mind with MongoDB usage :-

- Querying can be performed using JDOQL or JPQL. Some components of a filter are handled in the datastore, and the remainder in-memory. Currently any expression of a field (in the same table), or a literal are handled in-datastore, as are the operators &&, ||, >, >=, <, <=, ==, and !=.
- If you want a query to be runnable on a slave MongoDB instance then you should set the query extension (JDO) / hint (JPA) slave-ok as true, and when executed it can be run on a slave instance.
- All objects of a class are persisted to a particular "document" (specifiable with the "table" in metadata), and a field of a class is persisted to a particular "field" ( "column" in the metadata).
- Relations : DataNucleus stores the id of the related object(s) in a field of the owning object. When a relation is bidirectional both ends of the relation will store the relation information.
- Capped collections : you can specify the extension metadata key mongodb.capped.size as the number of bytes of the size of the collection for the class in question.
- If you want to specify the max number of connections per host with MongoDB then set the persistence property datanucleus.mongodb.connectionsPerHost
- If you want to specify the MongoDB threadsAllowedToBlockForConnectionMultiplier, then set the persistence property datanucleus.mongodb.threadsAllowedToBlockForConnectionMultiplier
30.1.1 Mapping : Embedded Persistable fields

When you have a field in a class that is of a persistable type you sometimes want to store it with the owning object. In this case you can use JDO/ JPA embedding of the field. DataNucleus offers two ways of performing this embedding

- The default is to store the object in the field as a sub-document (nested) of the owning document. Similarly if that sub-object has a field of a persistable type then that can be further nested.
- The alternative is to store each field of the sub-object as a field of the owning document (flat embedding). Similarly if that sub-object has a field of a persistable type then it can be flat embedded in the same way

```java
@PersistenceCapable
public class A {
    @Embedded
    B b;
    ...
}
```

This example uses the default embedding, using a nested document within the owner document, and could look something like this

```json
{ "name" : "A Name" ,
  "id" : 1 ,
  "b" : { "name" : "B name" ,
          "description" : "the description"}
}
```

The alternative would be

```java
@PersistenceCapable
public class A {
    @Embedded
    @Extension(vendorName="datanucleus", key="nested", value="false")
    B b;
    ...
}
```

and this will use flat embedding, looking something like this

```json
{ "name" : "A Name" ,
  "id" : 1 ,
  "b_name" : "B name" ,
  "b_description" : "the description"
}```
30.1.2 Mapping: Embedded Collection elements

When you have a field in a class that is of a Collection type you sometimes want to store it with the owning object. In this case you can use JDO/ JPA embedding of the field. So if we have

```java
@PersistenceCapable
public class A
{
    @Element(embedded="true")
    Collection<b> bs;

    ...
}
```

and would look something like this

```json
{
    "name" : "A Name",
    "id" : 1,
    "bs" : [
        {
            "name" : "B Name 1",
            "description" : "desc 1"
        },
        {
            "name" : "B Name 2",
            "description" : "desc 2"
        },
        {
            "name" : "B Name 3",
            "description" : "desc 3"
        }
    ]
}
```

30.1.3 References

Below are some references using this support

- Sasa Jovancic - Use JPA with MongoDb and Datanucleus
31 Google AppEngine/Datastore

31.1 Google AppEngine/Datastore

DataNucleus supports persisting/retrieving objects to/from Google AppEngine/Datastore (using the datanucleus-appengine plugin). Simply specify your connection details as follows

```
datanucleus.ConnectionURL=appengine:
```

Version 3.0.x of the datanucleus-appengine plugin is for use with DataNucleus v3.2/v3.3 (see SVN trunk of the GAE plugin). Since Google have been incredibly lazy in releasing this version of their appengine plugin, we have put one in the DataNucleus Maven repository which you can use with DataNucleus AccessPlatform 3.3.

Once you have specified your connection details you then create your PMF/EMF as normal and use JDO/JPA as normal. Things to bear in mind with AppEngine/Datastore usage :-

- Querying can be performed using JDOQL or JPQL. Some query constructs are evaluated in the datastore, and you can request that any others are handled in-memory. If you want queries to automatically evaluate unsupported constructs in-memory just set the persistence property `datanucleus.appengine.query.inMemoryWhenUnsupported` to "true"
- Relation Mode : by default all relations are owned meaning that when related objects are persisted they are owned by the original object and their "id" contains the Key of the owner object. You can set `datanucleus.appengine.relationDefault` to `unowned` and make all relations default to unowned, just like all other supported datastores

31.1.1 References

Some references that may be of some use

- GAE/J and DataNucleus v3 - Part 1
- GAE/J and DataNucleus v3 - Part 2
32 Neo4j

32.1 Neo4j Datastores

DataNucleus supports persisting/retrieving objects to/from embedded Neo4j graph datastores (using the datanucleus-neo4j plugin, which utilises the Neo4j Java driver). Simply specify your "connectionURL" as follows:

```
datanucleus.ConnectionURL=neo4j:{db_location}
```

For example:

```
datanucleus.ConnectionURL=neo4j:myNeo4jDB
```

You then create your PMF/EMF as normal and use JDO/JPA as normal.

The jars required to use DataNucleus Neo4j persistence are datanucleus-core, datanucleus-api-jdo, datanucleus-api-jpa, datanucleus-neo4j and neo4j

Note that this is for embedded Neo4j. This is because at the time of writing there is no binary protocol for connecting Java clients to the server with Neo4j. When that is available we would hope to support it.

There are tutorials available for use of DataNucleus with Neo4j for JDO and for JPA

Things to bear in mind with Neo4j usage:

- Querying can be performed using JDOQL or JPQL. Some components of a filter are handled in the datastore, and the remainder in-memory. Currently any expression of a field (in the same 'table'), or a literal are handled in-datastore, as are the operators &&, ||, >, >=, <, <=, ==, and !=. Also the majority of ordering and result clauses are evaluatable in the datastore, as well as query result range restrictions.
- When an object is persisted it becomes a Node in Neo4j. You define the names of the properties of that node by specifying the "column" name using JDO/JPA metadata.
- Any 1-1, 1-N, M-N, N-1 relation is persisted as a Relationship object in Neo4j and any positioning of elements in a List or array is preserved via a property on the Relationship.
- If you wanted to specify some neo4j.properties file for use of your embedded database then specify the persistence property datanucleus.ConnectionPropertiesFile set to the filename.
- This plugin is in prototype stage so would welcome feedback and, better still, some contributions to fully exploit the power of Neo4j. Register your interest on the DataNucleus Forum

32.1.1 Persistence Implementation

Let's take some example classes, and then describe how these are persisted in Neo4j.
public class Store
{
    @Persistent(primaryKey="true", valueStrategy="identity")
    long id;

    Inventory inventory;

    ...;
}

public class Inventory
{
    @Persistent(primaryKey="true", valueStrategy="identity")
    long id;

    Set<Product> products;

    ...;
}

public class Product
{
    @Persistent(primaryKey="true", valueStrategy="identity")
    long id;

    String name;

    double value;

    ...;
}

When we persist a Store object, which has an Inventory, which has three Product objects, then we get the following:

- **Node** for the `Store`, with the "id" is represented as the node id
- **Node** for the `Inventory`, with the "id" is represented as the node id
- **Relationship** between the `Store` Node and the `Inventory` Node, with the relationship type as "SINGLE_VALUED", and with the property `DN_FIELD_NAME` as "inventory"
- **Node** for `Product #1`, with properties for "name" and "value" as well as the "id" represented as the node id
- **Node** for `Product #2`, with properties for "name" and "value" as well as the "id" represented as the node id
- **Node** for `Product #3`, with properties for "name" and "value" as well as the "id" represented as the node id
- **Relationship** between the `Inventory` Node and the `Product #1` Node, with the relationship type "MULTI_VALUED" and the property `DN_FIELD_NAME` as "products"
- **Relationship** between the `Inventory` Node and the `Product #2` Node, with the relationship type "MULTI_VALUED" and the property `DN_FIELD_NAME` as "products"
- **Relationship** between the `Inventory` Node and the `Product #3` Node, with the relationship type "MULTI_VALUED" and the property `DN_FIELD_NAME` as "products"
Index in "DN_TYPES" for the Store Node with "class" as "mydomain.Store"

Index in "DN_TYPES" for the Inventory Node with "class" as "mydomain.Inventory"

Index in "DN_TYPES" for the Product Node with "class" as "mydomain.Product"

Note that, to be able to handle polymorphism more easily, if we also have a class Book that extends Product then when we persist an object of this type we will have two entries in "DN_TYPES" for this Node, one with "class" as "mydomain.Book" and one with "class" as "mydomain.Product" so we can interrogate the Index to find the real inheritance level of this Node.

32.1.2 Query Implementation

In terms of querying, a JDOQL/JPQL query is converted into a generic query compilation, and then this is attempted to be converted into a Neo4j "Cypher" query. Not all syntaxes are convertable currently and the query falls back to in-memory evaluation in that case.
33 JSON

33.1 JSON Datastores

DataNucleus supports persisting/retrieving objects to/from JSON documents (using the \texttt{datanucleus-json} plugin). Simply specify your "connectionURL" as follows

\begin{verbatim}
datanucleus.ConnectionURL=json:{url}
\end{verbatim}

replacing \texttt{"{url}"} with some URL of your choice (e.g. "http://www.mydomain.com/somepath/"). You then create your PMF/EMF as normal and use JDO/JPA as normal.

Things to bear in mind with JSON usage :-

- Querying can be performed using JDOQL or JPQL. Any filtering/ordering will be performed \textbf{in-memory}.
- Relations: DataNucleus stores the id of the related object(s) in the element of the field. If a relation is bidirectional then it will be stored at both ends of the relation; this facilitates easy access to the related object with no need to do a query to find it.

33.1.1 Mapping : HTTP Mapping

The persistence to JSON datastore is performed via HTTP methods. HTTP response codes are used to validate the success or failure to perform the operations. The JSON datastore must respect the following:

<table>
<thead>
<tr>
<th>Method</th>
<th>Operation</th>
<th>URL format</th>
<th>HTTP response code</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUT</td>
<td>update objects</td>
<td>/(primary key)</td>
<td>HTTP Code 201 (created), 200 (ok) or 204 (no content)</td>
</tr>
<tr>
<td>HEAD</td>
<td>locate objects</td>
<td>/(primary key)</td>
<td>HTTP 404 if the object does not exist</td>
</tr>
<tr>
<td>POST</td>
<td>insert objects</td>
<td>/</td>
<td>HTTP Code 201 (created), 200 (ok) or 204 (no content)</td>
</tr>
<tr>
<td>GET</td>
<td>fetch objects</td>
<td>/(primary key)</td>
<td>HTTP Code 200 (ok) or 404 if object does not exist</td>
</tr>
<tr>
<td>GET</td>
<td>retrieve extent of classes</td>
<td>/</td>
<td>HTTP Code 200 (ok) or 404 if no objects exist</td>
</tr>
<tr>
<td>DELETE</td>
<td>delete objects</td>
<td>/(primary key)</td>
<td>HTTP Code 202 (accepted), 200 (ok) or 204 (no content)</td>
</tr>
</tbody>
</table>
33.1.2 Mapping : Persistent Classes

<table>
<thead>
<tr>
<th>Metadata API</th>
<th>Extension Element Attachment</th>
<th>Extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDO</td>
<td>/jdo/package/class/extension</td>
<td>url</td>
<td>Defines the location of the resources/objects for the class</td>
</tr>
</tbody>
</table>

```xml
<jdo>
   <package name="org.datanucleus.samples.models.company">
      <class name="Person" detachable="true">
         <extension vendor-name="datanucleus" key="url" value="/Person"/>
      </class>
   </package>
</jdo>
```

In this example, the `url` extension identifies the Person resources/objects as `/Person`. The persistence operations will be relative to this path. e.g `/Person/{primary key}` will be used for PUT (update), GET (fetch) and DELETE (delete) methods.
34 Amazon S3

34.1 Amazon Simple Storage Service Datastores

DataNucleus supports persisting/retrieving objects to/from Amazon Simple Storage Service (using the datanucleus-json plugin). Simply specify your connection details as follows:

```
datanucleus.ConnectionUserName={Access Key ID}
datanucleus.ConnectionPassword={Secret Access Key}
datanucleus.cloud.storage.bucket={bucket}
```

You then create your PMF/EMF as normal and use JDO/JPA as normal.

Things to bear in mind with Amazon S3 usage:

- Querying can be performed using JDOQL or JPQL. Any filtering/ordering will be performed in-memory.

34.1.1 References

Below are some references using this support:

- Simple Integration of Datanucleus 2.0.0 + AmazonS3
35 GoogleStorage

35.1 Google Storage Datastore

DataNucleus supports persisting/retrieving objects to/from Google Storage (using the datanucleus-json plugin). Simply specify your connection details as follows:

```
datanucleus.ConnectionURL=googlestorage:http://commondatastorage.googleapis.com/
datanucleus.ConnectionUserName={Access Key ID}
datanucleus.ConnectionPassword={Secret Access Key}
datanucleus.cloud.storage.bucket={bucket}
```

You then create your PMF/EMF as normal and use JDO/JPA as normal.

Things to bear in mind with GoogleStorage usage:

- Querying can be performed using JDOQL or JPQL. Any filtering/ordering will be performed in-memory.
36 LDAP

36.1 LDAP Datastores

DataNucleus supports persisting/retrieving objects to/from LDAP datastores (using the `datanucleus-ldap` plugin). If you wish to help out development of this plugin either by contributing or by sponsoring particular functionality please contact us via the DataNucleus Forum.

36.1.1 Datastore Connection

The following persistence properties will connect to an LDAP running on your local machine

```properties
datanucleus.ConnectionDriverName=com.sun.jndi.ldap.LdapCtxFactory
datanucleus.ConnectionURL=ldap://localhost:10389
datanucleus.ConnectionUserName=uid=admin,ou=system
datanucleus.ConnectionPassword=secret
```

So you create your `PersistenceManagerFactory` or `EntityManagerFactory` with these properties. Thereafter you have the full power of the JDO or JPA APIs at your disposal, for your LDAP datastore.

36.1.2 Queries

Access Platform allows you to query the objects in the datastore using the following

- **JDOQL** - language based around the objects that are persisted and using Java-type syntax
- **JPQL** - language based around the objects that are persisted and using SQL-like syntax

Queries are evaluated in-memory.

36.1.3 Mapping : LDAP Datastore Mapping

When persisting a Java object to an LDAP datastore clearly the user would like some control over where and how in the LDAP DIT (directory information tree) we are persisting the object. In general Java objects are mapped to LDAP entries and fields of the Java objects are mapped to attributes of the LDAP entries.

The following Java types are supported and stored as single-valued attribute to the LDAP entry:

- String, primitives (like int and double), wrappers of primitives (like `java.util.Long`, `java.util.BigDecimal`, `java.util.BigInteger`, `java.util.UUID`)
- `boolean` and `java.lang.Boolean` are converted to RFC 4517 "boolean" syntax (TRUE or FALSE)
- `java.util.Date` and `java.util.Calendar` are converted to RFC 4517 "generalized time" syntax

Arrays, Collections, Sets and Lists of these data types are stored as multi-valued attributes. Please note that when using Arrays and Lists no order could be guaranteed and no duplicate values are allowed!
36.1.4 Mapping: Relationships

By default PersistenceCapable objects are stored as separate LDAP entries. There are some options how to persist relationship references between PersistenceCapable objects:

- DN matching
- Attribute matching
- LDAP hierarchies

It is also possible to store PersistenceCapable objects embedded.

36.1.5 Examples

Here's an example using JDO XML MetaData:

```xml
<jdo>
  <package name="org.datanucleus.samples.models.company">
    <class name="Group" table="ou=Groups,dc/example,dc=com" schema="top,groupOfNames" detachable="true">
      <field name="name" column="cn" primary-key="true" />
      <field name="users" column="member" />
    </class>

    <class name="Person" table="ou/Users,dc/example,dc=com" schema="top,person,organizationalPerson,inetOrgPerson" detachable="true">
      <field name="personNum" column="cn" primary-key="true" />
      <field name="firstName" column="givenName" />
      <field name="lastName" column="sn" />
    </class>
  </package>
</jdo>
```

For the class as a whole we use the **table** attribute to set the distinguished name of the container under which to store objects of a type. So, for example, we are mapping all objects of class Group as subordinates to "ou=Groups,dc=example,dc=com". You can also use the extension "dn" to specify the same thing.

For the class as a whole we use the **schema** attribute to define the object classes of the LDAP entry. So, for example, all objects of type Person are mapped to the common "top,person,organizationalPerson/inetOrgPerson" object classes in LDAP. You can also use the extension "objectClass" to specify the same thing.

For each field we use the **column** attribute to define the LDAP attribute that we are mapping this field to. So, for example, we map the Group "name" to "cn" in our LDAP. You can also use the extension "attribute" to specify the same thing.

Some resulting LDAP entries would look like this:
Here's the same example using JDO Annotations:

```java
@PersistenceCapable(table = "ou=Groups,dc=example,dc=com", schema = "top,groupOfNames")
public class Group {
    @PrimaryKey
    @Column(name = "cn")
    String name;

    @Column(name = "member")
    protected Set<Person> users = new HashSet<Person>();
}

@PersistenceCapable(table = "ou=Users,dc=example,dc=com", schema = "top,person,organizationalPerson/inetOrgPerson")
public class Person {
    @PrimaryKey
    @Column(name = "cn")
    private long personNum;

    @Column(name = "givenName")
    private String firstName;

    @Column(name = "sn")
    private String lastName;
}
```

Here's the same example using JPA Annotations:
36.1.6 Known Limitations

The following are known limitations of the current implementation

- Datastore Identity is not currently supported
- Optimistic checking of versions is not supported
- Identity generators that operate using the datastore are not supported
- Cannot map inherited classes to the same LDAP type
37 Relations by DN

37.1 LDAP : Relationship Mapping by DN

A common way to model relationships between LDAP entries is to put the LDAP distinguished name of the referenced LDAP entry to an attribute of the referencing LDAP entry. For example entries with object class groupOfNames use the attribute member which contains distinguished names of the group members.

We just describe 1-N relationship mapping here and distinguish between unidirectional and bidirectional relationships. The metadata for 1-1, N-1 and M-N relationship mapping looks identical, the only difference is whether single-valued or multi-valued attributes are used in LDAP to store the relationships.

- Unidirectional
- Bidirectional

37.2 1-N Unidirectional

We use the following example LDAP tree and Java classes:

```
dc=example,dc=com
|-- ou=Departments
  |-- cn=Sales
  |-- cn=Engineering
  |-- ...
|-- ou=Employees
  |-- cn=Bugs Bunny
  |-- cn=Daffy Duck
  |-- cn=Speedy Gonzales
  |-- ...
```

We have a flat LDAP tree with one container for all the departments and one container for all the employees. We have two Java classes, Department and Employee. The Department class contains a Collection of type Employee. The Employee knows nothing about the Department it belongs to.

There are 2 ways that we can persist this relationship in LDAP because the DN reference could be stored at the one or at the other LDAP entry.

37.2.1 Owner Object Side

The obvious way is to store the reference at the owner object side, in our case at the department entry. This is possible since LDAP allows multi-valued attributes. The example department entry looks like this:
Our JDO metadata looks like this:

```xml
<jdo>
  <package name="com.example">
    <class name="Department" table="ou=Departments,dc=example,dc=com" schema="top,groupOfNames">
      <field name="name" primary-key="true" column="cn" />
      <field name="employees" column="member">
        <extension vendor-name="datanucleus" key="empty-value" value="uid=admin,ou=system"/>
      </field>
    </class>
    <class name="Employee" table="ou=Employees,dc=example,dc=com" schema="top,person,organizationalPerson,inetOrgPerson">
      <field name="fullName" primary-key="true column="cn" />
      <field name="firstName" column="givenName" />
      <field name="lastName" column="sn" />
    </class>
  </package>
</jdo>
```

So we define that the attribute `member` should be used to persist the relationship of field `employees`. Note: We use the extension `empty-value` here. The groupOfNames object class defines the member attribute as mandatory attribute. In case where you remove all the employees from a department would delete all member attributes which isn't allowed. In that case DataNucleus adds this empty value to the member attribute. This value is also filtered when DataNucleus reads the object from LDAP.

### 37.2.2 Non-Owner Object Side

Another possible way is to store the reference at the non-owner object side, in our case at the employee entry. The example employee entry looks like this:

```
dn: cn=Bugs Bunny,ou=Employees,dc=example,dc=com
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
cn: Bugs Bunny
givenName: Bugs
sn: Bunny
departmentNumber: cn=Sales,ou=Departments,dc=example,dc=com
```

Our JDO metadata looks like this:
We need to define the relationship at the department metadata because the employee doesn't know about the department it belongs to. With the `<element>` tag we specify that the relationship should be persisted at the other side, the `column` attribute defines the LDAP attribute to use. In this case the relationship is persisted in the `departmentNumber` attribute at the employee entry.

### 37.3 1-N Bidirectional

We use the following example LDAP tree and Java classes:

```java
<package name="com.example">
    <class name="Department" table="ou=Departments,dc=example,dc=com" schema="top,groupOfNames">
        <field name="name" primary-key="true" column="cn" />
        <field name="employees">
            <element column="departmentNumber" />
        </field>
    </class>

    <class name="Employee" table="ou=Employees,dc=example,dc=com" schema="top,person,organizationalPerson,inetOrgPerson">
        <field name="fullName" primary-key="true" column="cn" />
        <field name="firstName" column="givenName" />
        <field name="lastName" column="sn" />
    </class>
</package>
```

We have a flat LDAP tree with one container for all the departments and one container for all the employees. We have two Java classes, `Department` and `Employee`. The `Department` class contains a Collection of type `Employee`. Now each `Employee` has a reference to its `Department`.

It is possible to persist this relationship on both sides.
In this case we store the relation at the department entry side in a multi-valued attribute `member`. Now the employee metadata contains a department field that is `mapped-by` the employees field of department.

Note: We use the extension `empty-value` here. The groupOfNames object class defines the member attribute as mandatory attribute. In case where you remove all the employees from a department would delete all member attributes which isn't allowed. In that case DataNucleus adds this empty value to the member attribute. This value is also filtered when DataNucleus reads the object from LDAP.
38 Relations by Attribute

38.1 LDAP: Relationship Mapping by Attribute

Another way to model relationships between LDAP entries is to use attribute matching. This means two entries have the same attribute values. An example of this type of relationship is used by posixGroup and posixAccount object classes were posixGroup.memberUid points to posixAccount.uid.

We just describe 1-N relationship mapping here and distinguish between unidirectional and bidirectional relationships. The metadata for 1-1, N-1 and M-N relationship mapping looks identical, the only difference is whether single-valued or multi-valued attributes are used in LDAP to store the relationships.

- Unidirectional
- Bidirectional

38.2 1-N Unidirectional

We use the following example LDAP tree and Java classes:

```java
public class Department {
    String name;
    Set<Employee> employees;
}

public class Employee {
    String firstName;
    String lastName;
    String fullName;
    String uid;
}
```

We have a flat LDAP tree with one container for all the departments and one container for all the employees. We have two Java classes, Department and Employee. The Department class contains a Collection of type Employee. The Employee knows nothing about the Department it belongs to.

There are 2 ways that we can persist this relationship in LDAP because the reference could be stored at the one or at the other LDAP entry.

38.2.1 Owner Object Side

One way is to store the reference at the owner object side, in our case at the department entry. This is possible since LDAP allows multi-valued attributes. The example department entry looks like this:
38.2 Relations by Attribute

Our JDO metadata looks like this:

```xml
<jdo>
  <package name="com.example">
    <class name="Department" table="ou=Departments,dc=example,dc=com" schema="top,organizationalUnit,extensibleObject">
      <field name="name" primary-key="true" column="ou" />
      <field name="employees" column="memberUid">
        <join column="uid" />
      </field>
    </class>
    <class name="Employee" table="ou=Employees,dc=example,dc=com" schema="top,person,organizationalPerson,inetOrgPerson">
      <field name="fullName" primary-key="true" column="cn" />
      <field name="firstName" column="givenName" />
      <field name="lastName" column="sn" />
      <field name="uid" column="uid" />
    </class>
  </package>
</jdo>
```

So we define that the attribute `memberUid` at the department entry should be used to persist the relationship of field `employees`.

The important thing here is the `<join>` tag and its `column`. Firstly it signals DataNucleus to use attribute mapping. Secondly it specifies the attribute at the other side that should be used for relationship mapping. In our case, when we establish a relationship between a `Department` and an `Employee`, the `uid` value of the employee entry is stored in the `memberUid` attribute of the department entry.

### 38.2.2 Non-Owner Object Side

Another possible way is to store the reference at the non-owner object side, in our case at the employee entry. The example employee entry looks like this:

```xml
dn: ou=Sales,ou=Departments,dc=example,dc=com
objectClass: top
objectClass: organizationalUnit
objectClass: extensibleObject
ou: Sales
memberUid: bbunny
memberUid: dduck
```
38 Relations by Attribute

Our JDO metadata looks like this:

```xml
<jdo>
  <package name="com.example">
    <class name="Department" table="ou=Departments,dc=example,dc=com" schema="top,organizationalUnit">
      <field name="name" primary-key="true" column="ou" />
      <field name="employees">
        <element column="departmentNumber" />
        <join column="ou" />
      </field>
    </class>
    <class name="Employee" table="ou=Employees,dc=example,dc=com" schema="top,person,organizationalPerson/inetOrgPerson">
      <field name="fullName" primary-key="true" column="cn" />
      <field name="firstName" column="givenName" />
      <field name="lastName" column="sn" />
      <field name="uid" column="uid" />
    </class>
  </package>
</jdo>
```

We need to define the relationship at the department metadata because the employee doesn't know about the department it belongs to.

With the `<element>` tag we specify that the relationship should be persisted at the other side and the `column` attribute defines the LDAP attribute to use. In this case the relationship is persisted in the `departmentNumber` attribute at the employee entry.

The important thing here is the `<join>` tag and its `column`. As before it signals DataNucleus to use attribute mapping. Now, as the relation is persisted at the other side, it specifies the attribute at this side that should be used for relationship mapping. In our case, when we establish a relationship between a Department and an Employee, the `ou` value of the department entry is stored in the `departmentNumber` attribute of the employee entry.

### 38.3 1-N Bidirectional

We use the following example LDAP tree and Java classes:
We have a flat LDAP tree with one container for all the departments and one container for all the employees. We have two Java classes, `Department` and `Employee`. The `Department` class contains a Collection of type `Employee`. Now each `Employee` has a reference to its `Department`.

It is possible to persist this relationship on both sides.

In this case we store the relation at the employee entry side in a single-valued attribute `departmentNumber`. With the `<join>` tag and its `column` we specify that the `ou` value of the
department entry should be used as join value. Also note that employee field of Department is mapped-by the department field of the Employee.
39 Relations by Hierarchy

39.1 LDAP: Relationship Mapping by Hierarchy

As LDAP is a hierarchical data store it is possible to model relationships between LDAP entries using hierarchies. For example organisational structures like departments and their employees are often modeled hierarchical in LDAP. It is possible to map 1-1 and N-1/1-N relationships using LDAP hierarchies.

The main challenge with hierarchical mapping is that the distinguished name (DN) of children depends on the DN of their parent. Therefore each child class needs a reference to the parent class. The parent class metadata defines a (fixed) LDAP DN that is used as container for all objects of the parent type. The child class metadata contains a dynamic part in its DN definition. This dynamic part contains the name of the field holding the reference to the parent object, the name is surrounded by curly braces. This dynamic DN is the indicator for DataNucleus to use hierarchical mapping. The reference field itself won’t be persisted as attribute because it is used as dynamic parameter. If you query for child objects DataNucleus starts a larger LDAP search to find the objects (the container DN of the parent class as search base and subtree scope).

Note: Child objects are automatically dependent. If you delete the parent object all child objects are automatically deleted. If you null out the child object reference in the parent object or if you remove the child object from the parents collection, the child object is automatically deleted.

39.2 N-1 Unidirectional

This kind of mapping could be used if your LDAP tree has a huge number of child objects and you only work with these child objects.

We use the following example LDAP tree and Java classes:

```java
dc=example,dc=com
|-- ou=Sales
|  |-- cn=Bugs Bunny
|  |   |-- cn=Daffy Duck
|  |   |   |-- ...  
|  |   |-- ou=Engineering
|  |   |   |-- cn=Speedy Gonzales
|  |   |   |   |-- ... 

public class Department {
    String name;
}

public class Employee {
    String firstName;
    String lastName;
    String fullName;
    Department department;
}
```

In the LDAP tree we have departments (Sales and Engineering) and each department holds some associated employees. In our Java classes each Employee object knows its Department but not vice-versa.

The JDO metadata looks like this:
The Department objects are persisted directly under dc=example,dc=com. The Employee class has a dynamic DN definition {department}. So the DN of the Department instance is used as container for Employee objects.

### 39.3 N-1 (1-N) Bidirectional

If you need a reference from the parent object to the child objects you need to define a bidirectional relationship.

The example LDAP tree and Java classes looks like this:

```plaintext
dc=example,dc=com
|-- ou=Sales
| |-- cn=Bugs Bunny
| |-- cn=Daffy Duck
| |-- ...
|-- ou=Engineering
| |-- cn=Speedy Gonzales
| |-- ...
|-- ...
```

Now the Department class has a Collection containing references to its Employees.

The JDO metadata looks like this:
We added a new `employees` field to the `Department` class that is `mapped-by` the department field of the `Employee` class.

Please note: When loading the parent object all child object are loaded immediately. For a large number of child entries this may lead to performance and/or memory problems.

### 39.4 1-1 Unidirectional

1-1 unidirectional mapping is very similar to N-1 unidirectional mapping.

We use the following example LDAP tree and Java classes:

```java
<packagename="com.example">
  <class name="Department" table="dc=example,dc=com" schema="top,organizationalUnit">
    <field name="name" primary-key="true" column="ou" />
    <field name="employees" mapped-by="department" />
  </class>

  <class name="Employee" table="{department}" schema="top,person,organizationalPerson,inetOrgPerson">
    <field name="fullName" primary-key="true" column="cn" />
    <field name="firstName" column="givenName" />
    <field name="lastName" column="sn" />
    <field name="department" />
  </class>
</package>
</jdo>
```

In the LDAP tree we have persons and each person has one account. Each `Account` object knows to which `Person` it belongs to, but not vice-versa.

The JDO metadata looks like this:
The `Person` objects are persisted directly under `ou=People,dc=example,dc=com`. The `Account` class has a dynamic DN definition `{person}`. So the DN of the Person instance is used as container for the Account object.

### 39.5 1-1 Bidirectional

If you need a reference from the parent class to the child class you need to define a bidirectional relationship.

The example LDAP tree and Java classes looks like this:

```
dc=example,dc=com                                      public class Person {
 |                                              String firstName;
 | |-- ou=People                                               String lastName;
 | |                                                String fullName;
 | |   |-- cn=Bugs Bunny                                      Account account;
 | |   |   |-- uid=bbunny                                     }
 | |   |   |-- ...                                          
 | |   |-- cn=Daffy Duck                                     public class Account {
 | |   |   |-- uid=dduck                                      String uid;
 | |   |   |-- ...                                          String password;
 | |   | |-- ...                                           Person person;
 | |   |   |-- ...                                          }
```

Now the `Person` class has a reference to its `Account`.

The JDO metadata looks like this:
We added a new `account` field to the Person class that is `mapped-by` the person field of the Account class.
40 Embedded Objects

40.1 LDAP: Embedded Objects

With JDO it is possible to persist fields as embedded. This may be useful for LDAP datastores where often many attributes are stored within one entry however logically they describe different objects.

Let's assume we have the following entry in our directory:

dn: cn=Bugs Bunny,ou=Employees,dc=example,dc=com
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
cn: Bugs Bunny
givenName: Bugs
sn: Bunny
postalCode: 3578
l: Hollywood
street: Sunset Boulevard
uid: bbunny
userPassword: secret

This entry contains multiple types of information: a person, its address and its account data. So we will create the following Java classes:

```java
public class Employee {
    String firstName;
    String lastName;
    String fullName;
    Address address;
    Account account;
}

public class Address {
    int zip;
    String city
    String street;
}

public class Account {
    String id;
    String password;
}
```

The JDO metadata to map these objects to one LDAP entry would look like this:
<package name="com.example">
  <class name="Person" table="ou=Employees,dc=example,dc=com" schema="top,person,organizationalPerson">
    <field name="fullName" primary-key="true" column="cn" />
    <field name="firstName" column="givenName" />
    <field name="lastName" column="sn" />
    <field name="account">
      <embedded null-indicator-column="uid">
        <field name="id" column="uid" />
        <field name="password" column="userPassword" />
      </embedded>
    </field>
    <field name="address">
      <embedded null-indicator-column="l">
        <field name="zip" column="postalCode" />
        <field name="city" column="l" />
        <field name="street" column="street" />
      </embedded>
    </field>
  </class>
  <class name="Account" embedded-only="true">
    <field name="uid" />
    <field name="password" />
  </class>
  <class name="Address" embedded-only="true">
    <field name="zip" />
    <field name="city" />
    <field name="street" />
  </class>
</package>
41 NeoDatis

41.1 NeoDatis Datastores

NeoDatis is an object-oriented database for Java and .Net. It is simple and fast and supports various query mechanisms.

DataNucleus supports persisting/retrieving objects to NeoDatis datastores (using the datanucleus-neodatis plugin). If you wish to help out in this effort either by contributing or by sponsoring particular functionality please contact us via the DataNucleus Forum.

The jars required to use DataNucleus NeoDatis persistence are datanucleus-core, datanucleus-api-jdo, datanucleus-api-jpa, datanucleus-neodatis and neodatis

41.1.1 Datastore Connection

DataNucleus supports 2 modes of operation of neodatis - file-based, and client-server based. In order to do so and to fit in with the JDO/JPA APIs we have defined the following means of connection.

The following persistence properties will connect to a file-based NeoDatis running on your local machine

\[\text{datanucleus.ConnectionURL=neodatis:file:neodatisdb.odb}\]

Replacing "neodatis.odb" by your filename for the datastore, and can be absolute OR relative.

The following persistence properties will connect to embedded-server-based NeoDatis running with a local file

\[\text{datanucleus.ConnectionURL=neodatis:server:{my_neodatis_file}}\]
\[\text{datanucleus.ConnectionUserName=}\]
\[\text{datanucleus.ConnectionPassword=}\]

The filename \{my_neodatis_file\} can be absolute OR relative.

The following persistence properties will connect as a client to a TCP/IP NeoDatis Server

\[\text{datanucleus.ConnectionURL=neodatis:{neodatis_host}:{neodatis_port}/{identifier}}\]
\[\text{datanucleus.ConnectionUserName=}\]
\[\text{datanucleus.ConnectionPassword=}\]

NeoDatis doesn't itself use such URLs so it was necessary to define this DataNucleus-specific way of addressing NeoDatis.

So you create your PersistenceManagerFactory or EntityManagerFactory with these properties. Thereafter you have the full power of the JDO or JPA APIs at your disposal, for your NeoDatis datastore.

41.1.2 Queries

Access Platform allows you to query the objects in the datastore using the following

- **JDOQL** - language based around the objects that are persisted and using Java-type syntax
- **JPQL** - language based around the objects that are persisted and using SQL-like syntax
• **Native** - NeoDatis' own type-safe query language
• **Criteria** - NeoDatis' own Criteria query language

### 41.1.3 Queries: NeoDatis Native Queries

Note that if you choose to use NeoDatis Native Queries then these are not portable to any other datastore. Use JDOQL/JPQL for portability.

NeoDatis provides its own “native” query interface, and if you are using the JDO API you can utilise this for querying. To take a simple example

```java
// Find all employees older than 31
Query q = pm.newQuery("Native", new NativeQuery()
{
    public boolean match(Object e)
    {
        if (!(e instanceof Employee))
        {
            return false;
        }
        return ((Employee)e).getAge() >= 32;
    }
    public Class getObjectType()
    {
        return Employee.class;
    }
});
List results = (List)q.execute();
```

So we are utilising the JDO API to generate a query and passing in the NeoDatis "NativeQuery".

### 41.1.4 Queries: NeoDatis Criteria Queries

Note that if you choose to use NeoDatis Criteria Queries then these are not portable to any other datastore. Use JDOQL/JPQL for portability.

NeoDatis provides its own “criteria” query interface, and if you are using the JDO API you can utilise this for querying. To take a simple example

```java
// Find all employees older than 31
Query q = pm.newQuery("Criteria", new CriteriaQuery(Employee.class, Where.ge("age", 32)));
List results = (List)q.execute();
```

So we are utilising the JDO API to generate a query and passing in the NeoDatis "CriteriaQuery".
41.1.5 Known Limitations

The following are known limitations of the current implementation

- NeoDatis doesn’t have the concept of an "unloaded" field and so when you request an object from the datastore it comes with its graph of objects. Consequently there is no "lazy loading" and the consequent impact that can have on memory utilisation.
Java Data Objects (JDO) defines an interface (or API) to persist normal Java objects (or POJO's in some peoples terminology) to a datastore. JDO doesn't define the type of datastore; it is datastore-agnostic. You would use the same interface to persist your Java object to RDBMS, or OODBMS, or XML, or whatever form of data storage. The whole point of using such a standard interface is that users can, in principle, swap between implementations of JDO without changing their code. Make sure you have datanucleus-api-jdo.jar in your CLASSPATH for this API.

The process of mapping a class can be split into the following areas

- JDO categorises classes into 3 types, so you firstly decide which type your class is, and mark the class in that category
- JDO allows fields/properties to be defined for persistence, and you can control which of these are persisted, and how they are persisted.
- Some datastores allow a level of mapping between the object-oriented world and the structure of the datastore, and for this you can define (a level of) Object-Relational Mapping (ORM)

Note that with DataNucleus, you can map your classes using JDO MetaData (XML/Annotations) OR using JPA MetaData (XML/Annotations) and still use the JDO API with these classes.

At runtime with JDO you start with the creation of a PersistenceManagerFactory (PMF) which provides the connectivity to the datastore. The connection to the datastore is dependent on a set of persistence properties defining the datastore location, URL etc as well as behaviour of the persistence process.

With JDO, to persist/retrieve objects you require a PersistenceManager (PM) that provides the interface to persistence and querying of the datastore. You can perform persistence and querying within a transaction if required, or just use it non-transactionally.

JDO allows querying of the datastore using a range of query languages. The most utilised is JDOQL providing an object-oriented form of querying, whereas some datastores also permit SQL.

If in doubt about how things fit together, please make use of the JDO Tutorial

If you just want to get the JDO API javadocs, then you can access those here (Apache JDO)

42.1.1 JDO References

- Apache JDO
- JDO 3.0 Specification
- JDO 2.2 Specification
- JDO 2.1 Specification
- JDO 2.0 Specification
• JDO 1.0 Specification
• JDO 3.0 Javados
• JDO 2.0 Javados
• JDO 1.1 Javados
• Apache JDO mailing lists
• ORM comparison: JDO v. JPA
43 Class Mapping

43.1 JDO : Class Mapping

The first thing to decide when implementing your persistence layer is which classes are to be persisted. Let's take a sample class (Hotel) as an example. We can define a class as persistable using either annotations in the class, or XML metadata.

To achieve the above aim with XML metadata, we do this

```xml
<class name="Hotel">
  ...
</class>
```

Alternatively, using JDO Annotations, like this

```java
@PersistenceCapable
public class Hotel {
  ...
}
```

See also :-

- MetaData reference for <class> element
- Annotations reference for @PersistenceCapable

43.1.1 Persistence-Aware Classes

With JDO persistence all classes that are persisted have to be identified in XML or annotations as shown above. In addition, if any of your other classes access the fields of these persistable classes directly then these other classes should be defined as PersistenceAware. You do this as follows

```xml
<class name="MyClass" persistence-modifier="persistence-aware">
  ...
</class>
```

or with annotations

```java
@PersistenceAware
public class MyClass {
  ...
}
```

See also :-

- Annotations reference for @PersistenceAware
43.1.2 Read-Only

You can, if you wish, make a class *read-only*. This is a DataNucleus extension and you set it as follows:

```xml
<class name="MyClass">
    <extension vendor-name="datanucleus" key="read-only" value="true"/>
</class>
```

or with annotations:

```java
@PersistenceCapable
@Extension(vendorName="datanucleus", key="read-only", value="true")
public class MyClass
{
    ...
}
```
44 Datastore Identity

44.1 JDO : Datastore Identity

With **datastore identity** you are leaving the assignment of id's to DataNucleus and your class will **not** have a field for this identity - it will be added to the datastore representation by DataNucleus. It is, to all extents and purposes a **surrogate key** that will have its own column in the datastore. To specify that a class is to use **datastore identity** with JDO, you add the following to the MetaData for the class.

```xml
<class name="MyClass" identity-type="datastore">
  ...
</class>
```

or using JDO annotations

```java
@PersistenceCapable(identityType=IdentityType.DATASTORE)
public class MyClass
{
  ...
}
```

So you are specifying the **identity-type** as **datastore**. You don't need to add this because **datastore** is the default, so in the absence of any value, it will be assumed to be 'datastore'.

When you have an inheritance hierarchy, you should specify the identity type in the base class for the inheritance tree. This is then used for all persistent classes in the tree.

44.1.1 Generating identities

**JDO2**

By choosing **datastore identity** you are handing the process of identity generation to the JDO implementation. This does not mean that you haven't got any control over how it does this. JDO 2 defines many ways of generating these identities and DataNucleus supports all of these and provides some more of its own besides.

Defining which one to use is a simple matter of adding a MetaData element to your classes definition, like this

```xml
<class name="MyClass" identity-type="datastore">
  <datastore-identity strategy="sequence" sequence="MY_SEQUENCE"/>
  ...
</class>
```

```xml
<class name="MyClass" identity-type="datastore">
  <datastore-identity strategy="identity"/>
  ...
</class>
```
or using annotations, for example

```java
@PersistenceCapable
@DatastoreIdentity(strategy="sequence", sequence="MY_SEQUENCE")
public class MyClass {
    ...
}
```

Some of the datastore identity strategies require additional attributes, but the specification is straightforward.

See also:
- Identity Generation Guide - strategies for generating ids
- MetaData reference for `<datastore-identity>` element
- Annotations reference for @DatastoreIdentity

### 44.1.2 Accessing the Identity

When using datastore identity, the class has no associated field so you can't just access a field of the class to see its identity - if you need a field to be able to access the identity then you should be using application identity. There are, however, ways to get the identity for the datastore identity case, if you have the object.

```java
Object id = pm.getObjectId(obj);

Object id = JDOHelper.getObjectId(obj);
```

You should be aware however that the "identity" is in a complicated form, and is not available as a simple integer value for example. Again, if you want an identity of that form then you should use application identity

### 44.1.3 DataNucleus Implementation

When implementing datastore identity all JDO implementations have to provide a public class that represents this identity. If you call `pm.getObjectId(...)` for a class using datastore identity you will be passed an object which, in the case of DataNucleus will be of type `org.datanucleus.identity.OIDImpl`. If you were to call "toString()" on this object you would get something like

```
1[OID]mydomain.MyClass
This is made up of :-
    1 = identity number of this object
    class-name
```

The definition of this datastore identity is JDO implementation dependent. As a result you should not use the `org.datanucleus.identity.OID` class in your application if you want to remain implementation independent

DataNucleus allows you the luxury of being able to provide your own datastore identity class so you can have whatever formatting you want for identities.
44.1.4 Accessing objects by Identity

If you have the JDO identity then you can access the object with that identity like this

```java
Object obj = pm.getObjectById(id);
```

You can also access the object from the object class name and the toString() form of the datastore identity (e.g. "1[OID]mydomain.MyClass") like this

```java
Object obj = pm.getObjectById(MyClass.class, mykey);
```
45 Application Identity

45.1 JDO : Application Identity

With application identity you are taking control of the specification of id's to DataNucleus. Application identity requires a primary key class (unless you have a single primary-key field in which case the PK class is provided for you), and each persistent capable class may define a different class for its primary key, and different persistent capable classes can use the same primary key class, as appropriate. With application identity the field(s) of the primary key will be present as field(s) of the class itself. To specify that a class is to use application identity, you add the following to the MetaData for the class.

```xml
<class name="MyClass" objectid-class="MyIdClass">
   <field name="myPrimaryKeyField" primary-key="true"/>
   ...
</class>
```

For JDO we specify the primary-key and objectid-class. The objectid-class is optional, and is the class defining the identity for this class (again, if you have a single primary-key field then you can omit it). Alternatively, if we are using annotations

```java
@PersistenceCapable(objectIdClass=MyIdClass.class)
public class MyClass
{
    @Persistent(primaryKey="true")
    private long myPrimaryKeyField;
}
```

When you have an inheritance hierarchy, you should specify the identity type in the base instantiable class for the inheritance tree. This is then used for all persistent classes in the tree. This means that you can have superclass(es) using application-identity without any identity fields/properties but using subclass-table inheritance, and then the base instantiable class is the first persistable class which has the identity field(s).

See also :

- MetaData reference for <field> element
- Annotations reference for @Persistent

45.1.1 Primary Key

Using application identity requires the use of a Primary Key class. When you have a single primary-key field a built-in class is available meaning you don’t need to define this class. This is referred to as SingleFieldIdentity. We strongly recommend not to specify the PK class when you have a single PK field since these built-in classes are likely more optimised. Where the class has multiple fields that form the primary key a Primary Key class must be provided.

See also :

- Primary Key Guide - user-defined and built-in primary keys
45.1.2 Compound Identity
Where one of the fields that is primary-key of your class is a persistable object you have something known as compound identity since the identity of this class contains the identity of a related class. Please refer to the docs for Compound Identity

45.1.3 Generating identities
By choosing application identity you are controlling the process of identity generation for this class. This does not mean that you have a lot of work to do for this. JDO defines many ways of generating these identities and DataNucleus supports all of these and provides some more of its own besides.
See also :-
• Identity Generation Guide - strategies for generating ids

45.1.4 Accessing the Identity
When using application identity, the class has associated field(s) that equate to the identity. As a result you can simply access the values for these field(s). Alternatively you could use a JDO identity-independent way

Object id = pm.getObjectId(obj);

Object id = JDOHelper.getObjectId(obj);

45.1.5 Changing Identities
JDO allows implementations to support the changing of the identity of a persisted object. This is an optional feature and DataNucleus doesn't currently support it.

45.1.6 Accessing objects by Identity
If you have the JDO identity then you can access the object with that identity like this

Object obj = pm.getObjectById(id);

If you are using SingleField identity then you can access it from the object class name and the key value like this

Object obj = pm.getObjectById(MyClass.class, mykey);

If you are using your own PK class then the mykey value is the toString() form of the identity of your PK class.

45.2 JDO : PrimaryKey Classes
When you choose application identity you are defining which fields of the class are part of the primary key, and you are taking control of the specification of id's to DataNucleus. Application identity requires a primary key (PK) class, and each persistent capable class may define a different class for its primary key, and different persistent capable classes can use the same primary key class, as appropriate. If you have only a single primary-key field then there are built in PK classes so you can
forget this section. Where you have more than 1 primary key field, you would define the PK class like this

```xml
<class name="MyClass" identity-type="application" objectid-class="MyIdClass">
  ...
</class>
```
or using annotations

```java
@PersistenceCapable(objectIdClass=MyIdClass.class)
public class MyClass {
  ...
}
```

You now need to define the PK class to use. This is simplified for you because **if you have only one PK field then you dont need to define a PK class** and you only define it when you have a composite PK.

An important thing to note is that the PK can only be made up of fields of the following Java types

- Primitives: `boolean`, `byte`, `char`, `int`, `long`, `short`
- `java.lang`: `Boolean`, `Byte`, `Character`, `Integer`, `Long`, `Short`, `String`, `Enum`, `StringBuffer`
- `java.math`: `BigInteger`
- `java.sql`: `Date`, `Time`, `Timestamp`
- `java.util`: `Date`, `Currency`, `Locale`, `TimeZone`, `UUID`
- `java.net`: `URI`, `URL`
- `javax.jdo.spi`: `PersistenceCapable`

Note that the types in **bold** are JDO standard types. Any others are DataNucleus extensions and, as always, check the specific datastore docs to see what is supported for your datastore.

### 45.2.1 Single PrimaryKey field

The simplest way of using **application identity** is where you have a single PK field, and in this case you use **SingleFieldIdentity** mechanism. This provides a PrimaryKey and you don't need to specify the `objectid-class`. Let's take an example

```java
public class MyClass {
  long id;
  ...
}
```

```xml
<class name="MyClass" identity-type="application">
  <field name="id" primary-key="true"/>
  ...
</class>
```
or using annotations

```java
@PersistenceCapable
public class MyClass {

    @PrimaryKey
    long id;
    ...
}
```

So we didn't specify the JDO "objectid-class". You will, of course, have to give the field a value before persisting the object, either by setting it yourself, or by using a value-strategy on that field.

If you need to create an identity of this form for use in querying via `pm.getObjectById()` then you can create the identities in the following way.

For a "long" type:
```java
javax.jdo.identity.LongIdentity id = new javax.jdo.identity.LongIdentity(myClass, 101);
```

For a "String" type:
```java
javax.jdo.identity.StringIdentity id = new javax.jdo.identity.StringIdentity(myClass, "ABCD");
```

We have shown an example above for type "long", but you can also use this for the following:

- short, Short       - javax.jdo.identity.ShortIdentity
- int, Integer       - javax.jdo.identity.IntIdentity
- long, Long         - javax.jdo.identity.LongIdentity
- String             - javax.jdo.identity.StringIdentity
- char, Character    - javax.jdo.identity.CharIdentity
- byte, Byte         - javax.jdo.identity.ByteIdentity
- java.util.Date     - javax.jdo.identity.ObjectIdentity
- java.util.Currency - javax.jdo.identity.ObjectIdentity
- java.util.Locale   - javax.jdo.identity.ObjectIdentity

### 45.2.2 Multiple PrimaryKey field

Since there are many possible combinations of primary-key fields it is impossible for JDO to provide a series of built-in composite primary key classes. However the DataNucleus enhancer provides a mechanism for auto-generating a primary-key class for a persistable class. It follows the rules listed below and should work for all cases. Obviously if you want to tailor the output of things like the PK toString() method then you ought to define your own. The enhancer generation of primary-key class is only enabled if you don't define your own class.

### 45.2.3 Rules for User-Defined PrimaryKey classes

If you wish to use application identity and don't want to use the "SingleFieldIdentity" built-in PK classes then you must define a Primary Key class of your own. You can't use classes like
java.lang.String, or java.lang.Long directly. You must follow these rules when defining your primary key class.

- the Primary Key class must be public
- the Primary Key class must implement Serializable
- the Primary Key class must have a public no-arg constructor, which might be the default constructor
- the field types of all non-static fields in the Primary Key class must be serializable, and are recommended to be primitive, String, Date, or Number types
- all serializable non-static fields in the Primary Key class must be public
- the names of the non-static fields in the Primary Key class must include the names of the primary key fields in the JDO class, and the types of the common fields must be identical
- the equals() and hashCode() methods of the Primary Key class must use the value(s) of all the fields corresponding to the primary key fields in the JDO class
- if the Primary Key class is an inner class, it must be static
- the Primary Key class must override the toString() method defined in Object, and return a String that can be used as the parameter of a constructor
- the Primary Key class must provide a String constructor that returns an instance that compares equal to an instance that returned that String by the toString() method.
- the Primary Key class must be only used within a single inheritance tree.

Please note that if one of the fields that comprises the primary key is in itself a persistable object then you have **Compound Identity** and should consult the documentation for that feature which contains its own example.

### 45.2.4 PrimaryKey Example - Multiple Field

Here's an example of a composite (multiple field) primary key class
@PersistenceCapable(objectIdClass=ComposedIdKey.class)
public class MyClass {
    @PrimaryKey
    String field1;

    @PrimaryKey
    String field2;
    ...
}

public class ComposedIdKey implements Serializable {
    public String field1;
    public String field2;

    public ComposedIdKey () {
    }

    /**
     * Constructor accepting same input as generated by toString().
     */
    public ComposedIdKey(String value) {
        StringTokenizer token = new StringTokenizer (value, "::");
        token.nextToken();               // className
        this.field1 = token.nextToken(); // field1
        this.field2 = token.nextToken(); // field2
    }

    public boolean equals(Object obj) {
        if (obj == this) {
            return true;
        }
        if (!(obj instanceof ComposedIdKey)) {
            return false;
        }
        ComposedIdKey c = (ComposedIdKey)obj;
        return field1.equals(c.field1) && field2.equals(c.field2);
    }

    public int hashCode () {
        return this.field1.hashCode() ^ this.field2.hashCode();
    }

    public String toString () {
        // Give output expected by String constructor
        return this.getClass().getName() + "::" + this.field1 + "::" + this.field2;
    }
}

© 2015, DataNucleus • ALL RIGHTS RESERVED.
46 Nondurable Identity

46.1 JDO : Nondurable Identity

With nondurable identity your objects will not have a unique identity in the datastore. This type of identity is typically for log files, history files etc where you aren't going to access the object by key, but instead by a different parameter. In the datastore the table will typically not have a primary key. To specify that a class is to use nondurable identity with JDO you would add the following to the MetaData for the class.

```xml
<class name="MyClass" identity-type="nondurable">
  ...
</class>
```

or using annotations, for example

```java
@PersistenceCapable(identityType=IdentityType.NONDURABLE)
public class MyClass
{
  ...
}
```

DataNucleus provides support for "nondurable" identity for some datastores only currently (RDBMS, Excel, ODF, MongoDB, HBase). What this means for something like RDBMS is that the table of the class will not have a primary-key.
47 Compound Identity

47.1 JDO : Compound Identity Relationships

An identifying relationship (or "compound identity relationship" in JDO) is a relationship between two objects of two classes in which the child object must coexist with the parent object and where the primary key of the child includes the PersistenceCapable object of the parent. So effectively the key aspect of this type of relationship is that the primary key of one of the classes includes a PersistenceCapable field (hence why is is referred to as Compound Identity). This type of relation is available in the following forms

- 1-1 unidirectional
- 1-N collection bidirectional using ForeignKey
- 1-N map bidirectional using ForeignKey (key stored in value)

47.1.1 1-1 Relationship

Let's take the same classes as we have in the 1-1 Relationships. In the 1-1 relationships guide we note that in the datastore representation of the User and Account the ACCOUNT table has a primary key as well as a foreign-key to USER. In our example here we want to just have a primary key that is also a foreign-key to USER. To do this we need to modify the classes slightly and add primary-key fields and use "application-identity".

In addition we need to define primary key classes for our User and Account classes
public class User
{
    long id;

    ... (remainder of User class)

    /**
     * Inner class representing Primary Key
     */
    public static class PK implements Serializable
    {
        public long id;

        public PK()
        {
        }

        public PK(String s)
        {
            this.id = Long.valueOf(s).longValue();
        }

        public String toString()
        {
            return "" + id;
        }

        public int hashCode()
        {
            return (int)id;
        }

        public boolean equals(Object other)
        {
            if (other != null && (other instanceof PK))
            {
                PK otherPK = (PK)other;
                return otherPK.id == this.id;
            }
            return false;
        }
    }
}

public class Account
{
    User user;

    ... (remainder of Account class)

    /**
     * Inner class representing Primary Key
     */
    public static class PK implements Serializable
    {
        public User.PK user; // Use same name as the real field above

        public PK()
        {
        }

        public PK(String s)
        {
            StringTokenizer token = new StringTokenizer(s, "::");
            this.user = new User.PK(token.nextToken());
        }

        public String toString()
        {
            return this.user.toString();
        }

        public int hashCode()
        {
            return user.hashCode();
        }

        public boolean equals(Object other)
        {
            if (other != null && (other instanceof PK))
            {
                PK otherPK = (PK)other;
                return this.user.equals(otherPK.user);
            }
            return false;
        }
    }
}
To achieve what we want with the datastore schema we define the MetaData like this

```xml
<package name="mydomain">
  <class name="User" identity-type="application" objectid-class="User$PK">
    <field name="id" primary-key="true"/>
    <field name="login" persistence-modifier="persistent">
      <column length="20" jdbc-type="VARCHAR"/>
    </field>
  </class>
  
  <class name="Account" identity-type="application" objectid-class="Account$PK">
    <field name="user" persistence-modifier="persistent" primary-key="true">
      <column name="USER_ID"/>
    </field>
    <field name="firstName" persistence-modifier="persistent">
      <column length="50" jdbc-type="VARCHAR"/>
    </field>
    <field name="secondName" persistence-modifier="persistent">
      <column length="50" jdbc-type="VARCHAR"/>
    </field>
  </class>
</package>
```

So now we have the following datastore schema

![Datastore schema diagram](image)

Things to note:-

- You must use "application-identity" in both parent and child classes
- In the child Primary Key class, you must have a field with the same name as the relationship in the child class, and the field in the child Primary Key class must be the same type as the Primary Key class of the parent
- See also the [general instructions for Primary Key classes](#)
- You can only have one "Account" object linked to a particular "User" object since the FK to the "User" is now the primary key of "Account". To remove this restriction you could also add a "long id" to "Account" and make the "Account.PK" a composite primary-key

### 47.1.2 1-N Collection Relationship

Lets take the same classes as we have in the [1-N Relationships (FK)](#). In the 1-N relationships guide we note that in the datastore representation of the Account and Address classes the ADDRESS table has a primary key as well as a foreign-key to ACCOUNT. In our example here we want to have the
primary-key to ACCOUNT to include the foreign-key. To do this we need to modify the classes slightly, adding primary-key fields to both classes, and use "application-identity" for both.

In addition we need to define primary key classes for our Account and Address classes.
47 Compound Identity

public class Account
{
long id; // PK field
Set addresses = new HashSet();
... (remainder of Account class)
/**
* Inner class representing Primary Key
*/
public static class PK implements Serializable
{
public long id;
public PK()
{
}
public PK(String s)
{
this.id = Long.valueOf(s).longValue();
}
public String toString()
{
return "" + id;
}
public int hashCode()
{
return (int)id;
}
public boolean equals(Object other)
{
if (other != null && (other instanceof PK))
{
PK otherPK = (PK)other;
return otherPK.id == this.id;
}
return false;
}
}
}
public class Address
{
long id;
Account account;
.. (remainder of Address class)

©2015,

/**
* Inner class representing Primary Key
*/
public static class PK implements Serializable
DataNucleus • ALL RIGHTS RESERVED.
{
public long id; // Same name as real field above
public Account.PK account; // Same name as the real field above

185


To achieve what we want with the datastore schema we define the MetaData like this

```xml
<package name="mydomain">
  <class name="Account" identity-type="application" objectid-class="Account$PK">
    <field name="id" primary-key="true"/>
    <field name="firstName" persistence-modifier="persistent">
      <column length="50" jdbc-type="VARCHAR"/>
    </field>
    <field name="secondName" persistence-modifier="persistent">
      <column length="50" jdbc-type="VARCHAR"/>
    </field>
    <field name="addresses" persistence-modifier="persistent" mapped-by="account">
      <collection element-type="Address"/>
    </field>
  </class>
  <class name="Address" identity-type="application" objectid-class="Address$PK">
    <field name="id" primary-key="true"/>
    <field name="account" persistence-modifier="persistent" primary-key="true">
      <column name="ACCOUNT_ID"/>
    </field>
    <field name="city" persistence-modifier="persistent">
      <column length="50" jdbc-type="VARCHAR"/>
    </field>
    <field name="street" persistence-modifier="persistent">
      <column length="50" jdbc-type="VARCHAR"/>
    </field>
  </class>
</package>
```

So now we have the following datastore schema

```
ACCOUNT
+ACCOUNT_ID
FIRSTNAME
LASTNAME
```

```
ADDRESS
+ID
+ACCOUNT_ID
CITY
STREET
```

Things to note :-

- You must use "application-identity" in both parent and child classes
- In the child Primary Key class, you must have a field with the same name as the relationship in the child class, and the field in the child Primary Key class must be the same type as the Primary Key class of the parent
- See also the general instructions for Primary Key classes
- If we had omitted the "id" field from "Address" it would have only been possible to have one "Address" in the "Account" "addresses" collection due to PK constraints. For that reason we have the "id" field too.
47.1.3 1-N Map Relationship

Let's take the same classes as we have in the 1-N Relationships (FK). In this guide we note that in the datastore representation of the Account and Address classes the ADDRESS table has a primary key as well as a foreign-key to ACCOUNT. In our example here we want to have the primary-key to ACCOUNT to include the foreign-key. To do this we need to modify the classes slightly, adding primary-key fields to both classes, and use "application-identity" for both.

In addition we need to define primary key classes for our Account and Address classes
47 Compound Identity

public class Account
{
long id; // PK field
Set addresses = new HashSet();
... (remainder of Account class)
/**
* Inner class representing Primary Key
*/
public static class PK implements Serializable
{
public long id;
public PK()
{
}
public PK(String s)
{
this.id = Long.valueOf(s).longValue();
}
public String toString()
{
return "" + id;
}
public int hashCode()
{
return (int)id;
}
public boolean equals(Object other)
{
if (other != null && (other instanceof PK))
{
PK otherPK = (PK)other;
return otherPK.id == this.id;
}
return false;
}
}
}
public class Address
{
String alias;
Account account;
.. (remainder of Address class)

©2015,

/**
* Inner class representing Primary Key
*/
public static class PK implements Serializable
DataNucleus • ALL RIGHTS RESERVED.
{
public String alias; // Same name as real field above
public Account.PK account; // Same name as the real field above

188


To achieve what we want with the datastore schema we define the MetaData like this

```xml
<package name="com.mydomain">

<class name="Account" objectid-class="Account$PK">
    <field name="id" primary-key="true"/>
    <field name="firstname" persistence-modifier="persistent">
        <column length="100" jdbc-type="VARCHAR"/>
    </field>
    <field name="lastname" persistence-modifier="persistent">
        <column length="100" jdbc-type="VARCHAR"/>
    </field>
    <field name="addresses" persistence-modifier="persistent" mapped-by="account">
        <map key-type="java.lang.String" value-type="com.mydomain.Address"/>
        <key mapped-by="alias"/>
    </field>
</class>

<class name="Address" objectid-class="Address$PK">
    <field name="account" persistence-modifier="persistent" primary-key="true"/>
    <field name="alias" null-value="exception" primary-key="true">
        <column name="KEY" length="20" jdbc-type="VARCHAR"/>
    </field>
    <field name="city" persistence-modifier="persistent">
        <column length="50" jdbc-type="VARCHAR"/>
    </field>
    <field name="street" persistence-modifier="persistent">
        <column length="50" jdbc-type="VARCHAR"/>
    </field>
</class>
</package>
```

So now we have the following datastore schema

![Diagram of datastore schema]

Things to note:-

- You must use "application-identity" in both parent and child classes
- In the child Primary Key class, you must have a field with the same name as the relationship in the child class, and the field in the child Primary Key class must be the same type as the Primary Key class of the parent
- See also the [general instructions for Primary Key classes](#)
• If we had omitted the "alias" field from "Address" it would have only been possible to have one "Address" in the "Account" "addresses" collection due to PK constraints. For that reason we have the "alias" field too as part of the PK.
48 Versioning

48.1 JDO: Versioning of Objects

JDO allows objects of classes to be versioned. The version is typically used as a way of detecting if the object has been updated by another thread or PersistenceManager since retrieval using the current PersistenceManager - for use by Optimistic Transactions. JDO defines several "strategies" for generating the version of an object. The strategy has the following possible values

- **none**: stores a number like the version-number but will not perform any optimistic checks.
- **version-number**: stores a number (starting at 1) representing the version of the object.
- **date-time**: stores a Timestamp representing the time at which the object was last updated. Note that not all RDBMS store milliseconds in a Timestamp!
- **state-image**: stores a Long value being the hash code of all fields of the object. DataNucleus does not currently support this option.

48.1.1 Versioning using a surrogate column

JDO2s mechanism for versioning of objects in RDBMS datastores is via a surrogate column in the table of the class. In the MetaData you specify the details of the surrogate column and the strategy to be used. For example

```xml
<package name="mydomain">
  <class name="User" table="USER">
    <version strategy="version-number" column="VERSION"/>
    <field name="name" column="NAME"/>
    ...
  </class>
</package>
```

Alternatively using annotations

```java
@PersistenceCapable
@Version(strategy=VersionStrategy.VERSION_NUMBER, column="VERSION")
public class MyClass {
    ...
}
```

The specification above will create a table with an additional column called "VERSION" that will store the version of the object.

48.1.2 Versioning using a field of the class

DataNucleus provides a valuable extension to JDO whereby you can have a field of your class store the version of the object. This equates to JPA's versioning process whereby you have to have a field present. To do this let's take a class

```java
@Extension
```
public class User
{
    String name;
    ...
    long myVersion;
}

and we want to store the version of the object in the field "myVersion". So we specify the metadata as follows

```xml
<package name="mydomain">
    <class name="User" table="USER">
        <version strategy="version-number">
            <extension vendor-name="datanucleus" key="field-name" value="myVersion"/>
        </version>
        <field name="name" column="NAME"/>
        ...
        <field name="myVersion" column="VERSION"/>
    </class>
</package>
```

alternatively using annotations

```java
@PersistenceCapable
@Version(strategy=VersionStrategy.VERSION_NUMBER, column="VERSION",
    extensions={@Extension(vendorName="datanucleus", key="field-name", value="myVersion")})
public class MyClass
{
    protected long myVersion;
    ...
}
```

and so now objects of our class will have access to the version via the "myVersion" field.
49 Inheritance

49.1 JDO: Inheritance Strategies

In Java it is a normal situation to have inheritance between classes. With JDO you have choices to make as to how you want to persist your classes for the inheritance tree. For each class you select how you want to persist that classes information. You have the following choices.

1. The first and simplest to understand option is where each class has its own table in the datastore. In JDO this is referred to as new-table.
2. The second way is to select a class to have its fields persisted in the table of its subclass. In JDO this is referred to as subclass-table.
3. The third way is to select a class to have its fields persisted in the table of its superclass. In JDO this is known as superclass-table.
4. JDO3.1 introduces support for having all classes in an inheritance tree with their own table containing all fields. This is known as complete-table and is enabled by setting the inheritance strategy of the root class to use this.

In order to demonstrate the various inheritance strategies we need an example. Here are a few simple classes representing products in a (online) store. We have an abstract base class, extending this to to provide something that we can represent any product by. We then provide a few specialisations for typical products. We will use these classes later when defining how to persistent these objects in the different inheritance strategies.
JDO imposes a "default" inheritance strategy if none is specified for a class. If the class is a base class and no inheritance strategy is specified then it will be set to `new-table` for that class. If the class has a superclass and no inheritance strategy is specified then it will be set to `superclass-table`. This means that, when no strategy is set for the classes in an inheritance tree, they will default to using a single table managed by the base class.

You can control the "default" strategy chosen by way of a

![PMF Property](image)

This is specified by way of a PMF property `datanucleus.defaultInheritanceStrategy`. The default is `JDO2` which will give the above default behaviour for all classes that have no strategy specified. The other option is `TABLE_PER_CLASS` which will use "new-table" for all classes which have no strategy specified.

Please note that you must specify the identity of objects in the root persistable class of the inheritance hierarchy. You cannot redefine it down the inheritance tree.

See also:
- MetaData reference for `<inheritance>` element
49.1.1 Discriminator

Applicable to RDBMS, HBase, MongoDB

A **discriminator** is an extra "column" stored alongside data to identify the class of which that information is part. It is useful when storing objects which have inheritance to provide a quick way of determining the object type on retrieval. There are two types of discriminator supported by JDO

- **class-name**: where the actual name of the class is stored as the discriminator
- **value-map**: where a (typically numeric) value is stored for each class in question, allowing simple look-up of the class it equates to

You specify a discriminator as follows

```xml
<class name="Product">
    <inheritance>
        <discriminator strategy="class-name"/>
    </inheritance>
</class>
```

or with annotations

```java
@PersistenceCapable
@Discriminator(strategy=DiscriminatorStrategy.CLASS_NAME)
public class Product {...}
```

49.1.2 New Table

Applicable to RDBMS

Here we want to have a separate table for each class. This has the advantage of being the most normalised data definition. It also has the disadvantage of being slower in performance since multiple tables will need to be accessed to retrieve an object of a sub type. Let's try an example using the simplest to understand strategy **new-table**. We have the classes defined above, and we want to persist our classes each in their own table. We define the Meta-Data for our classes like this
```xml
<class name="AbstractProduct">
    <inheritance strategy="new-table"/>
    <field name="id" primary-key="true">
        <column name="PRODUCT_ID"/>
    </field>
    <field name="name">
        <column name="NAME"/>
    </field>
    <field name="description">
        <column name="DESCRIPTION"/>
    </field>
</class>

<class name="Product">
    <inheritance strategy="new-table"/>
    <field name="price">
        <column name="PRICE"/>
    </field>
</class>

<class name="Book">
    <inheritance strategy="new-table"/>
    <field name="isbn">
        <column name="ISBN"/>
    </field>
    <field name="author">
        <column name="AUTHOR"/>
    </field>
    <field name="title">
        <column name="TITLE"/>
    </field>
</class>

<class name="TravelGuide">
    <inheritance strategy="new-table"/>
    <field name="country">
        <column name="COUNTRY"/>
    </field>
</class>

<class name="CompactDisc">
    <inheritance strategy="new-table"/>
    <field name="artist">
        <column name="ARTIST"/>
    </field>
    <field name="title">
        <column name="TITLE"/>
    </field>
</class>
```

or with annotations
We use the `inheritance` element to define the persistence of the inherited classes.

In the datastore, each class in an inheritance tree is represented in its own datastore table (tables `ABSTRACTPRODUCT`, `PRODUCT`, `BOOK`, `TRAVELGUIDE`, and `COMPACTDISC`), with the subclasses tables’ having foreign keys between the primary key and the primary key of the superclass’ table.
In the above example, when we insert a TravelGuide object into the datastore, a row will be inserted into ABSTRACTPRODUCT, PRODUCT, BOOK, and TRAVELGUIDE.

49.1.3 Subclass table

DataNucleus supports persistence of classes in the tables of subclasses where this is required. This is typically used where you have an abstract base class and it doesn't make sense having a separate table for that class. In our example we have no real interest in having a separate table for the AbstractProduct class. So in this case we change one thing in the Meta-Data quoted above. We now change the definition of AbstractProduct as follows
This subtle change of use the `inheritance` element has the effect of using the `PRODUCT` table for both the `Product` and `AbstractProduct` classes, containing the fields of both classes.

In the above example, when we insert a TravelGuide object into the datastore, a row will be inserted into `PRODUCT`, `BOOK`, and `TRAVELGUIDE`.

**DataNucleus** doesn't currently support the use of classes defined with `subclass-table` strategy as having relationships where there are more than a single subclass that has a table. If the class has a single subclass with its own table then there should be no problem.
49.1.4 Superclass table

Applicable to RDBMS

DataNucleus supports persistence of classes in the tables of superclasses where this is required. This has the advantage that retrieval of an object is a single SQL call to a single table. It also has the disadvantage that the single table can have a very large number of columns, and database readability and performance can suffer, and additionally that a discriminator column is required. In our example, let's ignore the AbstractProduct class for a moment and assume that Product is the base class. We have no real interest in having separate tables for the Book and CompactDisc classes and want everything stored in a single table PRODUCT. We change our MetaData as follows:

```xml
<class name="Product">
  <inheritance strategy="new-table">
    <discriminator strategy="class-name">
      <column name="PRODUCT_TYPE"/>
    </discriminator>
  </inheritance>
  <field name="id" primary-key="true">
    <column name="PRODUCT_ID"/>
  </field>
  <field name="price">
    <column name="PRICE"/>
  </field>
</class>
<class name="Book">
  <inheritance strategy="superclass-table"/>
  <field name="isbn">
    <column name="ISBN"/>
  </field>
  <field name="author">
    <column name="AUTHOR"/>
  </field>
  <field name="title">
    <column name="TITLE"/>
  </field>
</class>
<class name="TravelGuide">
  <inheritance strategy="superclass-table"/>
  <field name="country">
    <column name="COUNTRY"/>
  </field>
</class>
<class name="CompactDisc">
  <inheritance strategy="superclass-table"/>
  <field name="artist">
    <column name="ARTIST"/>
  </field>
  <field name="title">
    <column name="DISCTITLE"/>
  </field>
</class>
```
This change of use of the **inheritance** element has the effect of using the PRODUCT table for all classes, containing the fields of **Product**, **Book**, **CompactDisc**, and **TravelGuide**. You will also note that we used a **discriminator** element for the **Product** class. The specification above will result in an extra column (called PRODUCT_TYPE) being added to the PRODUCT table, and containing the class name of the object stored. So for a Book it will have "com.mydomain.samples.store.Book" in that column. This column is used in discriminating which row in the database is of which type. The final thing to note is that in our classes **Book** and **CompactDisc** we have a field that is identically named. With **CompactDisc** we have defined that its column will be called DISCTITLE since both of these fields will be persisted into the same table and would have had identical names otherwise - this gets around the problem.

In the above example, when we insert a TravelGuide object into the datastore, a row will be inserted into the PRODUCT table only.
JDO 2 allows two types of discriminators. The example above used a discriminator strategy of *classname*. This inserts the class name into the discriminator column so that we know what the class of the object really is. The second option is to use a discriminator strategy of *value-map*. With this we will define a "value" to be stored in this column for each of our classes. The only thing here is that we have to define the "value" in the MetaData for ALL classes that use that strategy. So to give the equivalent example :-
As you can see from the MetaData DTD it is possible to specify the column details for the discriminator. DataNucleus supports this, but only currently supports the following values of jdbc-type: VARCHAR, CHAR, INTEGER, BIGINT, NUMERIC. The default column type will be a VARCHAR.
49.1.5 Complete table

Applicable to RDBMS, Neo4j, NeoDatis, Excel, OOXML, ODF, HBase, JSON, AmazonS3, GoogleStorage, MongoDB, LDAP

With "complete-table" we define the strategy on the root class of the inheritance tree and it applies to all subclasses. Each class is persisted into its own table, having columns for all fields (of the class in question plus all fields of superclasses). So taking the same classes as used above

```
<class name="Product">
  <inheritance strategy="complete-table"/>
  <field name="id" primary-key="true">
    <column name="PRODUCT_ID"/>
  </field>
  <field name="price">
    <column name="PRICE"/>
  </field>
</class>
<class name="Book">
  <field name="isbn">
    <column name="ISBN"/>
  </field>
  <field name="author">
    <column name="AUTHOR"/>
  </field>
  <field name="title">
    <column name="TITLE"/>
  </field>
</class>
<class name="TravelGuide">
  <field name="country">
    <column name="COUNTRY"/>
  </field>
</class>
<class name="CompactDisc">
  <field name="artist">
    <column name="ARTIST"/>
  </field>
  <field name="title">
    <column name="DISCTITLE"/>
  </field>
</class>
```

or with annotations

```java
@PersistenceCapable
@Inheritance(strategy=InheritanceStrategy.COMPLETE_TABLE)
public class AbstractProduct {...}
```

So the key thing is the specification of inheritance strategy at the root only. This then implies a datastore schema as follows
So any object of explicit type **Book** is persisted into the table "BOOK". Similarly any **TravelGuide** is persisted into the table "TRAVELGUIDE". In addition if any class in the inheritance tree is abstract then it won't have a table since there cannot be any instances of that type. **DataNucleus currently has limitations when using a class using this inheritance as the element of a collection.**

### 49.1.6 Retrieval of inherited objects

Applicable to all datastores

JDO provides particular mechanisms for retrieving inheritance trees. These are accessed via the Extent/Query interface. Taking our example above, we can then do

```java
tx.begin();
Extent e = pm.getExtent(com.mydomain.samples.store.Product.class, true);
Query q = pm.newQuery(e);
Collection c=(Collection)q.execute();
tx.commit();
```

The second parameter passed to `pm.getExtent` relates to whether to return subclasses. So if we pass in the root of the inheritance tree (Product in our case) we get all objects in this inheritance tree returned. You can, of course, use far more elaborate queries using JDOQL, but this is just to highlight the method of retrieval of subclasses.
50 Fields/Properties

50.1 JDO : Persistent Fields or Properties

Now that we have defined the class as persistable we need to define how to persist the different fields/properties that are to be persisted. Please note that JDO cannot persist static or final fields. There are two distinct modes of persistence definition. The most common uses fields, whereas an alternative uses properties.

50.1.1 Persistent Fields

The most common form of persistence is where you have a field in a class and want to persist it to the datastore. With this mode of operation DataNucleus will persist the values stored in the fields into the datastore, and will set the values of the fields when extracting it from the datastore.

Requirement : you have a field in the class. This can be public, protected, private or package access, but cannot be static or final.

An example of how to define the persistence of a field is shown below

```java
@PersistenceCapable
public class MyClass
{
    @Persistent
    Date birthday;

    @NotPersistent
    String someOtherField;
}
```

So, using annotations, we have marked the field `birthday` as persistent, whereas field `someOtherField` is not persisted. Please note that in this particular case, Date is by default persistent so we could omit the @Persistent annotation (with non-default-persistent types we would definitely need the annotation). Using XML MetaData we would have done

```xml
<class name="MyClass">
    <field name="birthday" persistence-modifier="persistent"/>
    <field name="someOtherField" persistence-modifier="none"/>
</class>
```

Please note that the field Java type defines whether it is, by default, persistable. Look at the Types Guide and if the type has a tick in the column "Persistent?" then you don’t need to mark the persistence-modifier as "persistent".

50.1.2 Persistent Properties

A second mode of operation is where you have Java Bean-style getter/setter for a property. In this situation you want to persist the output from getXXX to the datastore, and use the setXXX to load up the value into the object when extracting it from the datastore.
Requirement: you have a property in the class with Java Bean getter/setter methods. These methods can be public, protected, private or package access, but cannot be static. The class must have BOTH getter AND setter methods.

An example of how to define the persistence of a property is shown below

```java
@PersistenceCapable
public class MyClass {
    @Persistent
    Date getBirthday()
    {
        ...
    }
    void setBirthday(Date date)
    {
        ...
    }
}
```

So, using annotations, we have marked this class as persistent, and the getter is marked as persistent. By default a property is non-persistent, so we have no need in specifying the `someOtherField` as not persistent. Using XML MetaData we would have done

```xml
<class name="MyClass">
    <property name="birthday" persistence-modifier="persistent"/>
</class>
```

### 50.1.3 Overriding Superclass Field/Property MetaData

If you are using XML MetaData you can also override the MetaData for fields/properties of superclasses. You do this by adding an entry for `{class-name}.fieldName`, like this

```xml
<class name="Hotel" detachable="true">
    ...
    <field name="HotelSuperclass.someField" default-fetch-group="false"/>
</class>
```

so we have changed the field "someField" specified in the persistent superclass "HotelSuperclass" to not be part of the DFG.

### 50.1.4 Field/Property positioning

With some datastores (notably spreadsheets) it is desirable to be able to specify the relative position of a column. The default (for DataNucleus) is just to put them in ascending alphabetical order. JDO allows definition of this using the `position` attribute on a `column`. Here's an example, using XML metadata
and with Annotations

```java
@PersistenceCapable(table="People")
public class Person {
    @Column(position=0)
    long personNum;

    @Column(position=1)
    String firstName;

    @Column(position=2)
    String lastName;
}
```
51 Java Types

51.1 JDO : Persistable Field Types
When persisting a class, a persistence solution needs to know how to persist the types of each field in the class. Clearly a persistence solution can only support a finite number of Java types; it cannot know how to persist every possible type creatable. The JDO specification define lists of types that are required to be supported by all implementations of those specifications. This support can be conveniently split into two parts

51.1.1 First-Class (FCO) Types
An object that can be referred to (object reference, providing a relation) and that has an "identity" is termed a First Class Object (FCO). DataNucleus supports the following Java types as FCO

- PersistenceCapable: any class marked for persistence can be persisted with its own identity in the datastore
- interface where the field represents a PersistenceCapable object
- java.lang.Object where the field represents a PersistenceCapable object

51.1.2 Supported Second-Class (SCO) Types
An object that does not have an "identity" is termed a Second Class Object (SCO). This is something like a String or Date field in a class, or alternatively a Collection (that contains other objects). The table below shows the currently supported SCO java types in DataNucleus. The table shows

- Extension?: whether the type is JDO2 standard, or is a DataNucleus extension
- default-fetch-group (DFG): whether the field is retrieved by default when retrieving the object itself
- persistence-modifier: whether the field is persisted by default, or whether the user has to mark the field as persistent in XML/annotations to persist it
- proxied: whether the field is represented by a "proxy" that intercepts any operations to detect whether it has changed internally.
- primary-key: whether the field can be used as part of the primary-key

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>datanucleus-core</td>
</tr>
<tr>
<td>byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>datanucleus-core</td>
</tr>
<tr>
<td>char</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>datanucleus-core</td>
</tr>
<tr>
<td>double</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>datanucleus-core</td>
</tr>
<tr>
<td>float</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>datanucleus-core</td>
</tr>
<tr>
<td>int</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>datanucleus-core</td>
</tr>
<tr>
<td>long</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>datanucleus-core</td>
</tr>
</tbody>
</table>

©2015, DataNucleus • ALL RIGHTS RESERVED.
<table>
<thead>
<tr>
<th>Type</th>
<th>boolean[]</th>
<th>byte[]</th>
<th>char[]</th>
<th>double[]</th>
<th>float[]</th>
<th>int[]</th>
<th>long[]</th>
<th>short[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>boolean[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>byte[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>char[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>double[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>float[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>long[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>short[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Boolean</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Character</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Double</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Float</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Integer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Long</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Short</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Boolean[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Byte[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Character[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Double[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Float[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Integer[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class Type</td>
<td>Extension</td>
<td>Sync</td>
<td>Async</td>
<td>datanucleus-core</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------</td>
<td>--------</td>
<td>--------</td>
<td>------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Long[</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Short[</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Numb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Object</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.String</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.String[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Class</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.math.BigDecimal[</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.math.BigInteger</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.math.BigInteger[</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.sql.Date</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.sql.Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.sql.Timestamp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.ArrayList</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.BitSet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Calendar</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Collection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Currency</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Date</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Date[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.GregorianCalendar</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.GregorianCalendar[7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>Extension</td>
<td>datanucleus-core</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-----------</td>
<td>------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.HashMap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.HashMap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.HashSet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Hashtable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.LinkedHashMap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.LinkedHashSet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.List</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util/locale</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Map</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Properties</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.PriorityQueue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Queue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Set</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.SortedSet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.SortedSet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Stack</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.TimeZone</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.TreeMap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.TreeSet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.UUID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Vector</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.awt.Color</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
java.awt.image
java.awt.Point
java.awt.Rectangle
java.net.URI
java.net.URL
java.io.Serializable
java.io.File
javax.jdo.spi.PersistenceCapable
javax.jdo.spi.PersistenceCapable[]
java.lang.Enum
java.lang.Enum[]
javax.time.calendar.LocalDateTime
javax.time.calendar.LocalTime
javax.time.calendar.LocalDate
org.joda.time.DateTime
org.joda.time.LocalTime
org.joda.time.LocalDate
org.joda.time.Duration
org.joda.time.Interval
org.joda.time.Period
com.google.common.collect.Multiset

[1] - java.util.SortedSet, java.util.TreeSet allow the specification of comparators via the "comparator-name" DataNucleus extension MetaData element (within <collection>). The headSet, tailSet, subSet methods are only supported when using cached collections.
5.1 Java Types

- [2] - `java.util.SortedMap, java.util.TreeMap` allow the specification of comparators via the "comparator-name" DataNucleus extension MetaData element (within `<map>`). The headMap, tailMap, subMap methods are only supported when using cached containers.

- [3] - `java.lang.StringBuffer` dirty check mechanism is limited to immutable mode, it means, if you change a StringBuffer object field, you must reassign it to the owner object field to make sure changes are propagated to the database.

- [4] - `java.lang.Number` will be stored in a column capable of storing a BigDecimal, and will store to the precision of the object to be persisted. On reading back the object will be returned typically as a BigDecimal since there is no mechanism for determining the type of the object that was stored.

- [5] - `java.util.LinkedHashMap` treated as a Map currently. No List-ordering is supported.


- [7] - `java.util.Calendar` can be stored into two columns (millsecs, Timezone) or into a single column (Timestamp). The single column option is not guaranteed to preserve the TimeZone of the input Calendar.

- [8] - available only for RDBMS, persisted into LONGVARBINARY, and retrieved as streamable so as not to adversely affect memory utilisation, hence suitable for large files. New in 3.2.8 of datanucleus-rdbms

Note that support is available for persisting other types depending on the datastore to which you are persisting

  - RDBMS GeoSpatial types via the DataNucleus RDBMS Spatial plugin

If you have support for any additional types and would either like to contribute them, or have them listed here, let us know

You can add support for other basic Java types quite easily, particularly if you can store it as a String or Long and then retrieve it back into its object form from that - See the Java Types plugin-point You can also define more specific support for it with RDBMS datastores - See the RDBMS Java Types plugin-point

Handling of second-class types uses wrappers and bytecode enhancement with DataNucleus. This contrasts to what Hibernate uses (proxies), and what Hibernate imposes on you. See this blog entry if you have doubts about this approach.

51.1.3 Enums

By default an Enum is persisted as either a String form (the name), or as an integer form (the ordinal). You control which form by specifying the column `jdbc-type`.

An extension to this is where you have an Enum that defines its own "value"s for the different enum options.
public enum MyColour {
    RED((short)1), GREEN((short)3), BLUE((short)5), YELLOW((short)8);

    private short value;

    private MyColour(short value) {
        this.value = value;
    }

    public short getValue() {
        return value;
    }

    public static MyColour getEnumByValue(short value) {
        switch (value) {
            case 1:
                return RED;
            case 3:
                return GREEN;
            case 5:
                return BLUE;
            default:
                return YELLOW;
        }
    }
}

With the default persistence it would persist as String-based, so persisting "RED" "GREEN" "BLUE" etc. With *jdbc-type* as INTEGER it would persist 0, 1, 2, 3 being the ordinal values. If you define the metadata as

```java
@Extensions({
    @Extension(vendorName="datanucleus", key="enum-getter-by-value", value="getEnumByValue"),
    @Extension(vendorName="datanucleus", key="enum-value-getter", value="getValue")
})
MyColour colour;
```

this will now persist 1, 3, 5, 8, being the "value" of each of the enum options.

51.1.4 TypeConverters

By default DataNucleus will store the value using its own internal configuration/default for the java type and for the datastore. The user can, however, change that by making use of a *TypeConverter*. 
You firstly need to define the `TypeConverter` class (assuming you aren’t going to use an internal DataNucleus converter, and for this you should refer to the `TypeConverter plugin-point`). Once you have the converter defined, and registered in a `plugin.xml` under a name you then mark the field/property to use it

```java
@Extension(vendorName="datanucleus", key="type-converter-name", value="kryo-serialise")
String longString;
```

In this case we have a String field but we want to serialise it, not using normal Java serialisation but using the "Kryo" library. When it is stored it will be converted into a serialised form and when read back in will be deserialised. You can see the example Kryo TypeConverter over on GitHub.

### 51.1.5 Eclipse EMF models

You could try to persist Eclipse EMF models using the Texo project to generate POJOs
52 Value Generation

52.1 JDO : Value generation

Fields of a class can either have the values set by you the user, or you can set DataNucleus to generate them for you. This is of particular importance with identity fields where you want unique identities. You can use this value generation process with any field in JDO. There are many different "strategies" for generating values, as defined by the JDO specifications, and also some DataNucleus extensions. Some strategies are specific to a particular datastore, and some are generic. You should choose the strategy that best suits your target datastore. The available strategies for JDO are :-

- **native** - this is the default and allows DataNucleus to choose the most suitable for the datastore.
- **sequence** - this uses a datastore sequence (if supported by the datastore)
- **identity** - these use autoincrement/identity/serial features in the datastore (if supported by the datastore)
- **increment** - this is datastore neutral and increments a sequence value using a table.
- **uuid-string** - this is a UUID in string form
- **uuid-hex** - this is a UUID in hexadecimal form
- **uuid** - provides a pure UUID utilising the JDK1.5 UUID class
- **auid** - provides a pure UUID following the OpenGroup standard
- **timestamp** - creates a java.sql.Timestamp of the current time
- **timestamp-value** - creates a long (millisecs) of the current time
- **max** - uses a max(column)+1 method (only in RDBMS)
- **datastore-uuid-hex** - UUID in hexadecimal form using datastore capabilities (only in RDBMS)
- **user-supplied value generators** - allows you to hook in your own identity generator

See also :-

- JDO MetaData reference for <class>
- JDO MetaData reference for <datastore-identity>
- JDO MetaData reference for <field>
- JDO Annotation reference for @DatastoreIdentity
- JDO Annotation reference for @Persistent
Please note that by defining a value-strategy for a field then it will, by default, always generate a value for that field on persist. If the field can store nulls and you only want it to generate the value at persist when it is null (i.e. you haven’t assigned a value yourself) then you can add the extension "strategy-when-notnull" as false.

52.1.1 native
With this strategy DataNucleus will choose the most appropriate strategy for the datastore being used. If you define the field as String-based then it will choose uuid-hex. Otherwise the field is numeric in which case it chooses identity if supported, otherwise sequence if supported, otherwise increment if supported otherwise throws an exception. On RDBMS you can get the behaviour used up until DN v3.0 by specifying the persistence property datanucleus.rdbms.useLegacyNativeValueStrategy as true.

52.1.2 sequence
A sequence is a user-defined database function that generates a sequence of unique numeric ids. The unique identifier value returned from the database is translated to a java type: java.lang.Long. DataNucleus supports sequences for the following datastores:

- Oracle
- PostgreSQL
- SAP DB
- DB2
- Firebird
- HSQLDB
- H2
- Derby (from v10.6)
- SQLServer (from v2012)
- NuoDB

To configure a class to use either of these generation methods with datastore identity you simply add this to the class’ Meta-Data

```xml
<sequence name="yourseq" datastore-sequence="YOUR_SEQUENCE_NAME" strategy="noncontiguous"/>
<class name="myclass" ... >
  <datastore-identity strategy="sequence" sequence="yourseq"/>
  ...
</class>
```

or using annotations

```java
@PersistenceCapable
@DatastoreIdentity(strategy="sequence", sequence="yourseq")
@Sequence(name="yourseq", datastore-sequence="YOUR_SEQUENCE_NAME", strategy=NONCONTIGUOUS)
public class MyClass
```
You replace "YOUR_SEQUENCE_NAME" with your sequence name. To configure a class to use either of these generation methods using application identity you would add the following to the class’ Meta-Data

```xml
<sequence name="yourseq" datastore-sequence="YOUR_SEQUENCE_NAME" strategy="noncontiguous"/>
<class name="myclass" ... >
  <field name="myfield" primary-key="true" value-strategy="sequence" sequence="yourseq"/>
  ...
</class>
```

or using annotations

```java
@PersistenceCapable
@Sequence(name="yourseq", datastore-sequence="YOUR_SEQUENCE_NAME", strategy=NONCONTIGUOUS)
public class MyClass
{
    @Persistent(valueStrategy="sequence", sequence="yourseq")
    private long myfield;
    ...
}
```

If the sequence does not yet exist in the database at the time DataNucleus needs a new unique identifier, a new sequence is created in the database based on the JDO Meta-Data configuration. Additional properties for configuring sequences are set in the JDO Meta-Data, see the available properties below. Unsupported properties by a database are silently ignored by DataNucleus.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>key-initial-value</td>
<td>the initial value for the sequence. In JDO3.1 this is specified in the standard metadata (initialValue)</td>
<td>No</td>
</tr>
</tbody>
</table>
| key-cache-size    | number of unique identifiers to cache in the PersistenceManagerFactory instance. Notes:  
1. The keys are pre-allocated, cached and used on demand. 
If key-cache-size is greater than 1, it may generate holes in the object keys in the database, if not all keys are used.  
In JDO3.1 this is specified in the standard metadata (allocationSize) | No       |
| key-min-value     | determines the minimum value a sequence can generate                        | No       |
| key-max-value     | determines the maximum value a sequence can generate                        | No       |
This value generator will generate values unique across different JVMs

### 52.1.3 identity

Auto-increment/identity/serial are primary key columns that are populated when a row is inserted in the table. These use the databases own keywords on table creation and so rely on having the table structure either created by DataNucleus or having the column with the necessary keyword.

DataNucleus supports auto-increment/identity/serial keys for many databases including:

- DB2 (IDENTITY)
- MySQL (AUTOINCREMENT)
- MSSQL (IDENTITY)
- Sybase (IDENTITY)
- HSQLDB (IDENTITY)
- H2 (IDENTITY)
- PostgreSQL (SERIAL)
- Derby (IDENTITY)
- MongoDB - String based
- Neo4j - long based
- NuoDB (IDENTITY)

This generation strategy should only be used if there is a single "root" table for the inheritance tree. If you have more than 1 root table (e.g using subclass-table inheritance) then you should choose a different generation strategy.

For a class using datastore identity you need to set the `strategy` attribute. You can configure the Meta-Data for the class something like this (replacing 'myclass' with your class name):

```xml
<class name="myclass">
    <datastore-identity strategy="identity"/>
    ...
</class>
```

For a class using application identity you need to set the `value-strategy` attribute on the primary key field. You can configure the Meta-Data for the class something like this (replacing 'myclass' and 'myfield' with your class and field names):

```xml
...
Please be aware that if you have an inheritance tree with the base class defined as using "identity" then the column definition for the PK of the base table will be defined as "AUTO_INCREMENT" or "IDENTITY" or "SERIAL" (dependent on the RDBMS) and all subtables will NOT have this identifier added to their PK column definitions. This is because the identities are assigned in the base table (since all objects will have an entry in the base table).

Please note that if using optimistic transactions, this strategy will mean that the value is only set when the object is actually persisted (i.e at flush() or commit())

This value generator will generate values unique across different JVMs

52.1.4 increment

This method is database neutral and uses a sequence table that holds an incrementing sequence value. The unique identifier value returned from the database is translated to a Java type: java.lang.Long. This strategy will work with any datastore. This method require a sequence table in the database and creates one if doesn't exist.

To configure a datastore identity class to use this generation method you simply add this to the classes Meta-Data.

```
<class name="myclass" ... >
  <datastore-identity strategy="increment"/>
  ...
</class>
```

To configure an application identity class to use this generation method you simply add this to the class’ Meta-Data. If your class is in an inheritance tree you should define this for the base class only.

```
<class name="myclass" ... >
  <field name="myfield" primary-key="true" value-strategy="increment"/>
  ...
</class>>
```

Additional properties for configuring this generator are set in the JDO Meta-Data, see the available properties below. Unsupported properties are silently ignored by DataNucleus.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>key-initial-value</td>
<td>First value to be allocated.</td>
<td>No. Defaults to 1</td>
</tr>
</tbody>
</table>
This value generator will generate values unique across different JVMs

### 52.1.5 uuid-string

This generator creates identities with 16 characters in string format. The identity contains the IP address of the local machine where DataNucleus is running, as well as other necessary components to provide uniqueness across time.

This generator can be used in concurrent applications. It is especially useful in situations where large numbers of transactions within a certain amount of time have to be made, and the additional overhead of synchronizing the concurrent creation of unique identifiers through the database would
break performance limits. It doesn't require datastore access to generate the identities and so has performance benefits over some of the other generators.

For a class using datastore identity you need to add metadata something like the following

```
<class name="myclass" ... >
    <datastore-identity strategy="uuid-string"/>
    ...
</class>
```

To configure an application identity class to use this generation method you simply add this to the class' JDO Meta-Data.

```
<class name="myclass" ... >
    <field name="myfield" primary-key="true" value-strategy="uuid-string"/>
    ...
</class>
```

### 52.1.6 uuid-hex

This generator creates identities with 32 characters in hexadecimal format. The identity contains the IP address of the local machine where DataNucleus is running, as well as other necessary components to provide uniqueness across time.

This generator can be used in concurrent applications. It is especially useful in situations where large numbers of transactions within a certain amount of time have to be made, and the additional overhead of synchronizing the concurrent creation of unique identifiers through the database would break performance limits. It doesn't require datastore access to generate the identities and so has performance benefits over some of the other generators.

For a class using datastore identity you need to add metadata something like the following

```
<class name="myclass" ... >
    <datastore-identity strategy="uuid-hex"/>
    ...
</class>
```

To configure an application identity class to use this generation method you simply add this to the class' JDO Meta-Data.

```
<class name="myclass" ... >
    <field name="myfield" primary-key="true" value-strategy="uuid-hex"/>
    ...
</class>
```

### 52.1.7 datastore-uuid-hex

©2015, DataNucleus • ALL RIGHTS RESERVED.
This method is like the "uuid-hex" option above except that it utilises datastore capabilities to generate the UUIDHEX code. Consequently this only works on some RDBMS (MSSQL, MySQL). The disadvantage of this strategy is that it makes a call to the datastore for each new UUID required. The generated UUID is in the same form as the AUID strategy where identities are generated in memory and so the AUID strategy is the recommended choice relative to this option.

For a class using datastore identity you need to add metadata something like the following:

```xml
<class name="myclass" ... >
  <datastore-identity strategy="datastore-uuid-hex"/>
  ...
</class>
```

To configure an application identity class to use this generation method you simply add this to the class' JDO Meta-Data.

```xml
<class name="myclass" ... >
  <field name="myfield" primary-key="true" value-strategy="datastore-uuid-hex"/>
  ...
</class>
```

### 52.1.8 max

This method is database neutral and uses the "select max(column) from table" + 1 strategy to create unique ids. The unique identifier value returned from the database is translated to a java type: java.lang.Long. It is however not recommended by DataNucleus since it makes a DB call for every record to be inserted and hence is inefficient. Each DB call will run a scan in all table contents causing contention and locks in the table. We recommend the use of either Sequence or Identity based value generators (see below) - which you use would depend on your RDBMS.

For a class using datastore identity you need to add metadata something like the following:

```xml
<class name="myclass" ... >
  <datastore-identity strategy="max"/>
  ...
</class>
```

To configure an application identity class to use this generation method you simply add this to the class' JDO Meta-Data.

```xml
<class name="myclass" ... >
  <field name="myfield" primary-key="true" value-strategy="max"/>
  ...
</class>
```

This value generator will NOT guarantee to generate values unique across different JVMs. This is because it will select the "max+1" and before creating the record another thread may come in and insert one.
52.1.9 uuid

This generator uses the JDK1.5 UUID class to generate values. The values are 128-bit (36 character) of the form "0e400c2c-b3a0-4786-a0c6-f2607bf643eb"

This generator can be used in concurrent applications. It is especially useful in situations where large numbers of transactions within a certain amount of time have to be made, and the additional overhead of synchronizing the concurrent creation of unique identifiers through the database would break performance limits.

For a class using datastore identity you need to add metadata something like the following:

```xml
<class name="myclass" ... >
  <datastore-identity strategy="uuid" />
  ...
</class>
```

To configure an application identity class to use this generation method you simply add this to the class' JDO Meta-Data:

```xml
<class name="myclass" ... >
  <field name="myfield" primary-key="true" value-strategy="uuid" />
  ...
</class>
```

Or using annotations

```java
public class MyClass
{
  @Persistent(customValueStrategy="uuid")
  String myField;
}
```

This value generator will generate values unique across different JVMs

52.1.10 auid

This generator uses a Java implementation of DCE UUIDs to create unique identifiers without the overhead of additional database transactions or even an open database connection. The identifiers are Strings of the form "LLLLLLL-MMMM-HHHH-CCCC-NNNNNNNNNNNN" where 'L', 'M', 'H', 'C' and 'N' are the DCE UUID fields named time low, time mid, time high, clock sequence and node.

This generator can be used in concurrent applications. It is especially useful in situations where large numbers of transactions within a certain amount of time have to be made, and the additional overhead of synchronizing the concurrent creation of unique identifiers through the database would break performance limits.
For a class using datastore identity you need to add metadata something like the following:

```xml
<class name="myclass" ... >
  <datastore-identity strategy="auid"/>
  ...
</class>
```

To configure an application identity class to use this generation method you simply add this to the class' JDO Meta-Data.

```xml
<class name="myclass" ... >
  <field name="myfield" primary-key="true" value-strategy="auid"/>
  ...
</class>
```

This value generator will generate values unique across different JVMs.

### 52.1.11 timestamp

This method will create a java.sql.Timestamp of the current time (at insertion in the datastore).

For a class using datastore identity you need to add metadata something like the following:

```xml
<class name="myclass" ... >
  <datastore-identity strategy="timestamp"/>
  ...
</class>
```

To configure an application identity class to use this generation method you simply add this to the class' JDO Meta-Data.

```xml
<class name="myclass" ... >
  <field name="myfield" primary-key="true" value-strategy="timestamp"/>
  ...
</class>
```

### 52.1.12 timestamp-value

This method will create a long of the current time in millisecs (at insertion in the datastore).

For a class using datastore identity you need to add metadata something like the following:

```xml
<class name="myclass" ... >
  <datastore-identity strategy="timestamp-value"/>
  ...
</class>
```

```xml
<class name="myclass" ... >
  <field name="myfield" primary-key="true" value-strategy="timestamp-value"/>
  ...
</class>
```
To configure an application identity class to use this generation method you simply add this to the class' JDO Meta-Data.

```xml
<class name="myclass" ... >
  <datastore-identity strategy="timestamp-value"/>
  ...
</class>
```

### 52.2 Standalone ID generation

This section describes how to use the DataNucleus Value Generator API for generating unique keys for objects outside the DataNucleus (JDO) runtime. DataNucleus defines a framework for identity generation and provides many built-in strategies for identities. You can make use of the same strategies described above but for generating identities manually for your own use. The process is described below.

The DataNucleus Value Generator API revolves around 2 classes. The entry point for retrieving generators is the `ValueGenerationManager`. This manages the appropriate `ValueGenerator` classes. Value generators maintain a block of cached ids in memory which avoid reading the database each time it needs a new unique id. Caching a block of unique ids provides you the best performance but can cause "holes" in the sequence of ids for the stored objects in the database.

Let's take an example. Here we want to obtain an identity using the `TableGenerator` ("increment" above). This stores identities in a datastore table. We want to generate an identity using this. Here is what we add to our code.

```xml
<class name="myclass" ... >
  <field name="myfield" primary-key="true" value-strategy="timestamp-value"/>
  ...
</class>
```
PersistenceManagerImpl pm = (PersistenceManagerImpl) ... // cast your pm to impl;

// Obtain a ValueGenerationManager
ValueGenerationManager mgr = new ValueGenerationManager();

// Obtain a ValueGenerator of the required type
Properties properties = new Properties();
properties.setProperty("sequence-name", "GLOBAL"); // Use a global sequence number (for all tables)
ValueGenerator generator = mgr.createValueGenerator("MyGenerator",
    org.datanucleus.store.rdbms.valuegenerator.TableGenerator.class, props, pm.getStoreManager(),
    new ValueGenerationConnectionProvider()
    {
        RDBMSManager rdbmsManager = null;
        ManagedConnection con;
        public ManagedConnection retrieveConnection()
        {
            rdbmsManager = (RDBMSManager) pm.getStoreManager();
            try
            {
                // important to use TRANSACTION_NONE like DataNucleus does
                con = rdbmsManager.getConnection(Connection.TRANSACTION_NONE);
                return con;
            }
            catch (SQLException e)
            {
                logger.error("Failed to obtain new DB connection for identity generation!");
                throw new RuntimeException(e);
            }
        }
        public void releaseConnection()
        {
            try
            {
                con.close();
                con = null;
            }
            catch (DataNucleusException e)
            {
                logger.error("Failed to close DB connection for identity generation!");
                throw new RuntimeException(e);
            }
            finally
            {
                rdbmsManager = null;
            }
        }
    });

// Retrieve the next identity using this strategy
Long identifier = (Long)generator.next();
Some ValueGenerators are specific to RDBMS datastores, and some are generic, so bear this in mind when selecting and adding your own.
53 Sequences

53.1 JDO : Datastore Sequences

Particularly when specifying the identity of an object, sequences are a very useful facility. DataNucleus supports the automatic assignment of sequence values for object identities. However such sequences may also have use when a user wishes to assign such identity values themselves, or for other roles within an application. JDO 2 defines an interface for sequences for use in an application - known as Sequence.

There are 2 forms of "sequence" available through this interface - the ones that DataNucleus provides utilising datastore capabilities, and ones that a user provides using something known as a "factory class".

53.1.1 DataNucleus Sequences

DataNucleus internally provides 2 forms of sequences. When the underlying datastore supports native sequences, then these can be leveraged through this interface. Alternatively, where the underlying datastore doesn't support native sequences, then a table-based incrementing sequence can be used. The first thing to do is to specify the Sequence in the Meta-Data for the package requiring the sequence. This is done as follows

```xml
<jdo>
    <package name="MyPackage">
        <class name="MyClass">
            ...
        </class>
        <sequence name="ProductSequence" datastore-sequence="PRODUCT_SEQ" strategy="contiguous"/>
        <sequence name="ProductSequenceNontrans" datastore-sequence="PRODUCT_SEQ_NONTRANS" strategy="nontransactional"/>
    </package>
</jdo>
```

So we have defined two Sequences for the package MyPackage. Each sequence has a symbolic name that is referred to within JDO (within DataNucleus), and it has a name in the datastore. The final attribute represents whether the sequence is transactional or not.

All we need to do now is to access the Sequence in our persistence code in our application. This is done as follows

```java
PersistenceManager pm = pmf.getPersistenceManager();
Sequence seq = pm.getSequence("MyPackage.ProductSequence");

long value = seq.nextValue();
```

Please be aware that when you have a Sequence declared with a strategy of "contiguous" this means "transactional contiguous" and that you need to have a Transaction open when you access it.
JDO3.1 allows control over the allocation size (default=50) and initial value (default=1) for the sequence. So we can do

```xml
<sequence name="ProductSequence" datastore-sequence="PRODUCT_SEQ" strategy="contiguous"
  allocation-size="10"/>
```

which will allocate 10 new sequence values each time the allocated sequence values is exhausted.

### 53.1.2 Factory Class Sequences

It is equally possible to provide your own Sequence capability using a factory class. This is a class that creates an implementation of the JDO Sequence. Let's give an example of what you need to provide. Firstly you need an implementation of the JDO Sequence interface, so we define ours like this
public class SimpleSequence implements Sequence
{
    String name;
    long current = 0;

    public SimpleSequence(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }

    public Object next()
    {
        current++;
        return new Long(current);
    }

    public long nextValue()
    {
        current++;
        return current;
    }

    public void allocate(int arg0)
    {
    }

    public Object current()
    {
        return new Long(current);
    }

    public long currentValue()
    {
        return current;
    }
}

So our sequence simply increments by 1 each call to next(). The next thing we need to do is provide a factory class that creates this Sequence. This factory needs to have a static newInstance method that returns the Sequence object. We define our factory like this
package org.datanucleus.samples.sequence;

import javax.jdo.datastore.Sequence;

public class SimpleSequenceFactory
{
    public static Sequence newInstance()
    {
        return new SimpleSequence("MySequence");
    }
}

and now we define our MetaData like this

<jdo>
    <package name="MyPackage">
        <class name="MyClass">
            ...
        </class>

        <sequence name="ProductSequenceFactory" strategy="nontransactional"
            factory-class="org.datanucleus.samples.sequence.SimpleSequenceFactory"/>
    </package>
</jdo>

So now we can call

PersistenceManager pm = pmf.getPersistenceManager();

Sequence seq = pm.getSequence("MyPackage.ProductSequenceFactory");
54 Embedded Fields

54.1 JDO : Embedded Fields

The JDO persistence strategy typically involves persisting the fields of any class into its own table, and representing any relationships from the fields of that class across to other tables. There are occasions when this is undesirable, maybe due to an existing datastore schema, or because a more convenient datastore model is required. JDO allows the persistence of fields as **embedded** typically into the same table as the "owning" class.

One important decision when defining objects of a type to be embedded into another type is whether objects of that type will ever be persisted in their own right into their own table, and have an identity. JDO provides a MetaData attribute that you can use to signal this.

```xml
<jdo>
  <package name="com.mydomain.samples.embedded">
    <class name="MyClass" embedded-only="true">
      ...
    </class>
  </package>
</jdo>
```

With the above MetaData (using the **embedded-only** attribute), in our application any objects of the class **MyClass** cannot be persisted in their own right. They can only be embedded into other objects.

JDO's definition of embedding encompasses several types of fields. These are described below

- **Embedded PersistenceCapable objects** - where you have a 1-1 relationship and you want to embed the other PersistenceCapable into the same table as your object.
- **Embedded Nested PersistenceCapable objects** - like the first example except that the other object also has another PersistenceCapable object that also should be embedded
- **Embedded Collection elements** - where you want to embed the elements of a collection into a join table (instead of persisting them into their own table)
- **Embedded Map keys/values** - where you want to embed the keys/values of a map into a join table (instead of persisting them into their own table)

54.1.1 Embedding PersistenceCapable objects

Applicable to RDBMS, Excel, OOXML, ODF, HBase, MongoDB, Neo4j.

In a typical 1-1 relationship between 2 classes, the 2 classes in the relationship are persisted to their own table, and a foreign key is managed between them. With JDO and DataNucleus you can persist the related PersistenceCapable object as embedded into the same table. This results in a single table in the datastore rather than one for each of the 2 classes.

Let's take an example. We are modelling a **Computer**, and in our simple model our **Computer** has a graphics card and a sound card. So we model these cards using a **ComputerCard** class. So our classes become
public class Computer
{
    private String operatingSystem;

    private ComputerCard graphicsCard;

    private ComputerCard soundCard;

    public Computer(String osName,
                     ComputerCard graphics,
                     ComputerCard sound)
    {
        this.operatingSystem = osName;
        this.graphicsCard = graphics;
        this.soundCard = sound;
    }

    ...
}

class ComputerCard
{
    public static final int ISA_CARD = 0;
    public static final int PCI_CARD = 1;
    public static final int AGP_CARD = 2;

    private String manufacturer;

    private int type;

    public ComputerCard(String manufacturer,
                         int type)
    {
        this.manufacturer = manufacturer;
        this.type = type;
    }

    ...
}

The traditional (default) way of persisting these classes would be to have a table to represent each class. So our datastore will look like this
However we decide that we want to persist `Computer` objects into a table called `COMPUTER` and we also want to persist the PC cards into the same table. We define our MetaData like this:

```xml
<package name="com.mydomain.samples.embedded">
  <class name="Computer" identity-type="datastore" table="COMPUTER">
    <field name="operatingSystem">
      <column name="OS_NAME" length="40" jdbc-type="CHAR"/>
    </field>
    <field name="graphicsCard" persistence-modifier="persistent">
      <embedded null-indicator-column="GRAPHICS_MANUFACTURER">
        <field name="manufacturer" column="GRAPHICS_MANUFACTURER"/>
        <field name="type" column="GRAPHICS_TYPE"/>
      </embedded>
    </field>
    <field name="soundCard" persistence-modifier="persistent">
      <embedded null-indicator-column="SOUND_MANUFACTURER">
        <field name="manufacturer" column="SOUND_MANUFACTURER"/>
        <field name="type" column="SOUND_TYPE"/>
      </embedded>
    </field>
  </class>
  <class name="ComputerCard" table="COMPUTER_CARD">
    <field name="manufacturer"/>
    <field name="type"/>
  </class>
</package>
```

So here we will end up with a TABLE called "COMPUTER" with columns "COMPUTER_ID", "OS_NAME", "GRAPHICS_MANUFACTURER", "GRAPHICS_TYPE", "SOUND_MANUFACTURER", "SOUND_TYPE". If we call makePersistent() on any objects of type `Computer`, they will be persisted into this table.

You will notice in the MetaData our use of the attribute `null-indicator-column`. This is used when retrieving objects from the datastore and detecting if it is a NULL embedded object. In the case we have here, if the column `GRAPHICS_MANUFACTURER` is null at retrieval, then the embedded "graphicsCard" field will be set as null. Similarly for the "soundCard" field when `SOUND_MANUFACTURER` is null.
If the **ComputerCard** class above has a reference back to the related **Computer**, JDO defines a mechanism whereby this will be populated. You would add the XML element `owner-field` to the `<embedded>` tag defining the field within **ComputerCard** that represents the **Computer** it relates to. When this is specified DataNucleus will populate it automatically, so that when you retrieve the **Computer** and access the **ComputerCard** objects within it, they will have the link in place.

It should be noted that in this latter (embedded) case we can still persist objects of type **ComputerCard** into their own table - the MetaData definition for **ComputerCard** is used for the table definition in this case.

Please note that if, instead of specifying the `<embedded>` block we had specified `embedded` in the field element we would have ended up with the same thing, just that the fields and columns would have been mapped using their default mappings, and that the `<embedded>` provides control over how they are mapped.

**Note that by default the embedded objects cannot have inheritance.** Inheritance in embedded objects is only support for RDBMS and MongoDB, and involves defining a discriminator in the metadata of the embedded type.

See also :-

- MetaData reference for `<embedded>` element
- Annotations reference for @Embedded

### 54.1.2 Embedding Nested PersistenceCapable objects

Applicable to RDBMS, Excel, OOXML, ODF, HBase, MongoDB, Neo4j.

In the above example we had an embedded PersistenceCapable object within a persisted object. What if our embedded PersistenceCapable object also contain another PersistenceCapable object. So, using the above example what if **ComputerCard** contains an object of type **Connector**?
public class ComputerCard
{
    ... 
    Connector connector;

    public ComputerCard(String manufacturer,
                         int type,
                         Connector conn)
    {
        this.manufacturer = manufacturer;
        this.type = type;
        this.connector = conn;
    }

    ... 
}

public class Connector
{
    int type;
}

Well we want to store all of these objects into the same record in the COMPUTER table.
So we simply nest the embedded definition of the Connector objects within the embedded definition of the ComputerCard definitions for Computer. JDO supports this to as many levels as you require! The Connector objects will be persisted into the GRAPHICSCONNECTOR_TYPE, and SOUNDCONNECTOR_TYPE columns in the COMPUTER table.
54.1.3 Embedding Collection Elements

Applicable to RDBMS, MongoDB

In a typical 1-N relationship between 2 classes, the 2 classes in the relationship are persisted to their own table, and either a join table or a foreign key is used to relate them. With JDO and DataNucleus you have a variation on the join table relation where you can persist the objects of the "N" side into the join table itself so that they don't have their own identity, and aren't stored in the table for that class. This is supported in DataNucleus with the following provisos

- You can have inheritance in embedded keys/values and a discriminator is added (you must define the discriminator in the metadata of the embedded type).
- When retrieving embedded elements, all fields are retrieved in one call. That is, fetch plans are not utilised. This is because the embedded element has no identity so we have to retrieve all initially.

It should be noted that where the collection "element" is not PersistenceCapable or of a "reference" type (Interface or Object) it will always be embedded, and this functionality here applies to PersistenceCapable elements only. DataNucleus doesn't support the embedding of reference type objects currently.

Let's take an example. We are modelling a Network, and in our simple model our Network has collection of Devices. So we define our classes as

<table>
<thead>
<tr>
<th>COMPUTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPUTER_ID</td>
</tr>
<tr>
<td>OS_NAME</td>
</tr>
<tr>
<td>GRAPHICS_MANUFACTURER</td>
</tr>
<tr>
<td>GRAPHICS_TYPE</td>
</tr>
<tr>
<td>GRAPHICS_CONNECTOR_TYPE</td>
</tr>
<tr>
<td>SOUND_MANUFACTURER</td>
</tr>
<tr>
<td>SOUND_TYPE</td>
</tr>
<tr>
<td>SOUND_CONNECTOR_TYPE</td>
</tr>
</tbody>
</table>
public class Network {
    private String name;
    private Collection devices = new HashSet();
    public Network(String name) {
        this.name = name;
    }
    ...
}

class Device {
    private String name;
    private String ipAddress;
    public Device(String name, String addr) {
        this.name = name;
        this.ipAddress = addr;
    }
    ...
}

We decide that instead of Device having its own table, we want to persist them into the join table of its relationship with the Network since they are only used by the network itself. We define our MetaData like this
So here we will end up with a table called "NETWORK" with columns "NETWORK_ID", and "NAME", and a table called "NETWORK_DEVICES" with columns "NETWORK_ID", "ADPT_PK_IDX", "DEVICE_NAME", "DEVICE_IP_ADDR". When we persist a Network object, any devices are persisted into the NETWORK_DEVICES table.

Please note that if, instead of specifying the <embedded> block we had specified embedded-element in the collection element we would have ended up with the same thing, just that the fields and columns would be mapped using their default mappings, and that the <embedded> provides control over how they are mapped.
You note that in our example above DataNucleus has added an extra column "ADPT_PK_IDX" to provide the primary key of the join table now that we're storing the elements as embedded. A variation on this would have been if we wanted to maybe use the "DEVICE_IP_ADDR" as the other part of the primary key, in which case the "ADPT_PK_IDX" would not be needed. You would specify this as follows:

```xml
<field name="devices" persistence-modifier="persistent" table="NETWORK_DEVICES">
    <collection element-type="com.mydomain.samples.embedded.Device"/>
    <join>
        <primary-key name="NETWORK_DEV_PK">
            <column name="NETWORK_ID"/>
            <column name="DEVICE_IP_ADDR"/>
        </primary-key>
        <column name="NETWORK_ID"/>
    </join>
    <element>
        <embedded>
            <field name="name">
                <column name="DEVICE_NAME" allows-null="true"/>
            </field>
            <field name="ipAddress">
                <column name="DEVICE_IP_ADDR" allows-null="true"/>
            </field>
        </embedded>
    </element>
</field>
```

This results in the join table only having the columns "NETWORK_ID", "DEVICE_IP_ADDR", and "DEVICE_NAME", and having a primary key as the composite of "NETWORK_ID" and "DEVICE_IP_ADDR".

See also:

- MetaData reference for <embedded> element
- MetaData reference for <element> element
- MetaData reference for <join> element
- Annotations reference for @Embedded
- Annotations reference for @Element

### 54.1.4 Embedding Map Keys/Values

**Applicable to RDBMS, MongoDB**

In a typical 1-N map relationship between classes, the classes in the relationship are persisted to their own table, and a join table forms the map linkage. With JDO and DataNucleus you have a variation on the join table relation where you can persist either the key class or the value class, or both key class and value class into the join table. **This is supported in DataNucleus with the following provisos**

- You can have inheritance in embedded keys/values and a discriminator is added (you must define the discriminator in the metadata of the embedded type).
• When retrieving embedded keys/values, all fields are retrieved in one call. That is, fetch plans are not utilised. This is because the embedded key/value has no identity so we have to retrieve all initially.

It should be noted that where the map "key"/"value" is not PersistenceCapable or of a "reference" type (Interface or Object) it will always be embedded, and this functionality here applies to PersistenceCapable keys/values only. DataNucleus doesn't support embedding reference type elements currently.

Let's take an example. We are modelling a FilmLibrary, and in our simple model our FilmLibrary has map of Films, keyed by a String alias. So we define our classes as

```java
public class FilmLibrary {
    private String owner;
    private Map films = new HashMap();
    public FilmLibrary(String owner) {
        this.owner = owner;
    }
    ...
}
public class Film {
    private String name;
    private String director;
    public Film(String name, String director) {
        this.name = name;
        this.director = director;
    }
    ...
}
```

We decide that instead of Film having its own table, we want to persist them into the join table of its map relationship with the FilmLibrary since they are only used by the library itself. We define our MetaData like this
So here we will end up with a table called "FILM_LIBRARY" with columns "FILM_LIBRARY_ID", and "OWNER", and a table called "FILM_LIBRARY_FILMS" with columns "FILM_LIBRARY_ID", "FILM_ALIAS", "FILM_NAME", "FILM_DIRECTOR". When we persist a FilmLibrary object, any films are persisted into the FILM_LIBRARY_FILMS table.

Please note that if, instead of specifying the <embedded> block we had specified embedded-key of embedded-value in the map element we would have ended up with the same thing, just that the fields and columns would be mapped using their default mappings, and that the <embedded> provides control over how they are mapped.
See also :-

- MetaData reference for `<embedded>` element
- MetaData reference for `<key>` element
- MetaData reference for `<value>` element
- MetaData reference for `<join>` element
- Annotations reference for `@Embedded`
- Annotations reference for `@Key`
- Annotations reference for `@Value`
55 Serialised Fields

55.1 JDO : Serialising Fields

JDO provides a way for users to specify that a field will be persisted serialised. This is of use, for example, to collections/maps/arrays which typically are stored using join tables or foreign-keys to other records. By specifying that a field is serialised a column will be added to store that field and the field will be serialised into it.

JDO's definition of serialising encompasses several types of fields. These are described below:

- **Serialised Array fields** - where you want to serialise the array into a single "BLOB" column.
- **Serialised Collection fields** - where you want to serialise the collection into a single "BLOB" column.
- **Serialised Collection elements** - where you want to serialise the collection elements into a single column in a join table.
- **Serialised Map fields** - where you want to serialise the map into a single "BLOB" column.
- **Serialised Map keys/values** - where you want to serialise the map keys and/or values into single column(s) in a join table.
- **Serialised PersistenceCapable fields** - where you want to serialise a PC object into a single "BLOB" column.
- **Serialised Reference (Interface/Object) fields** - where you want to serialise a reference field into a single "BLOB" column.
- **Serialised field to local disk** - not part of the JDO spec but available as an option for RDBMS datastores usage.

Perhaps the most important thing to bear in mind when deciding to serialise a field is that that object must implement `java.io.Serializable`.

55.1.1 Serialised Collections

Applicable to RDBMS, HBase, MongoDB

Collections are usually persisted by way of either a *join table*, or by use of a *foreign-key* in the element table. In some situations it is required to store the whole collection in a single column in the table of the class being persisted. This prohibits the querying of such a collection, but will persist the collection in a single statement. Let's take an example. We have the following classes:

![Class Diagram](image)

and we want the *animals* collection to be serialised into a single column in the table storing the *Farm* class, so we define our MetaData like this:
So we make use of the `serialized` attribute of `<field>`. This specification results in a table like this

Provisos to bear in mind are

- Queries cannot be performed on collections stored as serialised.

There are some other combinations of MetaData tags that result in serialising of the whole collection in the same way. These are as follows

- **Collection of non- PersistenceCapable elements, and no `<join>` is specified.** Since the elements don't have a table of their own, the only option is to serialise the whole collection and it appears as a single BLOB field in the table of the main class.

- **Collection of PersistenceCapable elements, with "embedded-element" set to true and no `<join>` is specified.** Since the elements are embedded and there is no join table, then the whole collection is serialised as above.

See also :-

- MetaData reference for `<field>` element
- Annotations reference for `@Persistent`
- Annotations reference for `@Serialized`

### 55.1.2 Serialised Collection Elements

Applicable to RDBMS

Collections are usually persisted by way of either a `join table`, or by use of a `foreign-key` in the element table. In some situations you may want to serialise the element into a single column in the join table. Let's take an example. We have the same classes as in the previous case and we want the `animals` collection to be stored in a join table, and the element serialised into a single column storing the "Animal" object. We define our MetaData like this
So we make use of the `serialized-element` attribute of `<collection>`. This specification results in tables like this

![Diagram of FARM and FARM_ANIMALS tables](image)

Provisos to bear in mind are

- Queries cannot be performed on collection elements stored as serialised.

See also :-

- MetaData reference for `<collection>` element
- MetaData reference for `<join>` element
- Annotations reference for @Element

### 55.1.3 Serialised Maps

Maps are usually persisted by way of a `join table`, or very occasionally using a `foreign-key` in the value table. In some situations it is required to store the whole map in a single column in the table of the class being persisted. This prohibits the querying of such a map, but will persist the map in a single statement. Let's take an example. We have the following classes
and we want the children map to be serialised into a single column in the table storing the **ClassRoom** class, so we define our MetaData like this

```xml
<class name="ClassRoom">
  <field name="level">
    <column name="LEVEL"/>
  </field>
  <field name="children" serialized="true">
    <map key-type="java.lang.String" value-type="Child">
      <column name="CHILDREN"/>
    </map>
  </field>
</class>
<class name="Child"/>
```

So we make use of the **serialized** attribute of `<field>`. This specification results in a table like this

![ClassRoom diagram](image.png)

Provisos to bear in mind are

- Queries cannot be performed on maps stored as serialised.

There are some other combinations of MetaData tags that result in serialising of the whole map in the same way. These are as follows

- **Map**<non-PersistenceCapable, non-PersistenceCapable>, and no `<join>` is specified. Since the keys/values don't have a table of their own, the only option is to serialise the whole map and it appears as a single BLOB field in the table of the main class.

- **Map**<non-PersistenceCapable, PersistenceCapable>, with "embedded-value" set to true and no `<join>` is specified. Since the keys/values are embedded and there is no join table, then the whole map is serialised as above.

See also :-

- MetaData reference for `<map>` element
- Annotations reference for `@Key`
- Annotations reference for `@Value`
- Annotations reference for `@Serialized`
55.1.4 Serialised Map Keys/Values

Maps are usually persisted by way of a join table, or very occasionally using a foreign-key in the value table. In the join table case you have the option of serialising the keys and/or the values each into a single (BLOB) column in the join table. This is performed in a similar way to serialised elements for collections, but this time using the "serialized-key", "serialized-value" attributes. We take the example in the previous section, with "a classroom of children" and the children stored in a map field. This time we want to serialise the child object into the join table of the map.

```xml
<class name="ClassRoom">
    <field name="level">
        <column name="LEVEL"/>
    </field>
    <field name="children" table="CLASS_CHILDREN">
        <map key-type="java.lang.String" value-type="Child" serialized-value="true"/>
        <join column="CLASSROOM_ID"/>
        <key column="ALIAS"/>
        <value column="CHILD"/>
    </field>
</class>
<class name="Child"/>
```

So we make use of the serialized-value attribute of <map>. This results in a schema like this:

![Diagram showing the serialized map keys/values structure](image)

Provisos to bear in mind are:

- Queries cannot be performed on map keys/values stored as serialised.

See also:

- MetaData reference for <map> element
- MetaData reference for <join> element
- MetaData reference for <key> element
- MetaData reference for <value> element
- Annotations reference for @Key
- Annotations reference for @Value

55.1.5 Serialised PersistenceCapable Fields

Applicable to RDBMS, HBase, MongoDB
A field that is a PersistenceCapable object is typically stored as a foreign-key relation between the container object and the contained object. In some situations it is not necessary that the contained object has an identity of its own, and for efficiency of access the contained object is required to be stored in a BLOB column in the containing object’s datastore table. Let's take an example. We have the following classes

![Diagram showing the relationship between ClassRoom and Teacher objects.](image)

and we want the teacher object to be serialised into a single column in the table storing the ClassRoom class, so we define our MetaData like this

```xml
<class name="ClassRoom">
    <field name="level">
        <column name="LEVEL"/>
    </field>
    <field name="teacher" serialized="true">
        <column name="TEACHER"/>
    </field>
</class>
```

So we make use of the serialized attribute of <field>. This specification results in a table like this

![Table showing the serialized fields.](image)

Provisos to bear in mind are

- Queries cannot be performed on PersistenceCapable objects stored as serialised.

### 55.1.6 Serialised Reference (Interface/Object) Fields

**Applicable to RDBMS**

A reference (Interface/Object) field is typically stored as a foreign-key relation between the container object and the contained implementation of the reference. In some situations it is not necessary that the contained object has an identity of its own, and for efficiency of access the contained object is required to be stored in a BLOB column in the containing object's datastore table. Let's take an example using an interface field. We have the following classes
and we want the `teacher` object to be serialised into a single column in the table storing the `ClassRoom` class, so we define our MetaData like this

```xml
<class name="ClassRoom">
  <field name="level">
    <column name="LEVEL"/>
  </field>
  <field name="teacher" serialized="true">
    <column name="TEACHER"/>
  </field>
</class>
<class name="Teacher">
</class>
```

So we make use of the `serialized` attribute of `<field>`. This specification results in a table like this

![Diagram of table structure]

Provisos to bear in mind are

- Queries cannot be performed on Reference (Interface/Object) fields stored as serialised.

See also :-

- METAData reference for `<implements>` element
- Annotations reference for `@Serialized`
55.1.7 Serialised Field to Local File

Applicable to RDBMS

If you have a non-relation field that implements Serializable you have the option of serialising it into a file on the local disk. This could be useful where you have a large file and don't want to persist very large objects into your RDBMS. Obviously this will mean that the field is no longer queryable, but then if its a large file you likely don't care about that. So let's give an example

```java
@PersistenceCapable
public class Person
{
    @PrimaryKey
    long id;

    @Persistent
    @Extension(vendorName="datanucleus", key="serializeToFileLocation"
        value="person_avatars")
    AvatarImage image;
}
```

Or using XML

```xml
<class name="Person">
    ...
    <field name="image" persistence-modifier="persistent">
        <extension vendor-name="datanucleus" key="serializeToFileLocation"
            value="person_avatars"/>
    </field>
</class>
```

So this will now persist a file into a folder `person_avatars` with filename as the String form of the identity of the owning object. In a real world example you likely will specify the extension value as an absolute path name, so you can place it anywhere in the local disk.
56 Interface Fields

56.1 JDO : Interface Fields

JDO requires that implementations support the persistence of interfaces as first class objects (FCO's). DataNucleus provides this capability. It follows the same general process as for java.lang.Object since both interfaces and java.lang.Object are basically references to some persistable object.

To demonstrate interface handling let's introduce some classes. Let's suppose you have an interface with a selection of classes implementing the interface something like this:

```
public class ShapeHolder {
    protected Shape shape = null;
    ...
}
```

JDO doesn't define how an interface is persisted in the datastore. Obviously there can be many implementations and so no obvious solution. DataNucleus allows the following:

- **per-implementation**: a FK is created for each implementation so that the datastore can provide referential integrity. The other advantage is that since there are FKs then querying can be performed. The disadvantage is that if there are many implementations then the table can become large with many columns not used.

- **identity**: a single column is added and this stores the class name of the implementation stored, as well as the identity of the object. The advantage is that if you have large numbers of implementations then this can cope with no schema change. The disadvantages are that no querying can be performed, and that there is no referential integrity.

- **xcalia**: a slight variation on "identity" whereby there is a single column yet the contents of that column are consistent with what Xcalia XIC JDO implementation stored there.

The user controls which one of these is to be used by specifying the extension mapping-strategy on the field containing the interface. The default is "per-implementation"
In terms of the implementations of the interface, you can either leave the field to accept any known about implementation, or you can restrict it to only accept some implementations (see "implementation-classes" metadata extension). If you are leaving it to accept any persistable implementation class, then you need to be careful that such implementations are known to DataNucleus at the point of encountering the interface field. By this we mean, DataNucleus has to have encountered the metadata for the implementation so that it can allow for the implementation when handling the field. You can force DataNucleus to know about a persistable class by using an autostart mechanism, or using persistence.xml, or by placement of the package.jdo file so that when the owning class for the interface field is encountered so is the metadata for the implementations.

56.1.1 1-1

JDO2

To allow persistence of this interface field with DataNucleus you have 2 levels of control. The first level is global control. Since all of our Square, Circle, Rectangle classes implement Shape then we just define them in the MetaData as we would normally.

```xml
<package name="mydomain">
  <class name="Square">
    ...
  </class>
  <class name="Circle">
    ...
  </class>
  <class name="Rectangle">
    ...
  </class>
</package>
```

The global way means that when mapping that field DataNucleus will look at all PersistenceCapable classes it knows about that implement the specified interface.

JDO also allows users to specify a list of classes implementing the interface on a field-by-field basis, defining which of these implementations are accepted for a particular interface field. To do this you define the Meta-Data like this

```xml
<package name="mydomain">
  <class name="ShapeHolder">
    <field name="shape" persistence-modifier="persistent"
            field-type="mydomain.Circle,mydomain.Rectangle,mydomain.Square"/>
  </class>
</package>
```

That is, for any interface object in a class to be persisted, you define the possible implementation classes that can be stored there. DataNucleus interprets this information and will map the above example classes to the following in the database:
So DataNucleus adds foreign keys from the containers table to all of the possible implementation tables for the \textit{shape} field.

If we use \textbf{mapping-strategy} of "identity" then we get a different datastore schema.

\begin{verbatim}
<class name="ShapeHolder">
  <field name="shape" persistence-modifier="persistent">
    <extension vendor-name="datanucleus" key="mapping-strategy" value="identity"/>
  </field>
</class>
\end{verbatim}

and the datastore schema becomes

and the column "SHAPE" will contain strings such as \textit{mydomain.Circle:1} allowing retrieval of the related implementation object.
56.1.2 1-N

You can have a Collection/Map containing elements of an interface type. You specify this in the same way as you would any Collection/Map. **You can have a Collection of interfaces as long as you use a join table relation and it is unidirectional.** The "unidirectional" restriction is that the interface is not persistent on its own and so cannot store the reference back to the owner object. Use the 1-N relationship guides for the metadata definition to use.

You need to use a DataNucleus extension tag "implementation-classes" if you want to restrict the collection to only contain particular implementations of an interface. For example

```xml
<class name="ShapeHolder">
  <field name="shapes" persistence-modifier="persistent">
    <collection element-type="mydomain.Shape"/>
    <join/>
    <extension vendor-name="datanucleus" key="implementation-classes" value="mydomain.Circle,mydomain.Rectangle,mydomain.Square,mydomain.Triangle"/>
  </field>
</class>
```

So the `shapes` field is a Collection of `mydomain.Shape` and it will accept the implementations of type `Circle`, `Rectangle`, `Square` and `Triangle`. If you omit the implementation-classes tag then you have to give DataNucleus a way of finding the metadata for the implementations prior to encountering this field.

56.1.3 Dynamic Schema Updates

The default mapping strategy for interface fields and collections of interfaces is to have separate FK column(s) for each possible implementation of the interface. Obviously if you have an application where new implementations are added over time the schema will need new FK column(s) adding to match. This is possible if you enable the persistence property `datanucleus.rdbms.dynamicSchemaUpdates`, setting it to `true`. With this set, any insert/update operation of an interface related field will do a check if the implementation being stored is known about in the schema and, if not, will update the schema accordingly.
57 Object Fields

57.1 JDO: Fields of type java.lang.Object

JDO requires that implementations support the persistence of java.lang.Object as first class objects (FCO's). DataNucleus provides this capability and also provides that java.lang.Object can be stored as serialised. It follows the same general process as for Interfaces since both interfaces and java.lang.Object are basically references to some persistable object.

java.lang.Object cannot be used to persist non-persistable types with fixed schema datastore (e.g. RDBMS). Think of how you would expect it to be stored if you think it ought to

JDO doesn't define how an object FCO is persisted in the datastore. Obviously there can be many "implementations" and so no obvious solution. DataNucleus allows the following:

- **per-implementation**: a FK is created for each "implementation" so that the datastore can provide referential integrity. The other advantage is that since there are FKs then querying can be performed. The disadvantage is that if there are many implementations then the table can become large with many columns not used
- **identity**: a single column is added and this stores the class name of the "implementation" stored, as well as the identity of the object. The disadvantages are that no querying can be performed, and that there is no referential integrity.
- **xcalia**: a slight variation on "identity" whereby there is a single column yet the contents of that column are consistent with what Xcalia XIC JDO implementation stored there.

The user controls which one of these is to be used by specifying the extension mapping-strategy on the field containing the interface. The default is "per-implementation"

57.1.1 FCO

Let's suppose you have a field in a class and you have a selection of possible persistable class that could be stored there, so you decide to make the field a java.lang.Object. So let's take an example. We have the following class

```java
public class ParkingSpace {
    String location;
    Object occupier;
}
```

So we have a space in a car park, and in that space we have an occupier of the space. We have some legacy data and so can't make the type of this "occupier" an interface type, so we just use java.lang.Object. Now we know that we can only have particular types of objects stored there (since there are only a few types of vehicle that can enter the car park). So we define our MetaData like this
<package name="mydomain.samples.object">
  <class name="ParkingSpace">
    <field name="location"/>
    <field name="occupier" persistence-modifier="persistent"
      field-type="mydomain.samples.vehicles.Car,
      mydomain.samples.vehicles.Motorbike"/>
  </field>
</class>

or using annotations

@Persistent(types={mydomain.samples.vehicles.Car.class, mydomain.samples.vehicles.Motorbike.class})
Object occupier;

This will result in the following database schema.

So DataNucleus adds foreign keys from the ParkingSpace table to all of the possible implementation tables for the occupier field.

In conclusion, when using "per-implementation" mapping for any java.lang.Object field in a class to be persisted (as non-serialised), you must define the possible "implementation" classes that can be stored there.

If we use mapping-strategy of "identity" then we get a different datastore schema.

<class name="ParkingSpace">
  <field name="location"/>
  <field name="occupier" persistence-modifier="persistent">
    <extension vendor-name="datanucleus" key="mapping-strategy" value="identity"/>
  </field>
</class>

and the datastore schema becomes
and the column "OCCUPIER" will contain strings such as `com.mydomain.samples.object.Car:1` allowing retrieval of the related implementation object.

### 57.1.2 Collections of Objects

You can have a Collection/Map containing elements of `java.lang.Object`. You specify this in the same way as you would any Collection/Map. DataNucleus supports having a Collection of references with multiple implementation types as long as you use a join table relation.

### 57.1.3 Serialised Objects

By default, a field of type `java.lang.Object` is stored as an instance of the underlying PersistenceCapable in the table of that object. If either your Object field represents non-PersistenceCapable objects or you simply wish to serialise the Object into the same table as the owning object, you need to specify the "serialized" attribute, like this:

```xml
<class name="MyClass">
  <field name="myObject" serialized="true"/>
</class>
```

Similarly, where you have a collection of Objects using a join table, the objects are, by default, stored in the table of the PersistenceCapable instance. If instead you want them to occupy a single BLOB column of the join table, you should specify the "embedded-element" attribute of `<collection>` like this:

```xml
<class name="MyClass">
  <field name="myCollection">
    <collection element-type="java.lang.Object" serialized-element="true"/>
    <join/>
  </field>
</class>
```

Please refer to the serialised fields guide for more details of storing objects in this way.
58 Array Fields

58.1 JDO: Array fields

JDO allows implementations to optionally support the persistence of arrays. DataNucleus provides full support for arrays in similar ways that collections are supported. DataNucleus supports persisting arrays as

- **Single Column** - the array is byte-streamed into a single column in the table of the containing object.
- **Serialised** - the array is serialised into single column in the table of the containing object.
- **Using a Join Table** - where the array relation is persisted into the join table, with foreign-key links to an element table where the elements of the array are *PersistenceCapable*
- **Using a Foreign-Key in the element** - only available where the array is of a *PersistenceCapable* type

JDO has no simple way of detecting changes to an arrays contents. To update an array you must either

- replace the array field with the new array value
- update the array element and then call `JDOHelper.makeDirty(obj, "fieldName");`

### 58.1.1 Single Column Arrays

Let's suppose you have a class something like this

![Account class diagram](image)

So we have an `Account` and it has a number of permissions, each expressed as a byte. We want to persist the permissions in a single-column into the table of the account (but we don't want them serialised). We then define `MetaData` something like this

```xml
<class name="Account" identity-type="datastore">
  <field name="firstName">
    <column name="FIRST_NAME" length="100" jdbc-type="VARCHAR"/>
  </field>
  <field name="lastName">
    <column column="LAST_NAME" length="100" jdbc-type="VARCHAR"/>
  </field>
  <field name="permissions" column="PERMISSIONS"/>
</class>
```

You could have added `<array>` to be explicit but the type of the field is an array, and the type declaration also defines the component type so nothing more is needed. This results in a datastore schema as follows
DataNucleus supports persistence of the following array types in this way: `boolean[]`, `byte[]`, `char[]`, `double[]`, `float[]`, `int[]`, `long[]`, `short[]`, `Boolean[]`, `Byte[]`, `Character[]`, `Double[]`, `Float[]`, `Integer[]`, `Long[]`, `Short[]`, `BigDecimal[]`, `BigInteger[]`

See also:
- MetaData reference for `<array>` element
- Annotations reference for `@Element`

### 58.1.2 Serialised Arrays

Let's suppose you have a class something like this:

```java
public class Account {
    private String firstName;
    private String lastName;
    private byte[] permissions;
}
```

So we have an `Account` and it has a number of permissions, each expressed as a byte. We want to persist the permissions as serialised into the table of the account. We then define MetaData something like this:

```xml
<class name="Account" identity-type="datastore">
    <field name="firstName">
        <column name="FIRST_NAME" length="100" jdbc-type="VARCHAR"/>
    </field>
    <field name="lastName">
        <column name="LAST_NAME" length="100" jdbc-type="VARCHAR"/>
    </field>
    <field name="permissions" serialized="true" column="PERMISSIONS"/>
</class>
```

That is, you define the field as `serialized`. To define arrays of short, long, int, or indeed any other supported array type you would do the same as above. This results in a datastore schema as follows:
DataNucleus supports persistence of many array types in this way, including: boolean[], byte[], char[], double[], float[], int[], long[], short[], Boolean[], Byte[], Character[], Double[], Float[], Integer[], Long[], Short[], BigDecimal[], BigInteger[], String[], java.util.Date[], java.util.Locale[]

See also:
- MetaData reference for <field> element
- MetaData reference for <array> element
- Annotations reference for @Persistent
- Annotations reference for @Element
- Annotations reference for @Serialized

### 58.1.3 Arrays persisted into Join Tables

DataNucleus will support arrays persisted into a join table. Let's take the example above and make the "permission" a class in its own right, so we have

So an Account has an array of Permissions, and both of these objects are PersistenceCapable. We want to persist the relationship using a join table. We define the MetaData as follows
This results in a datastore schema as follows

### ACCOUNT

- ACCOUNT_ID
- FIRST_NAME
- LAST_NAME

### ACCOUNT_PERMISSIONS

- ACCOUNT_ID
- PERMISSION_ID
- PERMISSION_ORDER_IDX

### PERMISSION

- PERMISSION_ID
- NAME

**See also:**

- MetaData reference for `<array>` element
- MetaData reference for `<element>` element
- MetaData reference for `<join>` element
- MetaData reference for `<order>` element
- Annotations reference for `@Element`
- Annotations reference for `@Order`

#### 58.1.4 Arrays persisted using Foreign-Keys

DataNucleus will support arrays persisted via a foreign-key in the element table. This is only applicable when the array is of a `PersistenceCapable` type. Let's take the same example above. So we have
So an **Account** has an array of **Permissions**, and both of these objects are **PersistenceCapable**. We want to persist the relationship using a foreign-key in the table for the Permission class. We define the MetaData as follows

```
<class name="Account" table="ACCOUNT">
  <field name="firstName">
    <column name="FIRST_NAME" length="100" jdbc-type="VARCHAR"/>
  </field>
  <field name="lastName">
    <column name="LAST_NAME" length="100" jdbc-type="VARCHAR"/>
  </field>
  <field name="permissions">
    <array/>
    <element column="ACCOUNT_ID"/>
    <order column="ACCOUNT_PERMISSION_ORDER_IDX"/>
  </field>
</class>
<class name="Permission" table="PERMISSION">
  <field name="name"/>
</class>
```

This results in a datastore schema as follows

See also :-

- MetaData reference for `<array>` element
- MetaData reference for `<element>` element
- MetaData reference for `<order>` element
- Annotations reference for `@Element`
- Annotations reference for `@Order`
59 1-to-1 Relations

59.1 JDO : 1-1 Relationships
You have a 1-to-1 relationship when an object of a class has an associated object of another class (only one associated object). It could also be between an object of a class and another object of the same class (obviously). You can create the relationship in 2 ways depending on whether the 2 classes know about each other (bidirectional), or whether only one of the classes knows about the other class (unidirectional). These are described below.

The various possible relationships are described below.

- 1-1 Unidirectional (where only 1 object is aware of the other)
- 1-1 Bidirectional (where both objects are aware of each other)
- 1-1 Unidirectional "Compound Identity" (object as part of PK in other object)

59.1.1 Unidirectional
For this case you could have 2 classes, User and Account, as below.

```
so the Account class knows about the User class, but not vice-versa. If you define the XML metadata for these classes as follows
```

```xml
<package name="mydomain">
  <class name="User" table="USER">
    <field name="id" primary-key="true">
      <column name="USER_ID"/>
    </field>
    ...
  </class>

  <class name="Account" table="ACCOUNT">
    <field name="id" primary-key="true">
      <column name="ACCOUNT_ID"/>
    </field>
    ...
    <field name="user">
      <column name="USER_ID"/>
    </field>
  </class>
</package>
```
or alternatively using annotations

```java
public class Account {
    ...

    @Column(name="USER_ID")
    User user;
}

public class User {
    ...
}
```

This will create 2 tables in the database, one for **User** (with name `USER`), and one for **Account** (with name `ACCOUNT` and a column `USER_ID`), as shown below.

Things to note :-

- **Account** has the object reference (and so owns the relation) to **User** and so its table holds the foreign-key.
- If you call `PM.deletePersistent()` on the end of a 1-1 unidirectional relation without the relation and that object is related to another object, an exception will typically be thrown (assuming the RDBMS supports foreign keys). To delete this record you should remove the other objects association first.
- If you invoke an operation that will retrieve the one-to-one field, and you only want it to get the foreign key value (and not join to the related table) you can add the metadata extension `fetch-fk-only` (set to "true") to the field/property.

### 59.1.2 Bidirectional

For this case you could have 2 classes, **User** and **Account** again, but this time as below. Here the **Account** class knows about the **User** class, and also vice-versa.
Here we create the 1-1 relationship with a single foreign-key. To do this you define the XML metadata as

```xml
<package name="mydomain">
  <class name="User" table="USER">
    <field name="id" primary-key="true">
      <column name="USER_ID"/>
    </field>
    ...
    <field name="account" mapped-by="user"/>
  </class>

  <class name="Account" table="ACCOUNT">
    <field name="id" primary-key="true">
      <column name="ACCOUNT_ID"/>
    </field>
    ...
    <field name="user">
      <column name="USER_ID"/>
    </field>
  </class>
</package>
```

or alternatively using annotations

```java
public class Account {
  ...
  @Column(name="USER_ID")
  User user;
}

public class User {
  ...
  @Persistent(mappedBy="user")
  Account account;
}
```
The difference is that we added `mapped-by` to the field of `User`. This represents the bidirectionality.

This will create 2 tables in the database, one for `User` (with name `USER`), and one for `Account` (with name `ACCOUNT`). With RDBMS the `ACCOUNT` table will have a column `USER_ID` (since RDBMS will place the FK on the side without the "mapped-by"). Like this

With non-RDBMS datastores both tables will have a column containing the "id" of the related object, that is `USER` will have an `ACCOUNT` column, and `ACCOUNT` will have a `USER_ID` column.

Things to note :-

- When forming the relation please make sure that **you set the relation at BOTH sides** since DataNucleus would have no way of knowing which end is correct if you only set one end.
- If you invoke an operation that will retrieve the one-to-one field (of the non-owner side), and you only want it to get the foreign key value (and not join to the related table) you can add the metadata extension `fetch-fk-only` (set to "true") to the field/property.

59.1.3 Embedded

The above 2 relationship types assume that both classes in the 1-1 relation will have their own table. You can, of course, embed the elements of one class into the table of the other. This is described in **Embedded PC Objects**.
60 1-to-N Relations

60.1 JDO : 1-N Relationships

You have a 1-N (one to many) when you have one object of a class that has a Collection/Map of objects of another class. In the `java.util` package there are an assortment of possible collection/map classes and they all have subtly different behaviour with respect to allowing nulls, allowing duplicates, providing ordering, etc. There are two ways in which you can represent a collection or map in a datastore: **Join Table** (where a join table is used to provide the relationship mapping between the objects), and **Foreign-Key** (where a foreign key is placed in the table of the object contained in the collection or map).

We split our documentation based on what type of collection/map you are using.

- 1-N using Collection types
- 1-N using Set types
- 1-N using List type
- 1-N using Map type
61 Collections

61.1 JDO : 1-N Relationships with Collections

You have a 1-N (one to many) or N-1 (many to one) when you have one object of a class that has a Collection of objects of another class. Please note that Collections allow duplicates, and so the persistence process reflects this with the choice of primary keys. There are two ways in which you can represent this in a datastore: Join Table (where a join table is used to provide the relationship mapping between the objects), and Foreign-Key (where a foreign key is placed in the table of the object contained in the Collection).

The various possible relationships are described below.

- 1-N Unidirectional using Join Table
- 1-N Unidirectional using Foreign-Key
- 1-N Bidirectional using Join Table
- 1-N Bidirectional using Foreign-Key
- 1-N Unidirectional of non-PC using Join Table
- 1-N embedded elements using Join Table
- 1-N Serialised collection
- 1-N using shared join table
- 1-N using shared foreign key
- 1-N Bidirectional "Compound Identity" (owner object as part of PK in element)

Important: If you declare a field as a Collection, you can instantiate it as either Set-based or as List-based. With a List an "ordering" column is required, whereas with a Set it isn't. Consequently DataNucleus needs to know if you plan on using it as Set-based or List-based. You do this by adding an "order" element to the field if it is to be instantiated as a List-based collection. If there is no "order" element, then it will be assumed to be Set-based.

Please note that RDBMS supports the full range of options on this page, whereas other datastores (ODF, Excel, HBase, MongoDB, etc) persist the Collection in a column in the owner object (as well as a column in the non-owner object when bidirectional) rather than using join-tables or foreign-keys since those concepts are RDBMS-only.

61.1.1 equals() and hashCode()

Important: The element of a Collection ought to define the methods equals and hashCode so that updates are detected correctly. This is because any Java Collection will use these to determine equality and whether an element is contained in the Collection. Note also that the hashCode() should be consistent throughout the lifetime of a persistable object. By that we mean that it should not use some basis before persistence and then use some other basis (such as the object identity) after persistence, for this reason we do not recommend usage of JDOHelper.getObjectId(obj) in the equals/hashCode methods.

61.2 1-N Collection Unidirectional

We have 2 sample classes Account and Address. These are related in such a way as Account contains a Collection of objects of type Address, yet each Address knows nothing about the Account objects that it relates to. Like this
There are 2 ways that we can persist this relationship. These are shown below

61.2.1 Using Join Table

If you define the XML metadata for these classes as follows

```xml
<package name="com.mydomain">
  <class name="Account">
    <field name="id" primary-key="true">
      <column name="ACCOUNT_ID"/>
    </field>
    ...
    <field name="addresses" table="ACCOUNT_ADDRESSES">
      <collection element-type="com.mydomain.Address"/>
      <join column="ACCOUNT_ID_OID"/>
      <element column="ADDRESS_ID_EID"/>
    </field>
  </class>

  <class name="Address">
    <field name="id" primary-key="true">
      <column name="ADDRESS_ID"/>
    </field>
    ...
  </class>
</package>
```

or alternatively using annotations
public class Account
{
    ...
    @Persistent(table="ACCOUNT_ADDRESSES")
    @Join(column="ACCOUNT_ID_OID")
    @Element(column="ADDRESS_ID_EID")
    Collection<Address> addresses;
}

public class Address
{
    ...
}

The crucial part is the join element on the field element - this signals to JDO to use a join table.

This will create 3 tables in the database, one for Address, one for Account, and a join table, as shown below.

<table>
<thead>
<tr>
<th>ACCOUNT</th>
<th>ACCOUNT_ADDRESSES</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ACCOUNT_ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIRSTNAME</td>
<td>+ACCOUNT_ID_OID</td>
<td></td>
</tr>
<tr>
<td>LASTNAME</td>
<td>#ADDRESS_ID_EID</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+ADPT_PK_IDX</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADDRESS_ID</td>
<td></td>
</tr>
<tr>
<td></td>
<td>STREET</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CITY</td>
<td></td>
</tr>
</tbody>
</table>

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table attribute on the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the field element.
- To specify the name of the join table, specify the table attribute on the field element with the collection.
- To specify the names of the join table columns, use the column attribute of join, element elements.
- To specify the foreign-key between container table and join table, specify <foreign-key> below the <join> element.
- To specify the foreign-key between join table and element table, specify <foreign-key> below either the <field> element or the <element> element.
- If you wish to share the join table with another relation then use the DataNucleus "shared join table" extension
- The join table will, by default, be given a primary key. If you want to omit this then you can turn it off using the DataNucleus metadata extension "primary-key" (within <join>) set to false.
- The column "ADPT_PK_IDX" is added by DataNucleus so that duplicates can be stored. You can control this by adding an <order> element and specifying the column name for the order
column, or you can override the default naming of this column by specifying the DataNucleus extension "adapter-column-name" (within <field>).

- If you want the set to include nulls, you can turn on this behaviour by adding the DataNucleus extension metadata "allow-nulls" to the <field> set to true

### 61.2.2 Using Foreign-Key

In this relationship, the Account class has a List of Address objects, yet the Address knows nothing about the Account. In this case we don't have a field in the Address to link back to the Account and so DataNucleus has to use columns in the datastore representation of the Address class. So we define the XML metadata like this

```xml
<package name="com.mydomain">
    <class name="Account">
        <field name="id" primary-key="true">
            <column name="ACCOUNT_ID"/>
        </field>
        ...
        <field name="addresses">
            <collection element-type="com.mydomain.Address"/>
            <element column="ACCOUNT_ID"/>
        </field>
    </class>
    <class name="Address">
        <field name="id" primary-key="true">
            <column name="ADDRESS_ID"/>
        </field>
        ...
    </class>
</package>
```

or alternatively using annotations

```java
public class Account
{
    ...

    @Element(column="ACCOUNT_ID")
    Collection<Address> addresses;
}

public class Address
{
    ...
}
```

Again there will be 2 tables, one for Address, and one for Account. Note that we have no "mapped-by" attribute specified, and also no "join" element. If you wish to specify the names of the columns used in the schema for the foreign key in the Address table you should use the `element` element within the field of the collection.
In terms of operation within your classes of assigning the objects in the relationship. You have to take your Account object and add the Address to the Account collection field since the Address knows nothing about the Account.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the `table` attribute on the `<class>` element.
- To specify the names of the columns where the fields of a class are stored, specify the `column` attribute on the `<field>` element.
- To specify the foreign-key between container table and element table, specify `<foreign-key>` below either the `<field>` element or the `<element>` element.

**Limitation**: Since each Address object can have at most one owner (due to the "Foreign Key") this mode of persistence will not allow duplicate values in the Collection. If you want to allow duplicate Collection entries, then use the "Join Table" variant above.

### 61.3 1-N Collection Bidirectional

We have 2 sample classes Account and Address. These are related in such a way as Account contains a Collection of objects of type Address, and each Address has a reference to the Account object that it relates to. Like this

There are 2 ways that we can persist this relationship. These are shown below

#### 61.3.1 Using Join Table

If you define the XML metadata for these classes as follows
The crucial part is the `join` element on the field element - this signals to JDO to use a join table.

This will create 3 tables in the database, one for Address, one for Account, and a join table, as shown below.

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.
If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the `table` attribute on the `class` element.
- To specify the names of the columns where the fields of a class are stored, specify the `column` attribute on the `field` element.
- To specify the name of the join table, specify the `table` attribute on the `field` element with the `collection`.
- To specify the names of the join table columns, use the `column` attribute of `join`, `element` elements.
- To specify the foreign-key between container table and join table, specify `<foreign-key>` below the `<join>` element.
- To specify the foreign-key between join table and element table, specify `<foreign-key>` below either the `<field>` element or the `<element>` element.
- If you wish to share the join table with another relation then use the DataNucleus "shared join table" extension.
- The join table will, by default, be given a primary key. If you want to omit this then you can turn it off using the DataNucleus metadata extension "primary-key" (within `<join>`) set to false.
- The column "ADPT_PK_IDX" is added by DataNucleus so that duplicates can be stored. You can control this by adding an `<order>` element and specifying the column name for the order column, or you can override the default naming of this column by specifying the DataNucleus extension "adapter-column-name" (within `<field>`).
- When forming the relation please make sure that you set the relation at BOTH sides since DataNucleus would have no way of knowing which end is correct if you only set one end.
- If you want the set to include nulls, you can turn on this behaviour by adding the extension metadata "allow-nulls" to the `<field>` set to true.

### 61.3.2 Using Foreign-Key

Here we have the 2 classes with both knowing about the relationship with the other.

If you define the XML metadata for these classes as follows...
or alternatively using annotations

```java
public class Account {
    ...
    @Persistent(mappedBy="account")
    Collection<Address> addresses;
}

public class Address {
    ...
}
```

The crucial part is the `mapped-by` on the "1" side of the relationship. This tells the JDO implementation to look for a field called `account` on the Address class.

This will create 2 tables in the database, one for Address (including an `ACCOUNT_ID` to link to the `ACCOUNT` table), and one for Account. Notice the subtle difference to this set-up to that of the Join Table relationship earlier.
If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the `table` attribute on the `class` element.
- To specify the names of the columns where the fields of a class are stored, specify the `column` attribute on the `field` element.
- To specify the foreign-key between container table and element table, specify `<foreign-key>` below either the `<field>` element or the `<element>` element.
- When forming the relation please make sure that you set the relation at BOTH sides since DataNucleus would have no way of knowing which end is correct if you only set one end.

**Limitation:** Since each Address object can have at most one owner (due to the "Foreign Key") this mode of persistence will not allow duplicate values in the Collection. If you want to allow duplicate Collection entries, then use the "Join Table" variant above.

### 61.4 1-N Collection of non-PersistenceCapable objects

All of the examples above show a 1-N relationship between 2 PersistenceCapable classes. DataNucleus can also cater for a Collection of primitive or Object types. For example, when you have a Collection of Strings. This will be persisted in the same way as the "Join Table" examples above. A join table is created to hold the collection elements. Let's take our example. We have an `Account` that stores a Collection of addresses. These addresses are simply Strings. We define the XML metadata like this

```xml
<package name="com.mydomain">
  <class name="Account">
    <field name="id" primary-key="true">
      <column name="ACCOUNT_ID"/>
    </field>
    <!-- ... -->
    <field name="addresses" persistence-modifier="persistent">
      <collection element-type="java.lang.String"/>
      <join/>
      <element column="ADDRESS"/>
    </field>
  </class>
</package>
```

or alternatively using annotations
public class Account
{
    ...

    @Persistent
    @Join
    @Element(column="ADDRESS")
    Collection<String> addresses;
}

In the datastore the following is created

The ACCOUNT table is as before, but this time we only have the "join table". In our MetaData we used the <element> tag to specify the column name to use for the actual address String.

Please note that the column ADPT_PK_IDX is added by DataNucleus so that duplicates can be stored. You can override the default naming of this column by specifying the DataNucleus extension "adapter-column-name" within the <field> for the Collection.

61.5 Embedded into a Join Table

The above relationship types assume that both classes in the 1-N relation will have their own table. A variation on this is where you have a join table but you embed the elements of the collection into this join table. To do this you use the embedded-element attribute on the collection MetaData element. This is described in Embedded Collection Elements.

61.6 Serialised into a Join Table

The above relationship types assume that both classes in the 1-N relation will have their own table. A variation on this is where you have a join table but you serialise the elements of the collection into this join table in a single column. To do this you use the serialised-element attribute on the collection MetaData element. This is described in Serialised Collection Elements.

61.7 Shared Join Tables

The relationships using join tables shown above rely on the join table relating to the relation in question. DataNucleus allows the possibility of sharing a join table between relations. The example below demonstrates this. We take the example as show above (1-N Unidirectional Join table relation), and extend Account to have 2 collections of Address records. One for home addresses and one for work addresses, like this
We now change the metadata we had earlier to allow for 2 collections, but sharing the join table

```xml
<package name="com.mydomain">
  <class name="Account">
    <field name="id" primary-key="true">
      <column name="ACCOUNT_ID"/>
    </field>
    ...
    <field name="workAddresses" persistence-modifier="persistent" table="ACCOUNT_ADDRESSES">
      <collection element-type="com.mydomain.Address">
        <join column="ACCOUNT_ID_OID"/>
        <element column="ADDRESS_ID_EID"/>
        <extension vendor-name="datanucleus" key="relation-discriminator-column" value="ADDRESS_TYPE"/>
        <extension vendor-name="datanucleus" key="relation-discriminator-pk" value="true"/>
        <extension vendor-name="datanucleus" key="relation-discriminator-value" value="work"/>
      </field>
    <field name="homeAddresses" persistence-modifier="persistent" table="ACCOUNT_ADDRESSES">
      <collection element-type="com.mydomain.Address">
        <join column="ACCOUNT_ID_OID"/>
        <element column="ADDRESS_ID_EID"/>
        <extension vendor-name="datanucleus" key="relation-discriminator-column" value="ADDRESS_TYPE"/>
        <extension vendor-name="datanucleus" key="relation-discriminator-pk" value="true"/>
        <extension vendor-name="datanucleus" key="relation-discriminator-value" value="home"/>
      </field>
    </class>

    <class name="Address">
      <field name="id" primary-key="true">
        <column name="ADDRESS_ID"/>
      </field>
      ...
    </class>
  </package>
```

So we have defined the same join table for the 2 collections "ACCOUNT_ADDRESSES", and the same columns in the join table, meaning that we will be sharing the same join table to represent both relations. The important step is then to define the 3 DataNucleus extension tags. These define a column in the join table (the same for both relations), and the value that will be populated when a row of that collection is inserted into the join table. In our case, all "home" addresses will have a value of "home" inserted into this column, and all "work" addresses will have "work" inserted. This means we can now identify easily which join table entry represents which relation field.

This results in the following database schema
61.8 Shared Foreign Key

The relationships using foreign keys shown above rely on the foreign key relating to the relation in question. DataNucleus allows the possibility of sharing a foreign key between relations between the same classes. The example below demonstrates this. We take the example as shown above (1-N Unidirectional Foreign Key relation), and extend Account to have 2 collections of Address records. One for home addresses and one for work addresses, like this:

We now change the metadata we had earlier to allow for 2 collections, but sharing the join table.
So we have defined the same foreign key for the 2 collections "ACCOUNT_ID_OID". The important step is then to define the 2 DataNucleus extension tags. These define a column in the element table (the same for both relations), and the value that will be populated when a row of that collection is inserted into the element table. In our case, all "home" addresses will have a value of "home" inserted into this column, and all "work" addresses will have "work" inserted. This means we can now identify easily which element table entry represents which relation field.

This results in the following database schema

![Database schema diagram]

©2015, DataNucleus • ALL RIGHTS RESERVED.
62 Sets

62.1 JDO : 1-N Relationships with Sets

You have a 1-N (one to many) or N-1 (many to one) when you have one object of a class that has a Set of objects of another class. Please note that Sets do not allow duplicates, and so the persistence process reflects this with the choice of primary keys. There are two ways in which you can represent this in a datastore: Join Table (where a join table is used to provide the relationship mapping between the objects), and Foreign-Key (where a foreign key is placed in the table of the object contained in the Set).

The various possible relationships are described below.

- 1-N Unidirectional using Join Table
- 1-N Unidirectional using Foreign-Key
- 1-N Bidirectional using Join Table
- 1-N Bidirectional using Foreign-Key
- 1-N Unidirectional of non-PC using Join Table
- 1-N embedded elements using Join Table
- 1-N Serialised Set
- 1-N using shared join table
- 1-N using shared foreign key
- 1-N Bidirectional "Compound Identity" (owner object as part of PK in element)

This page is aimed at Set fields and so applies to fields of Java type java.util.HashSet, java.util.LinkedHashSet, java.util.Set, java.util.SortedSet, java.util.TreeSet

Please note that RDBMS supports the full range of options on this page, whereas other datastores (ODF, Excel, HBase, MongoDB, etc) persist the Set in a column in the owner object (as well as a column in the non-owner object when bidirectional) rather than using join-tables or foreign-keys since those concepts are RDBMS-only.

62.1.1 equals() and hashCode()

Important: The element of a Collection ought to define the methods equals and hashCode so that updates are detected correctly. This is because any Java Collection will use these to determine equality and whether an element is contained in the Collection. Note also that the hashCode() should be consistent throughout the lifetime of a persistable object. By that we mean that it should not use some basis before persistence and then use some other basis (such as the object identity) after persistence, for this reason we do not recommend usage of JDOHelper.getObjectId(obj) in the equals/hashCode methods.

62.2 1-N Set Unidirectional

We have 2 sample classes Account and Address. These are related in such a way as Account contains a Set of objects of type Address, yet each Address knows nothing about the Account objects that it relates to. Like this
There are 2 ways that we can persist this relationship. These are shown below

62.2.1 Using Join Table

If you define the XML metadata for these classes as follows

```xml
<package name="com.mydomain">
  <class name="Account">
    <field name="id" primary-key="true">
      <column name="ACCOUNT_ID"/>
    </field>
    ...
    <field name="addresses">
      <collection element-type="com.mydomain.Address"/>
      <join/>
    </field>
  </class>

  <class name="Address">
    <field name="id" primary-key="true">
      <column name="ADDRESS_ID"/>
    </field>
    ...
  </class>
</package>
```

or alternatively using annotations
public class Account {
    ...

    @Join
    Set<Address> addresses;
}

public class Address {
    ...
}

The crucial part is the join element on the field element - this signals to JDO to use a join table.

This will create 3 tables in the database, one for Address, one for Account, and a join table, as shown below.

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table attribute on the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the field element.
- To specify the name of the join table, specify the table attribute on the field element with the collection.
- To specify the names of the join table columns, use the column attribute of join, element elements.
- To specify the foreign-key between container table and join table, specify <foreign-key> below the <join> element.
- To specify the foreign-key between join table and element table, specify <foreign-key> below either the <field> element or the <element> element.
- If you wish to share the join table with another relation then use the DataNucleus "shared join table" extension
- The join table will, by default, be given a primary key. If you want to omit this then you can turn it off using the DataNucleus metadata extension "primary-key" (within <join>) set to false.
- If you want the set to include nulls, you can turn on this behaviour by adding the extension metadata "allow-nulls" to the <field> set to true
62.2.2 Using Foreign-Key

In this relationship, the Account class has a List of Address objects, yet the Address knows nothing about the Account. In this case we don’t have a field in the Address to link back to the Account and so DataNucleus has to use columns in the datastore representation of the Address class. So we define the XML metadata like this

```xml
<package name="com.mydomain">
  <class name="Account">
    <field name="id" primary-key="true">
      <column name="ACCOUNT_ID"/>
    </field>
    ...
    <field name="addresses">
      <collection element-type="com.mydomain.Address"/>
      <element column="ACCOUNT_ID"/>
    </field>
  </class>

  <class name="Address">
    <field name="id" primary-key="true">
      <column name="ADDRESS_ID"/>
    </field>
    ...
  </class>
</package>
```

or alternatively using annotations

```java
class Account {
    ...
    
    @Element(column="ACCOUNT_ID")
    Set<Address> addresses;
}

class Address {
    ...
}
```

Again there will be 2 tables, one for Address, and one for Account. Note that we have no "mapped-by" attribute specified, and also no "join" element. If you wish to specify the names of the columns used in the schema for the foreign key in the Address table you should use the element element within the field of the collection.
In terms of operation within your classes of assigning the objects in the relationship. You have to take your Account object and add the Address to the Account collection field since the Address knows nothing about the Account. Also be aware that each Address object can have only one owner, since it has a single foreign key to the Account. If you wish to have an Address assigned to multiple Accounts then you should use the "Join Table" relationship above.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the `table` attribute on the `class` element
- To specify the names of the columns where the fields of a class are stored, specify the `column` attribute on the `field` element.
- To specify the foreign-key between container table and element table, specify `<foreign-key>` below either the `<field>` element or the `<element>` element.

### 62.3 1-N Set Bidirectional

We have 2 sample classes `Account` and `Address`. These are related in such a way as `Account` contains a `Set` of objects of type `Address`, and each `Address` has a reference to the `Account` object that it relates to. Like this

There are 2 ways that we can persist this relationship. These are shown below

#### 62.3.1 Using Join Table

If you define the XML Metadata for these classes as follows
<package name="com.mydomain">
  <class name="Account">
    <field name="id" primary-key="true">
      <column name="ACCOUNT_ID"/>
    </field>
    ...
    <field name="addresses" mapped-by="account">
      <collection element-type="com.mydomain.Address"/>
    </field>
  </class>
  <class name="Address">
    <field name="id" primary-key="true">
      <column name="ADDRESS_ID"/>
    </field>
    ...
    <field name="account"/>
  </class>
</package>

or alternatively using annotations

public class Account
{
  ...
  @Persistent(mappedBy="account")
  @Join
  Set<Address> addresses;
}

public class Address
{
  ...
  Account account;
}

The crucial part is the join element on the field element - this signals to JDO to use a join table.

This will create 3 tables in the database, one for Address, one for Account, and a join table, as shown below.

©2015, DataNucleus • ALL RIGHTS RESERVED.
The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the `table` attribute on the `class` element.
- To specify the names of the columns where the fields of a class are stored, specify the `column` attribute on the `field` element.
- To specify the name of the join table, specify the `table` attribute on the `field` element with the `collection`.
- To specify the names of the join table columns, use the `column` attribute of `join, element` elements.
- To specify the foreign-key between container table and join table, specify `<foreign-key>` below the `<join>` element.
- To specify the foreign-key between join table and element table, specify `<foreign-key>` below either the `<field>` element or the `<element>` element.
- If you wish to share the join table with another relation then use the DataNucleus "shared join table" extension.
- The join table will, by default, be given a primary key. If you want to omit this then you can turn it off using the DataNucleus metadata extension "primary-key" (within `<join>`) set to false.
- When forming the relation please make sure that you set the relation at BOTH sides since DataNucleus would have no way of knowing which end is correct if you only set one end.
- If you want the set to include nulls, you can turn on this behaviour by adding the extension metadata "allow-nulls" to the `<field>` set to true.

**62.3.2 Using Foreign-Key**

Here we have the 2 classes with both knowing about the relationship with the other.

If you define the XML metadata for these classes as follows
or alternatively using annotations

```java
public class Account
{
    ...

    @Persistent(mappedBy="account")
    Set<Address> addresses;
}

public class Address
{
    ...

    Account account;
}
```

The crucial part is the \textit{mapped-by} attribute of the field on the "1" side of the relationship. This tells the JDO implementation to look for a field called \textit{account} on the Address class.

This will create 2 tables in the database, one for Address (including an \textit{ACCOUNT_ID} to link to the \textit{ACCOUNT} table), and one for Account. Notice the subtle difference to this set-up to that of the \textit{Join Table} relationship earlier.
If you wish to fully define the schema table and column names etc, follow these tips:

- To specify the name of the table where a class is stored, specify the `table` attribute on the `class` element.
- To specify the names of the columns where the fields of a class are stored, specify the `column` attribute on the `field` element.
- To specify the foreign-key between container table and element table, specify `<foreign-key>` below either the `<field>` element or the `<element>` element.
- When forming the relation please make sure that **you set the relation at BOTH sides** since DataNucleus would have no way of knowing which end is correct if you only set one end.

### 62.4 1-N Set of non-PersistenceCapable objects

All of the examples above show a 1-N relationship between 2 `PersistenceCapable` classes. DataNucleus can also cater for a Collection of primitive or Object types. For example, when you have a Collection of Strings. This will be persisted in the same way as the "Join Table" examples above. A join table is created to hold the collection elements. Let's take our example. We have an `Account` that stores a Collection of addresses. These addresses are simply Strings. We define the Meta-Data like this:

```xml
<package name="com.mydomain">
  <class name="Account">
    <field name="id" primary-key="true">
      <column name="ACCOUNT_ID"/>
    </field>
    ...
    <field name="addresses" persistence-modifier="persistent">
      <collection element-type="java.lang.String"/>
      <join/>
      <element column="ADDRESS"/>
    </field>
  </class>
</package>
```

or alternatively using annotations
public class Account
{
    ...

    @Join
    @Element(column="ADDRESS")
    Set<String> addresses;
}

In the datastore the following is created

![ACCOUNT and ACCOUNT_ADDRESSES tables]

The ACCOUNT table is as before, but this time we only have the "join table". In our MetaData we used the <element> tag to specify the column name to use for the actual address String.

Please note that the column ADPT_PK_IDX is added by DataNucleus when the column type of the element is not valid to be part of a primary key (with the RDBMS being used). If the column type of your element is acceptable for use as part of a primary key then you will not have this "ADPT_PK_IDX" column. You can override the default naming of this column by specifying the DataNucleus extension "adapter-column-name" within the <field> for the Collection.

62.5 Embedded into a Join Table

The above relationship types assume that both classes in the 1-N relation will have their own table. A variation on this is where you have a join table but you embed the elements of the collection into this join table. To do this you use the embedded-element attribute on the collection MetaData element. This is described in Embedded Collection Elements.

62.6 Serialised into a Join Table

The above relationship types assume that both classes in the 1-N relation will have their own table. A variation on this is where you have a join table but you serialise the elements of the collection into this join table in a single column. To do this you use the serialised-element attribute on the collection MetaData element. This is described in Serialised Collection Elements.
63 Lists

63.1 JDO : 1-N Relationships with Lists

You have a 1-N (one to many) or N-1 (many to one) when you have one object of a class that has a List of objects of another class. There are two ways in which you can represent this in a datastore. Join Table (where a join table is used to provide the relationship mapping between the objects), and Foreign-Key (where a foreign key is placed in the table of the object contained in the List).

The various possible relationships are described below.

- 1-N Unidirectional using Join Table
- 1-N Unidirectional using Foreign-Key
- 1-N Ordered List using Foreign-Key
- 1-N Bidirectional using Join Table
- 1-N Bidirectional using Foreign-Key
- 1-N Unidirectional of non-PC using Join Table
- 1-N embedded elements using Join Table
- 1-N Serialised List
- 1-N using shared join table
- 1-N using shared foreign key
- 1-N Bidirectional "Compound Identity" (owner object as part of PK in element)

This page is aimed at List fields and so applies to fields of Java type java.util.ArrayList, java.util.LinkedList, java.util.List, java.util.Stack, java.util.Vector

Please note that RDBMS supports the full range of options on this page, whereas other datastores (ODF, Excel, HBase, MongoDB, etc) persist the List in a column in the owner object (as well as a column in the non-owner object when bidirectional) rather than using join-tables or foreign-keys since those concepts are RDBMS-only.

63.1.1 equals() and hashCode()

Important : The element of a Collection ought to define the methods equals and hashCode so that updates are detected correctly. This is because any Java Collection will use these to determine equality and whether an element is contained in the Collection. Note also that the hashCode() should be consistent throughout the lifetime of a persistable object. By that we mean that it should not use some basis before persistence and then use some other basis (such as the object identity) after persistence, for this reason we do not recommend usage of JDOHelper.getObjectId(obj) in the equals/ hashCode methods.

63.2 1-N List Unidirectional

We have 2 sample classes Account and Address. These are related in such a way as Account contains a List of objects of type Address, yet each Address knows nothing about the Account objects that it relates to. Like this
There are 2 ways that we can persist this relationship. These are shown below

### 63.2.1 Using Join Table

If you define the XML metadata for these classes as follows

```xml
<package name="com.mydomain">
  <class name="Account" identity-type="datastore">
    ...
    <field name="addresses">
      <collection element-type="com.mydomain.Address"/>
      <join/>
    </field>
  </class>
  <class name="Address" identity-type="datastore">
    ...
  </class>
</package>
```

or alternatively using annotations

```java
public class Account {
  ...
  @Join
  List<Address> addresses;
}

public class Address {
  ...
}
```

The crucial part is the `join` element on the field element - this signals to JDO to use a join table.

There will be 3 tables, one for Address, one for Account, and the join table. The difference from Set is in the contents of the join table. An index column (INTEGER_IDX) is added to keep track of the position of objects in the List. The name of this column can be controlled using the `<order>` MetaData element.
The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips:

- To specify the name of the table where a class is stored, specify the `table` attribute on the `class` element.
- To specify the names of the columns where the fields of a class are stored, specify the `column` attribute on the `field` element.
- To specify the name of the join table, specify the `table` attribute on the `field` element with the `collection` attribute.
- To specify the names of the join table columns, use the `column` attribute of `join`, `element` and `order` elements.
- To specify the foreign-key between container table and join table, specify `<foreign-key>` below the `<join>` element.
- To specify the foreign-key between join table and element table, specify `<foreign-key>` below either the `<field>` element or the `<element>` element.
- If you wish to share the join table with another relation then use the DataNucleus "shared join table" extension
- The join table will, by default, be given a primary key. If you want to omit this then you can turn it off using the DataNucleus metadata extension "primary-key" (within `<join>`) set to false.
- The column "ADPT_PK_IDX" is added by DataNucleus so that duplicates can be stored. You can control this by adding an `<order>` element and specifying the column name for the order column, or you can override the default naming of this column by specifying the DataNucleus extension "adapter-column-name" (within `<field>`).
- If you want the set to include nulls, you can turn on this behaviour by adding the extension metadata "allow-nulls" to the `<field>` set to true.

### 63.2.2 Using Foreign-Key

In this relationship, the Account class has a List of Address objects, yet the Address knows nothing about the Account. In this case we don't have a field in the Address to link back to the Account and so DataNucleus has to use columns in the datastore representation of the Address class. So we define the XML metadata like this:
or alternatively using annotations

```java
public class Account
{
    ...

    @Element(column="ACCOUNT_ID")
    List<Address> addresses;
}

public class Address
{
    ...
}
```

Again there will be 2 tables, one for Address, and one for Account. Note that we have no "mapped-by" attribute specified, and also no "join" element. If you wish to specify the names of the columns used in the schema for the foreign key in the Address table you should use the `element` element within the field of the collection.

In terms of operation within your classes of assigning the objects in the relationship. With DataNucleus and List-based containers you have to take your Account object and add the Address to the Account collection field since the Address knows nothing about the Account.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the `table` attribute on the `class` element
• To specify the names of the columns where the fields of a class are stored, specify the column attribute on the field element.
• To specify the foreign-key between container table and element table, specify <foreign-key> below either the <field> element or the <element> element.

Limitations
• Since each Address object can have at most one owner (due to the "Foreign Key") this mode of persistence will not allow duplicate values in the List. If you want to allow duplicate List entries, then use the "Join Table" variant above.

63.2.3 1-N Ordered List using Foreign-Key

This is the same as the case above except that we don’t want an indexing column adding to the element and instead we define an "ordering" criteria. This is a DataNucleus extension to JDO. So we define the XML metadata like this

```xml
<package name="com.mydomain">
  <class name="Account" identity-type="datastore">
    ...
    <field name="addresses">
      <collection element-type="com.mydomain.Address"/>
      <order>
        <extension vendor-name="datanucleus" key="list-ordering" value="city ASC"/>
      </order>
    </field>
  </class>
  <class name="Address" identity-type="datastore">
    ...
  </class>
</package>
```

or alternatively using annotations

```java
public class Account {
  ...
  @Order(extensions=Extension(vendorName="datanucleus", key="list-ordering", value="city ASC"))
  List<Address> addresses;
}
public class Address {
  ...
}
```

As above there will be 2 tables, one for Address, and one for Account. We have no indexing column, but instead we will order the elements using the "city" field in ascending order.
In terms of operation within your classes of assigning the objects in the relationship. With DataNucleus and List-based containers you have to take your Account object and add the Address to the Account collection field since the Address knows nothing about the Account.

**Limitations**

- Ordered lists are only ordered in the defined way when retrieved from the datastore.

### 63.3 1-N List Bidirectional

We have 2 sample classes Account and Address. These are related in such a way as Account contains a List of objects of type Address, and each Address has a reference to the Account object that it relates to. Like this

There are 2 ways that we can persist this relationship. These are shown below

#### 63.3.1 Using Join Table

If you define the XML metadata for these classes as follows

```xml
<package name="com.mydomain">
  <class name="Account" identity-type="datastore">
    ...
    <field name="addresses" mapped-by="account">
      <collection element-type="com.mydomain.Address"/>
    </field>
  </class>
  ...
  <class name="Address" identity-type="datastore">
    ...
    <field name="account"/>
  </class>
</package>
```

or alternatively using annotations
public class Account
{
    ...
    @Persistent(mappedBy="account")
    @Join
    List<Address> addresses;
}

public class Address
{
    ...
    Account account;
}

The crucial part is the join element on the field element - this signals to JDO to use a join table.

There will be 3 tables, one for Address, one for Account, and the join table. The difference from Set is in the contents of the join table. An index column (INTEGER_IDX) is added to keep track of the position of objects in the List. The name of this column can be controlled using the <order> MetaData element.

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

• To specify the name of the table where a class is stored, specify the table attribute on the class element
• To specify the names of the columns where the fields of a class are stored, specify the column attribute on the field element.
• To specify the name of the join table, specify the table attribute on the field element with the collection.
• To specify the names of the join table columns, use the column attribute of join, element and order elements.
• To specify the foreign-key between container table and join table, specify <foreign-key> below the <join> element.
• To specify the foreign-key between join table and element table, specify <foreign-key> below either the <field> element or the <element> element.
• If you wish to share the join table with another relation then use the DataNucleus "shared join table" extension
• The join table will, by default, be given a primary key. If you want to omit this then you can turn it off using the DataNucleus metadata extension "primary-key" (within <join>) set to false.
• The column "ADPT_PK_IDX" is added by DataNucleus so that duplicates can be stored. You can control this by adding an <order> element and specifying the column name for the order column, or you can override the default naming of this column by specifying the DataNucleus extension "adapter-column-name" (within <field>).
• When forming the relation please make sure that you set the relation at BOTH sides since DataNucleus would have no way of knowing which end is correct if you only set one end.
• If you want the set to include nulls, you can turn on this behaviour by adding the extension metadata "allow-nulls" to the <field> set to true.

63.3.2 Using Foreign-Key

Here we have the 2 classes with both knowing about the relationship with the other.

Please note that an Foreign-Key List will NOT, by default, allow duplicates. This is because it stores the element position in the element table. If you need a List with duplicates we recommend that you use the Join Table List implementation above. If you have an application identity element class then you could in principle add the element position to the primary key to allow duplicates, but this would imply changing your element class identity.

If you define the Meta-Data for these classes as follows

```xml
<package name="com.mydomain">
  <class name="Account" identity-type="datastore">
    ...
    <field name="addresses" mapped-by="account">
      <collection element-type="com.mydomain.Address"/>
    </field>
  </class>
</package>
```

or alternatively using annotations
```java
public class Account {
    ...
    @Persistent(mappedBy="account")
    List<Address> addresses;
}
public class Address {
    ...
    Account account;
}
```

The crucial part is the `mapped-by` attribute of the field on the "1" side of the relationship. This tells the JDO implementation to look for a field called `account` on the Address class.

Again there will be 2 tables, one for `Address`, and one for `Account`. The difference from the Set example is that the List index is placed in the table for `Address` whereas for a Set this is not needed.

In terms of operation within your classes of assigning the objects in the relationship. With DataNucleus and List-based containers you have to take your Account object and add the Address to the Account collection field (you can't just take the Address object and set its Account field since the position of the Address in the List needs setting, and this is done by adding the Address to the Account). In addition, if you are removing an object from a List, you cannot simply set the owner on the element to "null". You have to remove it from the List end of the relationship.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the `table` attribute on the `class` element.
- To specify the names of the columns where the fields of a class are stored, specify the `column` attribute on the `field` element.
- To specify the foreign-key between container table and element table, specify `<foreign-key>` below either the `<field>` element or the `<element>` element.
- When forming the relation please make sure that you set the relation at BOTH sides since DataNucleus would have no way of knowing which end is correct if you only set one end.

**Limitation**: Since each Address object can have at most one owner (due to the "Foreign Key") this mode of persistence will not allow duplicate values in the List. If you want to allow duplicate List entries, then use the "Join Table" variant above.
63.4 1-N List of non-PersistenceCapable objects

All of the examples above show a 1-N relationship between 2 PersistenceCapable classes. DataNucleus can also cater for a List of primitive or Object types. For example, when you have a List of Strings. This will be persisted in the same way as the "Join Table" examples above. A join table is created to hold the list elements. Let's take our example. We have an Account that stores a List of addresses. These addresses are simply Strings. We define the XML metadata like this

```xml
<package name="com.mydomain">
  <class name="Account" identity-type="datastore">
    ...
    <field name="addresses" persistence-modifier="persistent">
      <collection element-type="java.lang.String"/>
      <join/>
      <element column="ADDRESS"/>
    </field>
  </class>
</package>
```

or alternatively using annotations

```java
public class Account
{
    ...

    @Join
    @Element(column="ADDRESS")
    List<String> addresses;
}
```

In the datastore the following is created

The ACCOUNT table is as before, but this time we only have the "join table". In our MetaData we used the <element> tag to specify the column name to use for the actual address String. In addition we have an additional index column to form part of the primary key (along with the FK back to the ACCOUNT table). You can override the default naming of this column by specifying the <order> tag.

63.5 Embedded into a Join Table

The above relationship types assume that both classes in the 1-N relation will have their own table. A variation on this is where you have a join table but you embed the elements of the collection into this join table. To do this you use the embedded-element attribute on the collection MetaData element. This is described in Embedded Collection Elements.
63.6 Serialised into a Join Table

The above relationship types assume that both classes in the 1-N relation will have their own table. A variation on this is where you have a join table but you serialise the elements of the collection into this join table in a single column. To do this you use the `serialised-element` attribute on the `collection` MetaData element. This is described in Serialised Collection Elements.
64 Maps

64.1 JDO : 1-N Relationships with Maps

You have a 1-N (one to many) or N-1 (many to one) when you have one object of a class that has a Map of objects of another class. There are two general ways in which you can represent this in a datastore. **Join Table** (where a join table is used to provide the relationship mapping between the objects), and **Foreign-Key** (where a foreign key is placed in the table of the object contained in the Map.

The various possible relationships are described below.

- Map[PC, PC] using join table
- Map[Simple, PC] using join table
- Map[PC, Simple] using join table
- Map[Simple, Simple] using join table
- 1-N Bidirectional using Foreign-Key (key stored in the value class)
- 1-N Unidirectional using Foreign-Key (key stored in the value class)
- 1-N Unidirectional using Foreign-Key (value stored in the key class)
- 1-N embedded keys/values using Join Table
- 1-N Serialised map
- 1-N Bidirectional "Compound Identity" (owner object as part of PK in value)


Please note that RDBMS supports the full range of options on this page, whereas other datastores (ODF, Excel, HBase, MongoDB, etc) persist the Map in a column in the owner object rather than using join-tables or foreign-keys since those concepts are RDBMS-only.

64.2 1-N Map using Join Table

We have a class **Account** that contains a Map. With a Map we store values using keys. As a result we have 3 main combinations of key and value, bearing in mind whether the key or value is **PersistenceCapable**.

64.2.1 Map[PC, PC]

Here both the keys and the values are **PersistenceCapable**. Like this
If you define the Meta-Data for these classes as follows

```xml
<package name="com.mydomain">
  <class name="Account" identity-type="datastore">
    ...
    <field name="addresses" persistence-modifier="persistent">
      <map key-type="com.mydomain.Name" value-type="com.mydomain.Address"/>
      <join/>
    </field>
  </class>
  <class name="Address" identity-type="datastore">
    ...
  </class>
  <class name="Name" identity-type="datastore">
    ...
  </class>
</package>
```

This will create 4 tables in the datastore, one for Account, one for Address, one for Name and a join table containing foreign keys to the key/value tables.

If you want to configure the names of the columns in the "join" table you would use the <key> and <value> subelements of <field>, something like this

```xml
<package name="com.mydomain">
  <class name="Account" identity-type="datastore">
    ...
    <field name="addresses" persistence-modifier="persistent">
      <map key-type="com.mydomain.Name" value-type="com.mydomain.Address"/>
      <join/>
      <key>
        <value/>
      </key>
    </field>
  </class>
  <class name="Address" identity-type="datastore">
    ...
  </class>
  <class name="Name" identity-type="datastore">
    ...
  </class>
</package>
```
If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the `table` attribute on the `class` element.
- To specify the names of the columns where the fields of a class are stored, specify the `column` attribute on the `field` element.
- To specify the name of the join table, specify the `table` attribute on the `field` element.
- To specify the names of the columns of the join table, specify the `column` attribute on the `join`, `key`, and `value` elements.
- To specify the foreign-key between container table and join table, specify `<foreign-key>` below the `join` element.
- To specify the foreign-key between join table and key table, specify `<foreign-key>` below the `key` element.
- To specify the foreign-key between join table and value table, specify `<foreign-key>` below the `value` element.

Which changes the names of the join table to `ACCOUNT_ADDRESS` from `ACCOUNT_ADDRESSES` and the names of the columns in the join table from `ACCOUNT_ID_OID` to `ACCOUNT_ID`, from `NAME_ID_KID` to `NAME_ID`, and from `ADDRESS_ID_VID` to `ADDRESS_ID`.

### 64.2.2 Map[Simple, PC]

Here our key is a simple type (in this case a String) and the values are `PersistenceCapable`. Like this

![uml_diagram](https://via.placeholder.com/150)

If you define the Meta-Data for these classes as follows
This will create 3 tables in the datastore, one for Account, one for Address and a join table also containing the key.

If you want to configure the names of the columns in the "join" table you would use the <key> and <value> subelements of <field> as shown above.

Please note that the column ADPT_PK_IDX is added by DataNucleus when the column type of the key is not valid to be part of a primary key (with the RDBMS being used). If the column type of your key is acceptable for use as part of a primary key then you will not have this "ADPT_PK_IDX" column.

64.2.3 Map[PC, Simple]
This operates exactly the same as "Map[Simple, PC]" except that the additional table is for the key instead of the value.

64.2.4 Map[Simple, Simple]
Here our keys and values are of simple types (in this case a String). Like this
If you define the Meta-Data for these classes as follows:

```xml
<package name="com.mydomain">
  <class name="Account" identity-type="datastore">
    ...
    <field name="addresses" persistence-modifier="persistent">
      <map key-type="java.lang.String" value-type="java.lang.String"/>
      <join/>
    </field>
  </class>
</package>
```

This results in just 2 tables. The "join" table contains both the key AND the value.

If you want to configure the names of the columns in the "join" table you would use the `<key>` and `<value>` subelements of `<field>` as shown above.

Please note that the column ADPT_PK_IDX is added by DataNucleus when the column type of the key is not valid to be part of a primary key (with the RDBMS being used). If the column type of your key is acceptable for use as part of a primary key then you will not have this "ADPT_PK_IDX" column.

**64.2.5 Embedded**

The above relationship types assume that all PersistenceCapable classes in the 1-N relation will have their own table. A variation on this is where you have a join table but you embed the keys, the values, or the keys and the values of the map into this join table. This is described in Embedded Maps.

**64.3 1-N Map using Foreign-Key**
64.3.1 1-N Foreign-Key Bidirectional (key stored in value)

In this case we have an object with a Map of objects and we’re associating the objects using a foreign-key in the table of the value.

With these classes we want to store a foreign-key in the value table (ADDRESS), and we want to use the “alias” field in the Address class as the key to the map. If you define the Meta-Data for these classes as follows

```
<package name="com.mydomain">
  <class name="Account" identity-type="datastore">
    ...
    <field name="addresses" persistence-modifier="persistent" mapped-by="account">
      <map key-type="java.lang.String" value-type="com.mydomain.Address"/>
      <key mapped-by="alias"/>
    </field>
  </class>
  <class name="Address" identity-type="datastore">
    ...
    <field name="account" persistence-modifier="persistent"/>
    <field name="alias" null-value="exception">
      <column name="KEY" length="20" jdbc-type="VARCHAR"/>
    </field>
  </class>
</package>
```

This will create 2 tables in the datastore. One for Account, and one for Address. The table for Address will contain the key field as well as an index to the Account record (notated by the `mapped-by` tag).
64.3.2 1-N Foreign-Key Unidirectional (key stored in value)

In this case we have an object with a Map of objects and we're associating the objects using a foreign-key in the table of the value. As in the case of the bidirectional relation above we're using a field (alias) in the Address class as the key of the map.

In this relationship, the Account class has a Map of Address objects, yet the Address knows nothing about the Account. In this case we don't have a field in the Address to link back to the Account and so DataNucleus has to use columns in the datastore representation of the Address class. So we define the MetaData like this:

```
<package name="com.mydomain">
  <class name="Account" identity-type="datastore">
    ...
    <field name="addresses" persistence-modifier="persistent">
      <map key-type="java.lang.String" value-type="com.mydomain.Address"/>
      <key mapped-by="alias"/>
      <value column="ACCOUNT_ID_OID"/>
    </field>
  </class>
</package>
```

Again there will be 2 tables, one for Address, and one for Account. Note that we have no "mapped-by" attribute specified on the "field" element, and also no "join" element. If you wish to specify the names of the columns used in the schema for the foreign key in the Address table you should use the value element within the field of the map.

In terms of operation within your classes of assigning the objects in the relationship. You have to take your Account object and add the Address to the Account map field since the Address knows nothing
about the Account. Also be aware that each Address object can have only one owner, since it has a single foreign key to the Account. If you wish to have an Address assigned to multiple Accounts then you should use the "Join Table" relationship above.

64.3.3 1-N Foreign-Key Unidirectional (value stored in key)

In this case we have an object with a Map of objects and we're associating the objects using a foreign-key in the table of the key. We're using a field (businessAddress) in the Address class as the value of the map.

![Diagram](image)

In this relationship, the Account class has a Map of Address objects, yet the Address knows nothing about the Account. We don't have a field in the Address to link back to the Account and so DataNucleus has to use columns in the datastore representation of the Address class. So we define the MetaData like this

```xml
<package name="com.mydomain">
  <class name="Account" identity-type="datastore">
    ...
    <field name="phoneNumbers" persistence-modifier="persistent">
      <map key-type="com.mydomain.Address" value-type="java.lang.String"/>
      <key column="ACCOUNT_ID_OID"/>
      <value mapped-by="businessPhoneNumber"/>
    </field>
  </class>
  <class name="Address" identity-type="datastore">
    ...
    <field name="businessPhoneNumber" null-value="exception">
      <column name="BUS_PHONE" length="20" jdbc-type="VARCHAR"/>
    </field>
  </class>
</package>
```

There will be 2 tables, one for Address, and one for Account. The key thing here is that we have specified a "mapped-by" on the "value" element. Note that we have no "mapped-by" attribute specified on the "field" element, and also no "join" element. If you wish to specify the names of the columns used in the schema for the foreign key in the Address table you should use the key element within the field of the map.
In terms of operation within your classes of assigning the objects in the relationship. You have to take your Account object and add the Address to the Account map field since the Address knows nothing about the Account. Also be aware that each Address object can have only one owner, since it has a single foreign key to the Account. If you wish to have an Address assigned to multiple Accounts then you should use the "Join Table" relationship above.
65 N-to-1 Relations

65.1 JDO : N-1 Relationships
You have a N-to-1 relationship when an object of a class has an associated object of another class (only one associated object) and several of this type of object can be linked to the same associated object. From the other end of the relationship it is effectively a 1-N, but from the point of view of the object in question, it is N-1. You can create the relationship in 2 ways depending on whether the 2 classes know about each other (bidirectional), or whether only the "N" side knows about the other class (unidirectional). These are described below.

65.1.1 Unidirectional (Join Table)
For this case you could have 2 classes, User and Account, as below.

```
<package name="mydomain">
  <class name="User" identity-type="datastore">
    ...
  </class>
  <class name="Account" identity-type="datastore">
    ...
    <field name="user" persistence-modifier="persistent">
      <join/>
    </field>
  </class>
</package>
```

so the Account class ("N" side) knows about the User class ("1" side), but not vice-versa and the relation is stored using a join table. A particular user could be related to several accounts. If you define the XML metadata for these classes as follows

```
<package name="mydomain">
  <class name="User" identity-type="datastore">
    ...
  </class>
  <class name="Account" identity-type="datastore">
    ...
    <field name="user" persistence-modifier="persistent">
      <join/>
    </field>
  </class>
</package>
```

alternatively using annotations
public class Account
{
    ...

    @Join(table="ACCOUNT_USER")
    User user;
}

For RDBMS this will create 3 tables in the database, one for User (with name USER), one for Account (with name ACCOUNT), and a join table (with name ACCOUNT_USER) as shown below.

Note that in the case of non-RDBMS datastores there is no join-table, simply a "column" in the ACCOUNT "table", storing the "id" of the related object

Things to note :-

• If you wish to specify the names of the database tables and columns for these classes, you can use the attribute table (on the class element), the attribute name (on the column element) and the attribute name (on the column attribute under join)

65.1.2 Unidirectional (ForeignKey)

Here you have the same two classes as above but you have a foreign-key in the table of Account. For this case, just look at the 1-1 Unidirectional documentation since it is identical.

65.1.3 Bidirectional

This relationship is described in the guide for 1-N relationships. In particular there are 2 ways to define the relationship with RDBMS; the first uses a Join Table to hold the relationship, whilst the second uses a Foreign Key in the "N" object to hold the relationship. For non-RDBMS datastores each side will have a "column" (or equivalent) in the "table" of the N side storing the "id" of the related (owning) object. Please refer to the 1-N relationships bidirectional relations since they show this exact relationship.
66 M-to-N Relations

66.1 JDO : M-N Relationships

You have a M-to-N (or Many-to-Many) relationship if an object of a class A has associated objects of class B, and class B has associated objects of class A. This relationship may be achieved through Java Set, Map, List or subclasses of these, although the only one that supports a true M-N is for a Set/Collection.

With DataNucleus this can be set up as described in this section, using what is called a Join Table relationship. Let's take the following example and describe how to model it with the different types of collection classes. We have 2 classes, Product and Supplier as below.

Here the Product class knows about the Supplier class. In addition the Supplier knows about the Product class, however with DataNucleus (as with the majority of JDO implementations) these relationships are independent.

Please note that RDBMS supports the full range of options on this page, whereas other datastores (ODF, Excel, HBase, MongoDB, etc) persist the Collection in a column in the owner object and a column in the non-owner object rather than using join-tables since that concept is RDBMS-only.

The various possible relationships are described below.

- M-N Set relation
- M-N Ordered List relation
- M-N Indexed List - modelled as 2 1-N Unidirectional relations using Join Table
- M-N Map - modelled as 2 1-N Unidirectional using Join Table

66.2 Using Set

If you define the XML metadata for these classes as follows
<package name="mydomain">
  <class name="Product" identity-type="datastore">
    ...
    <field name="suppliers" table="PRODUCTS_SUPPLIERS">
      <collection element-type="mydomain.Supplier"/>
      <join>
        <column name="PRODUCT_ID"/>
      </join>
      <element>
        <column name="SUPPLIER_ID"/>
      </element>
    </field>
  </class>

  <class name="Supplier" identity-type="datastore">
    ...
    <field name="products" mapped-by="suppliers">
      <collection element-type="mydomain.Product"/>
    </field>
  </class>
</package>

alternatively using annotations

public class Product
{
  ...
  @Persistent(table="PRODUCTS_SUPPLIERS")
  @Join(column="PRODUCT_ID")
  @Element(column="SUPPLIER_ID")
  Set<Supplier> suppliers;
}

public class Supplier
{
  ...
  @Persistent(mappedBy="suppliers")
  Set<Products> products;
}

Note how we have specified the information only once regarding join table name, and join column names as well as the <join>. This is the JDO standard way of specification, and results in a single join table.
See also:

- M-N Worked Example
- M-N with Attributes Worked Example

### 66.3 Using Ordered Lists

If you define the Meta-Data for these classes as follows

```xml
<package name="mydomain">
  <class name="Product" identity-type="datastore">
    ...
    <field name="suppliers">
      <collection element-type="mydomain.Supplier"/>
      <order>
        <extension vendor-name="datanucleus" key="list-ordering" value="id ASC"/>
      </order>
      <join/>
    </field>
  </class>

  <class name="Supplier" identity-type="datastore">
    ...
    <field name="products">
      <collection element-type="mydomain.Product"/>
      <order>
        <extension vendor-name="datanucleus" key="list-ordering" value="id ASC"/>
      </order>
      <join/>
    </field>
  </class>
</package>
```

or using annotations
public class Product
{
    ...
    @Persistent(table="PRODUCTS_SUPPLIERS")
    @Join(column="PRODUCT_ID")
    @Element(column="SUPPLIER_ID")
    @Order(extensions=@Extension(vendorName="datanucleus", key="list-ordering", value="id ASC"))
    List<Supplier> suppliers
}

public class Supplier
{
    ...
    @Persistent
    @Order(extensions=@Extension(vendorName="datanucleus", key="list-ordering", value="id ASC"))
    List<Product> products
}

There will be 3 tables, one for Product, one for Supplier, and the join table. The difference from the Set example is that we now have <order-by> at both sides of the relation. This has no effect in the datastore schema but when the Lists are retrieved they are ordered using the specified order-by.

66.4 Using indexed Lists

Firstly a true M-N relation with Lists is impossible since there are two lists, and it is undefined as to which one applies to which side etc. What is shown below is two independent 1-N unidirectional join table relations. If you define the Meta-Data for these classes as follows
<package name="mydomain">
  <class name="Product" identity-type="datastore">
    ...
    <field name="suppliers" persistence-modifier="persistent">
      <collection element-type="mydomain.Supplier"/>
      <join/>
    </field>
  </class>

  <class name="Supplier" identity-type="datastore">
    ...
    <field name="products" persistence-modifier="persistent">
      <collection element-type="mydomain.Product"/>
      <join/>
    </field>
  </class>
</package>

Alternatively using annotations

```java
public class Product {
  ...
  @Join
  List<Supplier> suppliers;
}

g public class Supplier {
  ...
  @Join
  List<Products> products;
}
```

There will be 4 tables, one for Product, one for Supplier, and the join tables. The difference from the Set example is in the contents of the join tables. An index column is added to keep track of the position of objects in the Lists.
In the case of a List at both ends it doesn't make sense to use a single join table because the ordering can only be defined at one side, so you have to have 2 join tables.

66.5 Using Map

If you define the Meta-Data for these classes as follows

```xml
<package name="mydomain">
    <class name="Product" identity-type="datastore">
        ...
        <field name="suppliers" persistence-modifier="persistent">
            <map key-type="java.lang.String" value-type="mydomain.Supplier"/>
            <join/>
        </field>
    </class>
    <class name="Supplier" identity-type="datastore">
        ...
        <field name="products" persistence-modifier="persistent">
            <map key-type="java.lang.String" value-type="mydomain.Product"/>
            <join/>
        </field>
    </class>
</package>
```

This will create 4 tables in the datastore, one for Product, one for Supplier, and the join tables which also contains the keys to the Maps (a String).
66.6 Relationship Behaviour

Please be aware of the following.

- To add an object to an M-N relationship you need to set it at both ends of the relation since the relation is bidirectional and without such information the JDO implementation won’t know which end of the relation is correct.
- If you want to delete an object from one end of a M-N relationship you will have to remove it first from the other object’s relationship. If you don’t you will get an error message that the object to be deleted has links to other objects and so cannot be deleted.
67 Managing Relationships

67.1 JDO: Managing Relationships

The power of a Java persistence solution like DataNucleus is demonstrated when persisting relationships between objects. There are many types of relationships.

- **1-1 relationships** - this is where you have an object A relates to a second object B. The relation can be unidirectional where A knows about B, but B doesn't know about A. The relation can be bidirectional where A knows about B and B knows about A.

- **1-N relationships** - this is where you have an object A that has a collection of other objects of type B. The relation can be unidirectional where A knows about the objects B but the Bs don't know about A. The relation can be bidirectional where A knows about the objects B and the Bs know about A.

- **N-1 relationships** - this is where you have an object B1 that relates to an object A, and an object B2 that relates to A also etc. The relation can be unidirectional where the A doesn't know about the Bs. The relation can be bidirectional where the A has a collection of the Bs. [i.e. a 1-N relationship but from the point of view of the element]

- **M-N relationships** - this is where you have objects of type A that have a collection of objects of type B and the objects of type B also have a collection of objects of type A. The relation is always bidirectional by definition

- **Compound Identity relationships** when you have a relation and part of the primary key of the related object is the other persistent object. This is only available in JDO

67.1.1 Assigning Relationships

When the relation is unidirectional you simply set the related field to refer to the other object. For example we have classes A and B and the class A has a field of type B. So we set it like this

```java
A a = new A();
B b = new B();
a.setB(b); // "a" knows about "b"
```

When the relation is bidirectional you **have to set both sides** of the relation. For example, we have classes A and B and the class A has a collection of elements of type B, and B has a field of type A. So we set it like this

```java
A a = new A();
B b1 = new B();
a.addElement(b1); // "a" knows about "b1"
b1.setA(a); // "b1" knows about "a"
```

So it is really simple, with only 1 general rule. **With a bidirectional relation you should set both sides of the relation**

67.1.2 Managed Relationships

As previously mentioned, users should really set both sides of a bidirectional relation. DataNucleus provides a good level of managed relations in that it will attempt to correct any missing information in relations to make both sides consistent. This is defined below
For a 1-1 bidirectional relation, at persist you should set one side of the relation and the other side will be set to make it consistent. If the respective sides are set to inconsistent objects then an exception will be thrown at persist. At update of owner/non-owner side the other side will also be updated to make them consistent.

For a 1-N bidirectional relation and you only specify the element owner then the collection must be Set-based since DataNucleus cannot generate indexing information for you in that situation (you must position the elements). At update of element or owner the other side will also be updated to make them consistent. At delete of element the owner collection will also be updated to make them consistent. **If you are using a List you MUST set both sides of the relation.**

For an M-N bidirectional relation, at persist you MUST set the one side and the other side will be populated at commit/flush to make them consistent.

This management of relations can be turned on/off using a PMF property `datanucleus.manageRelationships`. If you always set both sides of a relation at persist or update then you could safely turn it off.

When performing management of relations there are some checks implemented to spot typical errors in user operations e.g add an element to a collection and then remove it (why?!). You can disable these checks using `datanucleus.manageRelationshipsChecks`, set to false.
68 Cascading

68.1 JDO : Cascading Operations

When defining your objects to be persisted and the relationships between them, it is often required to define dependencies between these related objects. What should happen when persisting an object and it relates to another object? What should happen to a related object when an object is deleted? You can define what happens with JDO and with DataNucleus. Let's take an example

```java
public class Owner {
    private DrivingLicense license;
    private Collection cars;
    ...
}

public class DrivingLicense {
    private String serialNumber;
    ...
}

public class Car {
    private String registrationNumber;
    private Owner owner;
    ...
}
```

So we have an Owner of a collection of vintage Car's (1-N), and the Owner has a DrivingLicense (1-1). We want to define lifecycle dependencies to match the relationships that we have between these objects. Firstly let's look at the basic Meta-Data for the objects.
68.1.1 Persistence

JDO2 defines a concept called **persistence-by-reachability**. This means that when you persist an object and it has a related persistable object then this other object is also persisted. So using our example if we do

```java
Owner bob = new Owner("Bob Smith");
DrivingLicense license = new DrivingLicense("011234BX4J");
bob.setLicense(license);
pm.makePersistent(bob); // "bob" knows about "license"
```

This results in both the **Owner** and the **DrivingLicense** objects being made persistent since the **Owner** is passed to the PM operation and it has a field referring to the unpersisted **DrivingLicense** object. So "reachability" will persist the license.

With DataNucleus you can actually turn off **persistence-by-reachability** for particular fields, by specifying in the MetaData a DataNucleus extension tag, as follows

```xml
<class name="Owner">
  <field name="license" persistence-modifier="persistent">
    <extension vendor-name="datanucleus" key="cascade-persist" value="false"/>
  </field>
  ...
</class>
```
So with this specification when we call `makePersistent()` with an object of type `Owner` then the field "license" will not be persisted at that time.

### 68.1.2 Update

As mentioned above JDO2 defines a concept called **persistence-by-reachability**. This applies not just to persist but also to update of objects, so when you update an object and its updated field has a persistable object then that will be persisted. So using our example if we do

```java
Owner bob = (Owner)pm.getObjectById(id);
DrivingLicense license2 = new DrivingLicense("233424BX4J");
bob.setLicense(license2);  // "bob" knows about "license2"
```

So when this field is updated the new `DrivingLicense` object will be made persistent since it is reachable from the persistent `Owner` object.

With DataNucleus you can actually turn off **update-by-reachability** for particular fields, by specifying in the MetaData a DataNucleus extension tag, as follows

```xml
<class name="Owner">
    <field name="license" persistence-modifier="persistent">
        <extension vendor-name="datanucleus" key="cascade-update" value="false"/>
    </field>
    ...
</class>
```

So with this specification when we call `makePersistent()` to update an object of type `Owner` then the field "license" will not be updated at that time.

### 68.1.3 Deletion, using Dependent Field

So we have an inverse 1-N relationship (no join table) between our `Owner` and his precious `Car's`, and a 1-1 relationship between the `Owner` and his `DrivingLicense`, because without his license he wouldn't be able to drive the cars :-0. What will happen to the `license` and the `cars` when the `owner` dies? Well in this particular case we want to define that the when the `owner` is deleted, then his `license` will also be deleted (since it is for him only), but that his `cars` will continue to exist, because his daughter will inherit them. In JDO2 this is called **Dependent Fields**. To utilise this concept to achieve our end goal we change the Meta-Data to be
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "file:/javax/jdo/jdo.dtd">
<jdo>
  <package name="com.mydomain.samples.cars">
    <class name="Owner">
      <field name="license" persistence-modifier="persistent" dependent="true"/>
      <field name="cars">
        <collection element-type="com.mydomain.samples.cars.Car" mapped-by="owner" dependent-element="false"/>
      </field>
    </class>
    <class name="DrivingLicense">
      <field name="serialNumber"/>
    </class>
    <class name="Car">
      <field name="registrationNumber"/>
      <field name="owner" persistence-modifier="persistent" dependent="false"/>
    </class>
  </package>
</jdo>

So it was as simple as just adding dependent and dependent-element attributes to our related fields. Notice that we also added one to the other end of the Owner-Car relationship, so that when a Car comes to the end of its life, the Owner will not die with it. It may be the case that the owner dies driving the car and they both die at the same time, but their deaths are independent!!

Just as we made use of dependent-element for collection fields, we also can make use of dependent-key and dependent-value for map fields, and dependent-element for array fields.

**Dependent Fields** is utilised in the following situations

- An object is deleted (using deletePersistent()) and that object has relations to other objects. If the other objects (either 1-1, 1-N, or M-N) are dependent then they are also deleted.
- An object has a 1-1 relation with another object, but the other object relation is nulled out. If the other object is dependent then it is deleted when the relation is nulled.
- An object has a 1-N collection relation with other objects and the element is removed from the collection. If the element is dependent then it will be deleted when removed from the collection. The same happens when the collections is cleared.
- An object has a 1-N map relation with other objects and the key is removed from the map. If the key or value are dependent and they are not present in the map more than once they will be deleted when they are removed. The same happens when the map is cleared.

**68.1.4 Deletion, using Foreign Keys (RDBMS)**

With JDO2 you can use “dependent-field” as shown above. As an alternative, when using RDBMS, you can use the datastore-defined foreign keys and let the datastore built-in “referential integrity” look after such deletions. DataNucleus provides a PMF property datanucleus.deletionPolicy allowing enabling of this mode of operation.

The default setting of datanucleus.deletionPolicy is "JDO2" which performs deletion of related objects as follows
1. If dependent-field is true then use that to define the related objects to be deleted.
2. Else, if the column of the foreign-key field is NULLable then NULL it and leave the related object alone
3. Else deleted the related object (and throw exceptions if this fails for whatever datastore-related reason)

The other setting of datanucleus.deletionPolicy is "DataNucleus" which performs deletion of related objects as follows

1. If dependent-field is true then use that to define the related objects to be deleted.
2. If a foreign-key is specified (in MetaData) for the relation field then leave any deletion to the datastore to perform (or throw exceptions as necessary)
3. Else, if the column of the foreign-key field is NULLable then NULL it and leave the related object alone
4. Else deleted the related object (and throw exceptions if this fails for whatever datastore-related reason)

So, as you can see, with the second option you have the ability to utilise datastore "referential integrity" checking using your MetaData-specified <foreign-key> elements.

68.1.5 Persisting Relationships - Reachability At Commit

One further complication is that with JDO there is also a process called persistence-by-reachability at commit. When objects are persisted, other objects are persisted with them. If some relations are changed before commit and some of these related objects are no longer required to be persistent then they will not be persisted. For example, using our classes above

```java
Owner bob = new Owner("Bob Smith");
DrivingLicense license = new DrivingLicense("233424BX4J");
bob.setLicense(license); // "bob" knows about "license"
pm.makePersistent(bob);

DrivingLicense license2 = new DrivingLicense("344566A99XH");
bob.setLicense(license2); // "bob" doesn't know about "license" now. It knows about "license2" now.

// "bob" and "license2" will be persisted but "license" won't be since not persisted explicitly
// and at commit it is no longer reachable from a persisted object

tx.commit();
```

With DataNucleus you can turn off persistence-by-reachability at commit by setting the PersistenceManagerFactory property datanucleus.persistenceByReachabilityAtCommit to false.
69 MetaData Reference

69.1 JDO : Metadata Overview

JDO requires the persistence of classes to be defined via Metadata. This Metadata can be provided in the following forms

- **XML**: the traditional mechanism, with XML files containing information for each class to be persisted. As a further complication you can define basic persistence metadata for a class in one file, and then ORM metadata for that class in a separate file (since the ORM metadata is specific to a certain datastore).
- **Annotations**: using JDK1.5+ annotations in the classes to be persisted
- **API**: a programmatic API allowing definition of which classes are to be persisted at runtime

When not using the MetaData API we recommend that you use either XML or annotations for the basic persistence information, but always use XML for ORM information. This is because it is liable to change at deployment time and hence is accessible when in XML form whereas in annotations you add an extra compile cycle (and also you may need to deploy to some other datastore at some point, hence needing a different deployment).

69.1.1 Metadata priority

JDO defines the priority order for metadata as being

- API Metadata
- ORM XML Metadata
- JDO XML Metadata
- Annotations

So if a class has Metadata defined by API then that will override all other Metadata. If a class has annotations and JDO XML Metadata then the XML Metadata will take precedence over the annotations (or rather be merged on top of the annotations). You can use whichever (or multiple) of the above forms at the same time for a single class.

69.1.2 XML Metadata loading

JDO expects the XML Meta-Data to be specified in a file or files in particular locations in the file system. For example, if you have a class `com.mycompany.sample.myexample`, JDO will look for any of the following files until it finds one (in the order stated):

```
META-INF/package.jdo
WEB-INF/package.jdo
package.jdo
com/package.jdo
com/mycompany/package.jdo
com/mycompany/sample/package.jdo
com/mycompany/sample/myexample.jdo
```

In addition, for this example, DataNucleus allows the previous JDO 1.0.0 alternatives of
In addition to the above, you can split your MetaData definitions between JDO MetaData files. For example if you have the following classes

```
com/mycompany/A.java
com/mycompany/B.java
com/mycompany/C.java
com/mycompany/app1/D.java
com/mycompany/app1/E.java
```

You could define the MetaData for these 5 classes in many ways -- for example put all class definitions in `com/mycompany/package.jdo`, or put the definitions for D and E in `com/mycompany/app1/package.jdo` and the definitions for A, B, C in `com/mycompany/package.jdo`, or have some in their class named MetaData files e.g `com/mycompany/app1/A.jdo`, or a mixture of the above. DataNucleus will always search for the MetaData file containing the class definition for the class that it requires.

69.1.3 XML Metadata validation

By default any XML Metadata will be validated for accuracy when loading it. Obviously XML is defined by a DTD or XSD schema and so should follow that. You can turn off such validations by setting the persistence property `datanucleus.metadata.validate` to false when creating your PMF. Note that this only turns off the XML strictness validation, and not the checks on inconsistency of specification of relations etc.

69.1.4 XML ORM Metadata usage

You can use ORM metadata to override particular datastore-specific things like table and column names. If your application doesn't make use of ORM metadata then you could turn off the searches for ORM Metadata files when a class is loaded up. You do this with the persistence property `datanucleus.metadata.supportORM` setting it to false.

69.1.5 Metadata discovery at class initialisation

JDO provides a mechanism whereby when a class is initialised (by the ClassLoader) any PersistenceManagerFactory is notified of its existence, and its Metadata can be loaded. This is enabled by the enhancement process. If you decided that you maybe only wanted some classes present in one PMF and other classes present in a different PMF then you can disable this and leave it to DataNucleus to discover the Metadata when operations are performed on that PMF. The persistence property to define to disable this is `datanucleus.metadata.autoregistration` (setting it to false).
70.1 JDO : XML Meta-Data Reference

JDO has always accepted Metadata in XML format. As described in the Metadata Overview this has to be contained in files with particular filenames in particular locations (relative to the name of the class), and that this metadata is discovered at runtime. You can provide JDO metadata, or alternatively ORM metadata, but with virtually identical format. This page defines the format of the XML Metadata. Here is an example header for package.jdo files with JDO XSD specification

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<jdo xmlns="http://xmlns.jcp.org/xml/ns/jdo/jdo"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/jdo/jdo
    http://xmlns.jcp.org/xml/ns/jdo/jdo_3_0.xsd" version="3.0">
...
</jdo>
```

Here is an example header for package.orm files with ORM XSD specification

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<orm xmlns="http://xmlns.jcp.org/xml/ns/jdo/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/jdo/orm
    http://xmlns.jcp.org/xml/ns/jdo/orm_3_0.xsd" version="3.0">
...
</orm>
```

What follows provides a reference guide to MetaData elements (refer to the relevant XSD for precise details).

- jdo
  - package
    - class
      - datastore-identity
      - column
      - extension
      - primary-key
      - column
      - inheritance
        - discriminator
          - column
          - join
          - column
- version
  - column
  - extension
- join
  - column
- foreign-key
  - column
  - field
  - property
- index
  - column
  - field
  - property
- unique
  - column
  - field
  - property
- field
  - collection
    - extension
  - map
    - extension
  - array
  - join
    - primary-key
    - index
    - column
  - embedded
    - field
      - column
  - element
    - column
  - key
    - column
  - value
    - column
• order
  • column
  • extension
• column
  • extension
• foreign-key
  • column
• index
  • column
• unique
  • column
• extension
• property
• collection
  • extension
• map
  • extension
• array
• join
  • primary-key
  • index
  • column
• embedded
  • field
    • column
• element
  • column
• key
  • column
• value
  • column
• order
  • column
• column
  • extension
• foreign-key
  • column
  • index
  • column
  • unique
  • column
  • extension
  • fetch-group
  • field
  • query
  • sequence
  • extension
  • fetch-plan
  • extension

70.1.1 Metadata for package tag
These are attributes within the `<package>` tag (jdo/package). This is used to denote a package, and all of the `<class>` elements that follow are in this Java package.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard (JDO) Tags</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>Name of the java package</td>
<td></td>
</tr>
<tr>
<td>catalog</td>
<td>Name of the catalog in which to persist the classes in this package. See also the property name &quot;javax.jdo.mapping.Catalog&quot; in the PMF Guide.</td>
<td></td>
</tr>
<tr>
<td>schema</td>
<td>Name of the schema in which to persist the classes in this package. See also the property name &quot;javax.jdo.mapping.Schema&quot; in the PMF Guide.</td>
<td></td>
</tr>
</tbody>
</table>

70.1.2 Metadata for class tag
These are attributes within the `<class>` tag (jdo/package/class). This is used to define the persistence definition for this class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
</table>
## Standard (JDO) Tags

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the class to persist</td>
<td></td>
</tr>
<tr>
<td>identity-type</td>
<td>The identity type, specifying whether they are uniquely provided by the JDO implementation (datastore identity), accessible fields in the object (application identity), or not at all (nondurable identity). <strong>DataNucleus only supports nondurable identity for SQL views.</strong></td>
<td>datastore, application, nondurable</td>
</tr>
<tr>
<td>objectid-class</td>
<td>The class name of the primary key. When using <strong>application identity.</strong></td>
<td></td>
</tr>
<tr>
<td>requires-extent</td>
<td>Whether the JDO implementation must provide an Extent for this class.</td>
<td>true, false</td>
</tr>
<tr>
<td>detachable</td>
<td>Whether the class is detachable from the persistence graph.</td>
<td>true, false</td>
</tr>
<tr>
<td>embedded-only</td>
<td>Whether this class should only be stored embedded in the tables for other classes.</td>
<td>true, false</td>
</tr>
<tr>
<td>persistence-modifier</td>
<td>What type of persistability type this class exhibits. Please refer to <strong>JDO Class Types.</strong></td>
<td>persistence-capable</td>
</tr>
<tr>
<td>persistence-capable-superclass</td>
<td>Class name of superclass that is persistent capable. This is deprecated in JDO2 and you no longer need to specify it, leaving it to DataNucleus to determine if there is a superclass that is <strong>PersistenceCapable.</strong></td>
<td></td>
</tr>
<tr>
<td>catalog</td>
<td>Name of the catalog in which to persist the class. See also the property name &quot;javax.jdo.mapping.Catalog&quot; in the <strong>PMF Guide.</strong></td>
<td></td>
</tr>
<tr>
<td>schema</td>
<td>Name of the schema in which to persist the class. See also the property name &quot;javax.jdo.mapping.Schema&quot; in the <strong>PMF Guide.</strong></td>
<td></td>
</tr>
<tr>
<td>table</td>
<td>Name of the table/view in which to persist the class. See also the property name &quot;datanucleus.identifier.case&quot; in the <strong>Persistence Properties Guide.</strong></td>
<td></td>
</tr>
<tr>
<td>cacheable</td>
<td>Whether the class can be cached in a Level 2 cache. <strong>From JDO2.2</strong></td>
<td>true</td>
</tr>
<tr>
<td>serializeRead</td>
<td>Whether to default to locking objects of this type when reading them. <strong>From JDO2.2</strong></td>
<td>true</td>
</tr>
</tbody>
</table>
### 70.1.3 Metadata for datastore-identity tag

These are attributes within the `<datastore-identity>` tag (jdo/package/class/datastore-identity). This is used when the `<class>` to which this pertains uses datastore identity. It is used to define the precise definition of datastore identity to be used. This element can contain `<column>` sub-elements allowing definition of the column details where required - these are optional.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard (JDO) Tags</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>strategy</strong></td>
<td>Strategy for datastore identity generation for this class. native allows DataNucleus to choose the most suitable for the datastore. sequence will use a sequence (specified by the attribute sequence) - if supported by the datastore. increment will use the id values in the datastore to decide the next id. uuid-string will use a UUID string generator (16-characters). uuid-hex will use a UUID string generator (32-characters). identity will use a datastore inbuilt auto-incrementing types. auid is a DataNucleus extension, that is an almost universal id generator (best possible derivate of a DCE UUID). max is a DataNucleus extension, that uses &quot;select max(column)+1 from table&quot; for the identity. timestamp is a DataNucleus extension, providing the current timestamp. timestamp-value is a DataNucleus extension, providing the current timestamp millisecs. [other values] to utilise user-supplied DataNucleus value generator plugins.</td>
<td>native</td>
</tr>
<tr>
<td><strong>sequence</strong></td>
<td>Name of the sequence to use to generate identity values, when using a strategy of sequence. Please see also the class extension tags for controlling the sequence.</td>
<td></td>
</tr>
<tr>
<td><strong>column</strong></td>
<td>Name of the column used for the datastore identity for this class.</td>
<td></td>
</tr>
</tbody>
</table>

These are attributes within the `<extension>` tag (jdo/package/class/datastore-identity/extension). These are for controlling the generation of ids when in datastore identity mode.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sequence-table-basis</code></td>
<td>This defines the basis on which to generate unique identities when using the TableValueGenerator (used by the &quot;increment&quot; strategy, and sometimes by &quot;native&quot;). You can either define identities unique against the base table name, or against the base class name (in an inheritance tree). Used when the strategy is set to native or increment.</td>
<td>class</td>
</tr>
<tr>
<td><code>sequence-catalog-name</code></td>
<td>The catalog used to store sequences for use by value generators. See Value Generation. Default catalog for the datastore will be used if not specified.</td>
<td></td>
</tr>
<tr>
<td><code>sequence-schema-name</code></td>
<td>The schema used to store sequences for use by value generators. See Value Generation. Default schema for the datastore will be used if not specified.</td>
<td></td>
</tr>
<tr>
<td><code>sequence-table-name</code></td>
<td>The table used to store sequences for use by value generators. See Value Generation.</td>
<td>SEQUENCE_TABLE</td>
</tr>
<tr>
<td><code>sequence-name-column-name</code></td>
<td>The column name in the sequence-table used to store the name of the sequence for use by value generators. See Value Generation.</td>
<td>SEQUENCE_NAME</td>
</tr>
<tr>
<td><code>sequence-nextval-column-name</code></td>
<td>The column name in the sequence-table used to store the next value in the sequence for use by value generators. See Value Generation.</td>
<td>NEXT_VAL</td>
</tr>
<tr>
<td><code>key-min-value</code></td>
<td>The minimum key value for use by value generators. Keys lower than this will not be generated. See Value Generation.</td>
<td></td>
</tr>
<tr>
<td><code>key-max-value</code></td>
<td>The maximum key value for use by value generators. Keys higher than this will not be generated. See Value Generation.</td>
<td></td>
</tr>
<tr>
<td><code>key-initial-value</code></td>
<td>The starting value for use by value generators. Keys will start from this value when being generated. See Value Generation.</td>
<td></td>
</tr>
<tr>
<td><code>key-cache-size</code></td>
<td>The cache size for keys for use by value generators. The cache of keys will be constrained by this value. See Value Generation.</td>
<td></td>
</tr>
</tbody>
</table>
The database cache size for keys for use by value generators. The cache of keys will be constrained by this value. See Value Generation.

70.1.4 Metadata for primary-key tag
These are attributes within the `<primary-key>` tag (jdo/package/class/primary-key or class/field/join/primary-key). It is used to specify the name of the primary key constraint in the datastore during the schema generation process. When used under `<join>` it specifies that the join table has a primary-key.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the primary key constraint.</td>
<td></td>
</tr>
<tr>
<td>column</td>
<td>Name of the column to use for the primary key</td>
<td></td>
</tr>
</tbody>
</table>

70.1.5 Metadata for inheritance tag
These are attributes within the `<inheritance>` tag (jdo/package/class/inheritance). It is used when this class is part of an inheritance tree, and to denote how the class is stored in the datastore since there are several ways (strategies) in which it can be stored.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>strategy</td>
<td>Strategy for inheritance of this class. Please refer to the Inheritance Guide. Note that &quot;complete-table&quot; is a DataNucleus extension to JDO2</td>
<td>new-table, subclass-table, superclass-table, complete-table</td>
</tr>
</tbody>
</table>

70.1.6 Metadata for discriminator tag
These are attributes within the `<discriminator>` tag (jdo/package/class/inheritance/discriminator). This is used to define a discriminator column that is used when this class is stored in the same table as another class in the same inheritance tree. The discriminator column will contain a value for objects of this class, and different values for objects of other classes in the inheritance tree.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>strategy</td>
<td>Strategy for the discrimination column</td>
<td>value-map</td>
</tr>
</tbody>
</table>
70.1.7 Metadata for version tag

These are attributes within the `<version>` tag (jdo/package/class/version). This is used to define whether and how this class is handled with respect to optimistic transactions.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>strategy</td>
<td>Strategy for versioning of this class. The &quot;version-number&quot; mode uses an incremental numbered value, and the &quot;date-time&quot; mode uses a java.sql.Timestamp value. <code>state-image</code> isn't currently supported.</td>
<td>state-image, date-time, version-number</td>
</tr>
<tr>
<td>column</td>
<td>Name of the column in the datastore to store this field</td>
<td></td>
</tr>
<tr>
<td>indexed</td>
<td>Whether the version column should be indexed. This is to be specified when defining index information</td>
<td>true</td>
</tr>
</tbody>
</table>

These are attributes within the `<extension>` tag (jdo/package/class/version/extension).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>field-name</td>
<td>This extension allows you to define a field that will be used to contain the version of the object. It is populated by DataNucleus at persist. See JDO Versioning</td>
<td></td>
</tr>
</tbody>
</table>

70.1.8 Metadata for query tag

These are attributes within the `<query>` tag (jdo/package/class/query). This element is used to define any "named queries" that are to be available for this class. This element contains the query single-string form as its content.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
name

Name of the query. This name is mandatory and is used in calls to `pm.newNamedQuery()`. Has to be unique for this class.

attribute

Language to use. Some datastores offer other languages. Values: `JDOQL`, `SQL`, `JPQL`.

unique

Whether the query is to return a unique result (only for SQL queries). Values: `true`, `false`.

result-class

Class name of any result class (only for SQL queries).

### 70.1.9 Metadata for field tag

These are attributes within the `<field>` tag (`jdo/package/class/field`). This is used to define the persistence behaviour of the fields of the class to which it pertains. Certain types of fields are, by default, persisted. This element can be used to change the default behaviour and maybe not persist a field, or to persist something that normally isn't persisted. It is used, in addition, to define more details about how the field is persisted in the datastore.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard (JDO) Tags</strong></td>
<td>Name of the field.</td>
<td>Name of the field.</td>
</tr>
<tr>
<td>name</td>
<td>Name of the field.</td>
<td>Name of the field.</td>
</tr>
<tr>
<td>persistence-modifier</td>
<td>The persistence-modifier specifies how JDO manage each field in your persistent class. There are three options: persistent, transactional and none.</td>
<td>persistent, transactional, none</td>
</tr>
<tr>
<td>primary-key</td>
<td>Whether the field is part of any primary key (if using application identity).</td>
<td>true, false</td>
</tr>
</tbody>
</table>

- **persistent** means that your field will managed by JDO and stored in the database on transaction commit.

- **transactional** means that your field will managed by JDO but not stored in the database. Transactional fields values will be saved by JDO when you start your transaction and restored when you roll back your transaction.

- **none** means that your field will not be managed by JDO.
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Allowed Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>null-value</td>
<td>How to treat null values of persistent fields during storage.</td>
<td>exception, default, none</td>
</tr>
<tr>
<td>default-fetch-group</td>
<td>Whether this field is part of the default fetch group for the class.</td>
<td>true, false</td>
</tr>
<tr>
<td>embedded</td>
<td>Whether this field should be stored, if possible, as part of the object</td>
<td>true, false</td>
</tr>
<tr>
<td>serialized</td>
<td>Whether this field should be stored serialised into a single column of the</td>
<td>true, false</td>
</tr>
<tr>
<td>dependent</td>
<td>Whether the field should be used to check for dependent objects, and to</td>
<td>true, false</td>
</tr>
<tr>
<td>mapped-by</td>
<td>The name of the field at the other end of a relationship. Used by 1-1, 1-N,</td>
<td></td>
</tr>
<tr>
<td>value-strategy</td>
<td>The strategy for populating values to this field. Is typically used for</td>
<td>native</td>
</tr>
<tr>
<td>sequence</td>
<td>Name of the sequence to use to generate values, when using a strategy of</td>
<td></td>
</tr>
<tr>
<td>recursion-depth</td>
<td>The depth that will be recursed when this field is self-referencing.</td>
<td>-1, 1, 2, ... (integer)</td>
</tr>
</tbody>
</table>
These are attributes within the `<extension>` tag (jdo/package/class/field/extension).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Extension (JDO) Tags</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cascade-persist</td>
<td>JDO defines that when an object is persisted then all fields will also be persisted using &quot;persistence-by-reachability&quot;. This extension allows you to turn off the persistence of a field relation.</td>
<td>true</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Examples</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>cascade-update</td>
<td>JDO defines that when an object is updated then all fields containing PersistenceCapable objects will also be updated using &quot;persistence-by-reachability&quot;. This extension allows you to turn off the update of a field relation.</td>
<td><strong>true</strong></td>
</tr>
<tr>
<td>cascade-refresh</td>
<td>When calling PersistenceManager.refresh() only fetch plan fields of the passed object will be refreshed. Setting this to true will refresh the fields of related PC objects in this field.</td>
<td><strong>true</strong></td>
</tr>
<tr>
<td>allow-nulls</td>
<td>When the field is a collection by default it will not be allowed to have nulls present but you can allow them by setting this DataNucleus extension tag.</td>
<td><strong>true</strong></td>
</tr>
<tr>
<td>insertable</td>
<td>Whether this field should be supplied when inserting into the datastore.</td>
<td><strong>true</strong></td>
</tr>
<tr>
<td>updateable</td>
<td>Whether this field should be supplied when updating the datastore.</td>
<td><strong>true</strong></td>
</tr>
<tr>
<td>adapter-column-name</td>
<td>In some situations DataNucleus will add a special datastore column to a join table so that collections can allow the storage of duplicate elements. This extension allows the specification of the column name to be used. This should be specified within the field at the collection end of the relationship. JDO2 doesn't allow a standard place for such a specification and so is an extension tag.</td>
<td><strong>INTEGER_IDX</strong></td>
</tr>
<tr>
<td>implementation-classes</td>
<td>Used to define the possible classes implementing this interface/Object field. This is used to limit the possible tables that this is a foreign key to (when this field is specified as an interface/Object in the class). Value should be comma-separated list of fully-qualified class names</td>
<td><strong>true</strong></td>
</tr>
<tr>
<td>key-implementation-classes</td>
<td>Used to define the possible classes implementing this interface/Object key. This is used to limit the possible tables that this is a foreign key to (when this key is specified as an interface/Object). Value should be comma-separated list of fully-qualified class names</td>
<td><strong>true</strong></td>
</tr>
</tbody>
</table>
value-implementation-classes
Used to define the possible classes implementing this interface/Object value. This is used to limit the possible tables that this is a foreign key to (when this value is specified as an interface/Object). Value should be comma-separated list of fully-qualified class names

strategy-when-notnull
This is to be used in conjunction with the "value-strategy" attribute. Default JDO2 behaviour when you have a "value-strategy" defined for a field is to always create a strategy value for that field regardless of whether you have set the value of the field yourself. This extension allows you to only apply the strategy if the field is null at persistence. This extension has no effect on primitive field types (which can't be null) and the value-strategy will always be applied to such fields.

true | false

relation-discriminator-column
Name of a column to use for discrimination of the relation used by objects stored. This is defined when, for example, a join table is shared by multiple relations and the objects placed in the join table need discriminating for which relation they are for.

RELATION_DISCRIM

relation-discriminator-pk
Whether the column added for the discrimination of relations is to be part of the PK when using a join table.

true | false

relation-discriminator-value
Value to use in the relation discriminator column for objects of this fields relation. This is defined when, for example, a join table is shared by multiple relations and the objects placed in the join table need discriminating for which relation they are for.

Fully-qualified class name

select-function
Permits to use a function when fetching contents from the database. A ? (question mark) is mandatory to have and will be replaced by the column name when generating the SQL statement. For example to specify a value of UPPER(?) will convert the field value to upper case on a datastore that supports that UPPER function.
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert-function</td>
<td>Permits to use a function when inserting into the database. A ? (question mark) is optional and will be replaced by the column name when generating the SQL statement. For example to specify a value of TRIM(?) will trim the field value on a datastore that supports that TRIM function.</td>
</tr>
<tr>
<td>update-function</td>
<td>Permits to use a function when updating into the database. A ? (question mark) is optional and will be replaced by the column name when generating the SQL statement. For example to specify a value of FUNC(?) will perform &quot;FUNC&quot; on the field value on a datastore that supports that FUNC function.</td>
</tr>
<tr>
<td>sequence-table-basis</td>
<td>This defines the basis on which to generate unique identities when using the TableValueGenerator (used by the &quot;increment&quot; strategy, and sometimes by &quot;native&quot;). You can either define identities unique against the base table name, or against the base class name (in an inheritance tree). Used when the strategy is set to native or increment.</td>
</tr>
<tr>
<td>sequence-catalog-name</td>
<td>The catalog used to store sequences for use by value generators. See Value Generation. Default catalog for the datastore will be used if not specified.</td>
</tr>
<tr>
<td>sequence-schema-name</td>
<td>The schema used to store sequences for use by value generators. See Value Generation. Default schema for the datastore will be used if not specified.</td>
</tr>
<tr>
<td>sequence-table-name</td>
<td>The table used to store sequences for use by value generators. See Value Generation.</td>
</tr>
<tr>
<td>sequence-name-column-name</td>
<td>The column name in the sequence-table used to store the name of the sequence for use by value generators. See Value Generation.</td>
</tr>
<tr>
<td>sequence-nextval-column-name</td>
<td>The column name in the sequence-table used to store the next value in the sequence for use by value generators. See Value Generation.</td>
</tr>
<tr>
<td>key-min-value</td>
<td>The minimum key value for use by value generators. Keys lower than this will not be generated. See Value Generation.</td>
</tr>
</tbody>
</table>
key-max-value | The maximum key value for use by value generators. Keys higher than this will not be generated. See Value Generation.

key-initial-value | The starting value for use by value generators. Keys will start from this value when being generated. See Value Generation.

key-cache-size | The cache size for keys for use by value generators. The cache of keys will be constrained by this value. See Value Generation.

key-database-cache-size | The database cache size for keys for use by value generators. The cache of keys will be constrained by this value. See Value Generation.

mapping-class | Specifies the mapping class to be used for mapping this field. This is only used where the user wants to override the default DataNucleus mapping class and provide their own mapping class for this field.

70.1.10 Metadata for property tag

These are attributes within the `<property>` tag (jdo/package/class/property). This is used to define the persistence behaviour of the Java Bean properties of the class to which it pertains. This element can be used to change the default behaviour and maybe not persist a property, or to persist something that normally isn't persisted. It is used, in addition, to define more details about how the property is persisted in the datastore.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard (JDO) Tags</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>Name of the property. The &quot;name&quot; of a property is obtained by taking the getXXX, setXXX method names and using the XXX and making the first letter lowercase.</td>
<td></td>
</tr>
<tr>
<td>Key</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>persistence-modifier</td>
<td>The persistence-modifier specifies how to manage each property in your persistent class. There are three options: persistent, transactional and none.</td>
<td>persistent, transactional, none</td>
</tr>
<tr>
<td>• persistent</td>
<td>means that your field will be managed and stored in the database on transaction commit.</td>
<td></td>
</tr>
<tr>
<td>• transactional</td>
<td>means that your field will be managed but not stored in the database. Transactional fields values will be saved by JDO when you start your transaction and restored when you roll back your transaction.</td>
<td></td>
</tr>
<tr>
<td>• none</td>
<td>means that your field will not be managed.</td>
<td></td>
</tr>
<tr>
<td>primary-key</td>
<td>Whether the property is part of any primary key (if using application identity).</td>
<td>true, false</td>
</tr>
<tr>
<td>null-value</td>
<td>How to treat null values of persistent properties during storage.</td>
<td>exception, default, none</td>
</tr>
<tr>
<td>default-fetch-group</td>
<td>Whether this property is part of the default fetch group for the class. Defaults to true for non-key fields of primitive types, java.util.Date, java.lang.<em>, java.math.</em>, etc.</td>
<td>true, false</td>
</tr>
<tr>
<td>embedded</td>
<td>Whether this property should be stored, if possible, as part of the object instead as its own object in the datastore. This defaults to true for primitive types, java.util.Date, java.lang.<em>, java.math.</em> etc and false for PersistenceCapable, reference (Object, Interface) and container types.</td>
<td>true, false</td>
</tr>
<tr>
<td>serialized</td>
<td>Whether this property should be stored serialised into a single column of the table of the containing object.</td>
<td>true, false</td>
</tr>
<tr>
<td>dependent</td>
<td>Whether the property should be used to check for dependent objects, and to delete them when this object is deleted. In other words cascade delete capable.</td>
<td>true, false</td>
</tr>
<tr>
<td>mapped-by</td>
<td>The name of the property at the other end of a relationship. Used by 1-1, 1-N, M-N to mark a relation as bidirectional.</td>
<td></td>
</tr>
<tr>
<td><strong>value-strategy</strong></td>
<td>The strategy for populating values to this property. Is typically used for generating primary key values. See the definitions under &quot;datastore-identity&quot;.</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>sequence</strong></td>
<td>Name of the sequence to use to generate values, when using a <strong>strategy</strong> of <strong>sequence</strong>. Please see also the <strong>class</strong> extension tags for controlling the sequence.</td>
<td></td>
</tr>
<tr>
<td><strong>recursion-depth</strong></td>
<td>The depth that will be recursed when this property is self-referencing. Should be used alongside FetchPlan.setMaxFetchDepth() to control the objects fetched.</td>
<td></td>
</tr>
<tr>
<td><strong>field-type</strong></td>
<td>Used to specify a more restrictive type than the property definition in the class. This might be required in order to map the field to the datastore. To be portable, specify the name of a single type that is itself able to be mapped to the datastore (e.g. a field of type Object can specify field-type=&quot;Integer&quot;).</td>
<td></td>
</tr>
<tr>
<td><strong>indexed</strong></td>
<td>Whether the column(s) for this property should be indexed. This is to be specified when defining index information</td>
<td></td>
</tr>
<tr>
<td><strong>table</strong></td>
<td>Table name to use for any join table overriding the default name provided by DataNucleus. This is used either for 1-N relationships with a join table or for Secondary Tables. See also the property name &quot;datanucleus.identifier.case&quot; in the Persistence Properties Guide.</td>
<td></td>
</tr>
<tr>
<td><strong>column</strong></td>
<td>Column name to use for this property (alternative to specifying column sub-elements if only one column).</td>
<td></td>
</tr>
<tr>
<td><strong>delete-action</strong></td>
<td>The foreign-key delete action. This is a shortcut to specifying foreign key information. Please refer to the <code>&lt;foreign-key&gt;</code> element for full details.</td>
<td></td>
</tr>
<tr>
<td><strong>cacheable</strong></td>
<td>Whether the field/property can be cached in a Level 2 cache. <strong>From JDO2.2</strong></td>
<td></td>
</tr>
</tbody>
</table>

Options:
- native
- sequence
- increment
- identity
- uuid-string
- uuid-hex
- auid
- max
- timestamp
- timestamp-value
- [other values]
- -1, 1, 2, ... (integer)
- true, false, unique
load-fetch-group
Name of a fetch group to activate when a load of this field is initiated (due to it being currently unloaded). Not used for getObjectById, queries, extents etc. Better to use "fetch-group" and define your groups.

These are attributes within the `<extension>` tag (jdo/package/class/property/extension).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>cascade-persist</td>
<td>JDO defines that when an object is persisted then all fields will also be persisted using &quot;persistence-by-reachability&quot;. This extension allows you to turn off the persistence of a field relation.</td>
<td>true</td>
</tr>
<tr>
<td>cascade-update</td>
<td>JDO defines that when an object is updated then all fields containing PersistenceCapable objects will also be updated using &quot;persistence-by-reachability&quot;. This extension allows you to turn off the update of a field relation.</td>
<td>true</td>
</tr>
<tr>
<td>cascade-refresh</td>
<td>When calling PersistenceManager.refresh() only fetch plan fields of the passed object will be refreshed. Setting this to true will refresh the fields of related PC objects in this field</td>
<td>true</td>
</tr>
<tr>
<td>allow-nulls</td>
<td>When the field is a collection by default it will not be allowed to have nulls present but you can allow them by setting this DataNucleus extension tag</td>
<td>true</td>
</tr>
<tr>
<td>insertable</td>
<td>Whether this field should be supplied when inserting into the datastore.</td>
<td>true</td>
</tr>
<tr>
<td>updateable</td>
<td>Whether this field should be supplied when updating the datastore.</td>
<td>true</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>adapter-column-name</td>
<td>In some situations DataNucleus will add a special datastore column to a join table so that collections can allow the storage of duplicate elements. This extension allows the specification of the column name to be used. This should be specified within the field at the collection end of the relationship. JDO2 doesn't allow a standard place for such a specification and so is an extension tag.</td>
<td></td>
</tr>
<tr>
<td>implementation-classes</td>
<td>Used to define the possible classes implementing this interface/Object field. This is used to limit the possible tables that this is a foreign key to (when this field is specified as an interface/Object in the class). Value should be comma-separated list of fully-qualified class names</td>
<td></td>
</tr>
<tr>
<td>key-implementation-classes</td>
<td>Used to define the possible classes implementing this interface/Object key. This is used to limit the possible tables that this is a foreign key to (when this key is specified as an interface/Object). Value should be comma-separated list of fully-qualified class names</td>
<td></td>
</tr>
<tr>
<td>value-implementation-classes</td>
<td>Used to define the possible classes implementing this interface/Object value. This is used to limit the possible tables that this is a foreign key to (when this value is specified as an interface/Object). Value should be comma-separated list of fully-qualified class names</td>
<td></td>
</tr>
<tr>
<td>strategy-when-notnull</td>
<td>This is to be used in conjunction with the &quot;value-strategy&quot; attribute. Default JDO2 behaviour when you have a &quot;value-strategy&quot; defined for a field is to always create a strategy value for that field regardless of whether you have set the value of the field yourself. This extension allows you to only apply the strategy if the field is null at persistence. This extension has no effect on primitive field types (which can't be null) and the value-strategy will always be applied to such fields.</td>
<td>true</td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
<td>Value</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>relation-discriminator-column</td>
<td>Name of a column to use for discrimination of the relation used by objects stored. This is defined when, for example, a join table is shared by multiple relations and the objects placed in the join table need discriminating for which relation they are for.</td>
<td>RELATION_DISCRIM</td>
</tr>
<tr>
<td>relation-discriminator-pk</td>
<td>Whether the column added for the discrimination of relations is to be part of the PK when using a join table.</td>
<td>true</td>
</tr>
<tr>
<td>relation-discriminator-value</td>
<td>Value to use in the relation discriminator column for objects of this fields relation. This is defined when, for example, a join table is shared by multiple relations and the objects placed in the join table need discriminating for which relation they are for.</td>
<td>Fully-qualified class name</td>
</tr>
<tr>
<td>select-function</td>
<td>Permits to use a function when fetching contents from the database. A ? (question mark) is mandatory to have and will be replaced by the column name when generating the SQL statement. For example to specify a value of UPPER(?) will convert to upper case the field value on a datastore that supports that UPPER function.</td>
<td></td>
</tr>
<tr>
<td>insert-function</td>
<td>Permits to use a function when inserting into the database. A ? (question mark) is optional and will be replaced by the column name when generating the SQL statement. For example to specify a value of TRIM(?) will trim the field value on a datastore that supports that TRIM function.</td>
<td></td>
</tr>
<tr>
<td>update-function</td>
<td>Permits to use a function when updating into the database. A ? (question mark) is optional and will be replaced by the column name when generating the SQL statement. For example to specify a value of FUNC(?) will perform FUNC() on the field value on a datastore that supports that FUNC function.</td>
<td></td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>sequence-table-basis</td>
<td>This defines the basis on which to generate unique identities when using the TableValueGenerator (used by the &quot;increment&quot; strategy, and sometimes by &quot;native&quot;). You can either define identities unique against the base table name, or against the base class name (in an inheritance tree). Used when the strategy is set to native or increment.</td>
<td>class</td>
</tr>
<tr>
<td>sequence-catalog-name</td>
<td>The catalog used to store sequences for use by value generators. See Value Generation. Default catalog for the datastore will be used if not specified.</td>
<td></td>
</tr>
<tr>
<td>sequence-schema-name</td>
<td>The schema used to store sequences for use by value generators. See Value Generation. Default schema for the datastore will be used if not specified.</td>
<td></td>
</tr>
<tr>
<td>sequence-table-name</td>
<td>The table used to store sequences for use by value generators. See Value Generation.</td>
<td>SEQUENCE_TABLE</td>
</tr>
<tr>
<td>sequence-name-column-name</td>
<td>The column name in the sequence-table used to store the name of the sequence for use by value generators. See Value Generation.</td>
<td>SEQUENCE_NAME</td>
</tr>
<tr>
<td>sequence-nextval-column-name</td>
<td>The column name in the sequence-table used to store the next value in the sequence for use by value generators. See Value Generation.</td>
<td>NEXT_VAL</td>
</tr>
<tr>
<td>key-min-value</td>
<td>The minimum key value for use by value generators. Keys lower than this will not be generated. See Value Generation.</td>
<td></td>
</tr>
<tr>
<td>key-max-value</td>
<td>The maximum key value for use by value generators. Keys higher than this will not be generated. See Value Generation.</td>
<td></td>
</tr>
<tr>
<td>key-initial-value</td>
<td>The starting value for use by value generators. Keys will start from this value when being generated. See Value Generation.</td>
<td></td>
</tr>
<tr>
<td>key-cache-size</td>
<td>The cache size for keys for use by value generators. The cache of keys will be constrained by this value. See Value Generation.</td>
<td></td>
</tr>
<tr>
<td>key-database-cache-size</td>
<td>The database cache size for keys for use by value generators. The cache of keys will be constrained by this value. See Value Generation.</td>
<td></td>
</tr>
</tbody>
</table>
mapping-class Specifies the mapping class to be used for mapping this field. This is only used where the user wants to override the default DataNucleus mapping class and provide their own mapping class for this field. Fully-qualified class name

70.1.11 Metadata for fetch-group tag

These are attributes within the `<fetch-group>` tag (jdo/package/class/fetch-group). This element is used to define fetch groups that are utilised at runtime, and are of particular use with attach/detach. This element can contain `fetch-group` sub-elements allowing definition of hierarchical groups. It can also contain `field` elements, defining the fields that are part of this fetch-group.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the fetch group.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Used with the fetch plan of the PersistenceManager.</td>
<td></td>
</tr>
<tr>
<td>post-load</td>
<td>Whether to call jdoPostLoad when the fetch group is invoked.</td>
<td>true</td>
</tr>
</tbody>
</table>

70.1.12 Metadata for embedded tag

These are attributes within the `<embedded>` tag (jdo/package/class/embedded). It is used when this field is a PersistenceCapable and is embedded into the same table as the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner-field</td>
<td>Name of the field in the embedded PersistenceCapable that is the link back to the owning object (if any).</td>
<td></td>
</tr>
<tr>
<td>null-indicator-column</td>
<td>Name of the column to be used for detecting if the embedded object is null.</td>
<td></td>
</tr>
<tr>
<td>null-indicator-value</td>
<td>Value of the null-indicator-column that signifies that the embedded object is null.</td>
<td></td>
</tr>
</tbody>
</table>

70.1.13 Metadata for key tag

These are attributes within the `<key>` tag (jdo/package/class/field/key). This element is used to define details for the persistence of a Map.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapped-by</td>
<td>When the map is formed by a foreign-key, the key can be a field in a value PersistenceCapable class. This attribute defines which field in the value class is used as the key.</td>
<td></td>
</tr>
<tr>
<td>column</td>
<td>Name of the column (if only one)</td>
<td></td>
</tr>
<tr>
<td>delete-action</td>
<td>Action to be performed when the owner object is deleted. This is to be specified when defining foreign key information</td>
<td>cascade</td>
</tr>
<tr>
<td>indexed</td>
<td>Whether the key column should be indexed. This is to be specified when defining index information</td>
<td>true</td>
</tr>
<tr>
<td>unique</td>
<td>Whether the key column should be unique. This is to be specified when defining unique key information</td>
<td>true</td>
</tr>
</tbody>
</table>

### 70.1.14 Metadata for value tag

These are attributes within the `<value>` tag (jdo/package/class/field/value). This element is used to define details for the persistence of a Map.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapped-by</td>
<td>When the map is formed by a foreign-key, the value can be a field in a key PersistenceCapable class. This attribute defines which field in the key class is used as the value.</td>
<td></td>
</tr>
<tr>
<td>column</td>
<td>Name of the column (if only one)</td>
<td></td>
</tr>
<tr>
<td>delete-action</td>
<td>Action to be performed when the owner object is deleted. This is to be specified when defining foreign key information</td>
<td>cascade</td>
</tr>
<tr>
<td>indexed</td>
<td>Whether the value column should be indexed. This is to be specified when defining index information</td>
<td>true</td>
</tr>
<tr>
<td>unique</td>
<td>Whether the value column should be unique. This is to be specified when defining unique key information</td>
<td>true</td>
</tr>
</tbody>
</table>
70.1.15 Metadata for order tag

These are attributes within the `<order>` tag (jdo/package/class/field/order). This is used to define the column details for the ordering column in a List.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard (JDO) Tags</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mapped-by</td>
<td>When a List is formed by a foreign-key, the ordering can be a field in the element PersistenceCapable class. This attribute defines which field in the element class is used as the ordering. The field must be of type int, Integer, long, Long. DataNucleus will write the index positions to this field (starting at 0 for the first item in the List)</td>
<td></td>
</tr>
<tr>
<td>column</td>
<td>Name of the column to use for ordering.</td>
<td></td>
</tr>
</tbody>
</table>

These are attributes within the `<extension>` tag (jdo/package/class/field/order/extension).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Extension (JDO) Tags</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>list-ordering</td>
<td>Used to make the list be an &quot;ordered list&quot; where it has no index column and instead will order the elements by the specified expression upon retrieval. The ordering expression takes names and ASC/DESC and can be a composite</td>
<td>{orderfield [ASC</td>
</tr>
</tbody>
</table>

70.1.16 Metadata for index tag

These are attributes within the `<index>` tag (jdo/package/class/field/index). This element is used where a user wishes to add specific indexes to the datastore to provide more efficient access to particular fields.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard (JDO) Tags</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>Name of the index in the datastore</td>
<td></td>
</tr>
<tr>
<td>unique</td>
<td>Whether the index is unique</td>
<td>true</td>
</tr>
<tr>
<td>column</td>
<td>Name of the column to use (alternative to specifying it as a sub-element).</td>
<td></td>
</tr>
</tbody>
</table>
These are attributes within the `<extension>` tag (jdo/package/class/field/index/extension).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Extension (JDO) Tags</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>extended-setting</td>
<td>Additional settings to the index. This extension is used to set database proprietary settings.</td>
<td></td>
</tr>
</tbody>
</table>

70.1.17 Metadata for foreign-key tag

These are attributes within the `<foreign-key>` tag (jdo/package/class/field/foreign-key). This is used where the user wishes to define the behaviour of the foreign keys added due to the relationships in the object model. This is to be read in conjunction with [foreign-key guide](#).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard (JDO) Tags</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>Name of the foreign key in the datastore</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deferred</td>
<td>Whether the constraints are initially deferred.</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>delete-action</td>
<td>Action to be performed when the owner object is deleted.</td>
<td>cascade</td>
<td>restrict</td>
</tr>
<tr>
<td>update-action</td>
<td>Action to be performed when the owner object is updated.</td>
<td>cascade</td>
<td>restrict</td>
</tr>
</tbody>
</table>

70.1.18 Metadata for unique tag

These are attributes within the `<unique>` tag (jdo/package/class/unique, jdo/package/class/field/unique). This element is used where a user wishes to add specific unique constraints to the datastore to provide more control over particular fields.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard (JDO) Tags</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>Name of the constraint in the datastore</td>
<td></td>
<td></td>
</tr>
<tr>
<td>column</td>
<td>Name of the column to use (alternative to specifying it as a sub-element).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# 70.1.19 Metadata for column tag

These are attributes within the `<column>` tag (`*/column`). This is used to define the details of a column in the datastore, and so can be used to match to an existing datastore schema.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description                                                                 #</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the column in the datastore. See also the property name &quot;datanucleus.identifier.case&quot; in the Persistence Properties Guide.</td>
<td></td>
</tr>
<tr>
<td>length</td>
<td>Length of the column in the datastore (for character types), or the precision of the column in the datastore (for floating point field types).</td>
<td>positive integer</td>
</tr>
<tr>
<td>scale</td>
<td>Scale of the column in the datastore (for floating point field types).</td>
<td>positive integer</td>
</tr>
<tr>
<td>jdbc-type</td>
<td>JDBC Type to use for this column in the datastore when the default value is not satisfactory. Please refer to JDBC for the valid types. Not all of these types are supported for all RDBMS mappings.</td>
<td>Valid JDBC Type (CHAR, VARCHAR, LONGVARCHAR, NUMERIC, DECIMAL, BIT, TINYINT, SMALLINT, INTEGER, BIGINT, REAL, FLOAT, DOUBLE, BINARY, VARBINARY, LONGVARBINARY, DATE, TIME, TIMESTAMP, BLOB, BOOLEAN, CLOB, DATALINK)</td>
</tr>
<tr>
<td>sql-type</td>
<td>SQL Type to use for this column in the datastore. This should not usually be necessary since the specification of JDBC type together with length/scale will likely define it.</td>
<td>Valid SQL Type (e.g VARCHAR, CHAR, NUMERIC etc)</td>
</tr>
<tr>
<td>allows-null</td>
<td>Whether the column in the datastore table should allow nulls or not. The default is &quot;false&quot; for primitives, and &quot;true&quot; otherwise.</td>
<td>true</td>
</tr>
<tr>
<td>default-value</td>
<td>Default value to use for this column when creating the table. If you want the default to be NULL, then put this as &quot;#NULL&quot;. This is particularly for cases where you have a table that stores multiple classes in an inheritance tree (subclass-table, superclass-table) so when you persist a superclass object it doesn't have the subclass fields in its INSERT and so the datastore uses the default-value settings that are embodied in the CREATE TABLE statement.</td>
<td>Default value expression</td>
</tr>
</tbody>
</table>

---

©2015, DataNucleus • ALL RIGHTS RESERVED.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Extension (JDO) Tags</strong></td>
<td>Specifies the datastore mapping class to be used for mapping this field. This is only used where the user wants to override the default DataNucleus datastore mapping class and provide their own mapping class for this field based on the database data type. This datastore mapping class must be available for the DataNucleus PersistenceManagerFactory classpath.</td>
<td>Fully-qualified class name</td>
</tr>
<tr>
<td><strong>target</strong></td>
<td>Declares the name of the primary key column for the referenced table. For columns contained in join elements, this is the name of the primary key column in the primary table. For columns contained in field, element, key, value, or array elements, this is the name of the primary key column of the primary table of the other side of the relationship.</td>
<td>target column name</td>
</tr>
<tr>
<td><strong>target-field</strong></td>
<td>Declares the name of the primary key field for the referenced class. For columns contained in join elements, this is the name of the primary key field in the base class. For columns contained in field, element, key, value, or array elements, this is the name of the primary key field of the base class of the other side of the relationship.</td>
<td>target field name</td>
</tr>
<tr>
<td><strong>insert-value</strong></td>
<td>Value to use for this column when it has no field in the class and an object is being inserted. If you want the inserted value to be NULL, then put this as &quot;#NULL&quot;.</td>
<td>Insert value</td>
</tr>
<tr>
<td><strong>position</strong></td>
<td>Position of the column in the table (0 = first).</td>
<td>positive integer</td>
</tr>
</tbody>
</table>

These are attributes within the `<extension>` tag (*/column/extension).
enum-check-constraint
Specifies that a CHECK constraint for this column must be generated based on the values of a java.lang.Enum type, e.g. enum Color (RED, GREEN, BLUE) where its name is persisted a CHECK constraint is defined as CHECK "COLUMN" IN ('RED', 'GREEN', 'BLUE').

true | false

70.1.20 Metadata for join tag
These are attributes within the <join> tag (jdo/package/class/field/join). This element is added when the field has a mapping to a "join" table (as part of a 1-N relationship). It is also used to specify overriding of details in an inheritance tree where the primary key columns are shared up the hierarchy. A further use (when specified under the <class> element) is for specifying the column details for joining to a Secondary Table.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>column</td>
<td>Name of the column used to join to the PK of the primary table (when only one column used). Used in Secondary Tables.</td>
<td></td>
</tr>
<tr>
<td>table</td>
<td>Table name used when joining the PK of a FCO class table to a secondary table. See Secondary Tables.</td>
<td></td>
</tr>
<tr>
<td>delete-action</td>
<td>Action to be performed when the owner object is deleted. This is to be specified when defining foreign key information</td>
<td>cascade</td>
</tr>
<tr>
<td>indexed</td>
<td>Whether the join table owner column should be indexed. This is to be specified when defining index information</td>
<td>true</td>
</tr>
<tr>
<td>unique</td>
<td>Whether the join table owner column should be unique. This is to be specified when defining unique key information</td>
<td>true</td>
</tr>
<tr>
<td>outer</td>
<td>Whether to use an outer join here. This is of particular relevance to secondary tables.</td>
<td>true</td>
</tr>
</tbody>
</table>

These are attributes within the <extension> tag (jdo/package/class/field/join/extension). These are for controlling the join table.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Extension (JDO) Tags**

**primary-key**

This parameter defines if the join table will be assigned a primary key. The default is true since it is considered a best practice to have primary keys on all tables. This allows the option of turning it off.

| true | false |

---

**70.1.21 Metadata for element tag**

These are attributes within the `<element>` tag (jdo/package/class/field/element). This element is added when the field has a mapping to a "element" (as part of a 1-N relationship).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapped-by</td>
<td>The name of the field at the other (&quot;N&quot;) end of a relationship when this field is the &quot;1&quot; side of a 1-N relationship (for FK relationships). This performs the same function as specifying &quot;mapped-by&quot; on the <code>&lt;field&gt;</code> element.</td>
<td></td>
</tr>
<tr>
<td>column</td>
<td>Name of the column (alternative to specifying it as a sub-element).</td>
<td></td>
</tr>
<tr>
<td>delete-action</td>
<td>Action to be performed when the owner object is deleted. This is to be specified when defining foreign key information.</td>
<td>cascade</td>
</tr>
<tr>
<td>indexed</td>
<td>Whether the element column should be indexed. This is to be specified when defining index information.</td>
<td>true</td>
</tr>
<tr>
<td>unique</td>
<td>Whether the element column should be unique. This is to be specified when defining unique key information.</td>
<td>true</td>
</tr>
</tbody>
</table>

---

**70.1.22 Metadata for collection tag**

These are attributes within the `<collection>` tag (jdo/package/class/field/collection). This is used to define the persistence of a Collection.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard (JDO) Tags</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>element-type</td>
<td>The type of element stored in this Collection or array (fully qualified class). This is not required when the field is an array. It is also not required when the Collection is defined using JDK 1.5 generics.</td>
<td></td>
</tr>
<tr>
<td>embedded-element</td>
<td>Whether the elements of a collection or array-valued persistent field should be stored embedded or as first-class objects. It's a hint for the JDO implementation to store, if possible, the elements of the collection as part of the it instead of as their own instances in the datastore. See the &lt;embedded&gt; element for details on how to define the field mappings for the embedded element.</td>
<td>true, false</td>
</tr>
<tr>
<td>dependent-element</td>
<td>Whether the elements of the collection are to be considered dependent on the owner object.</td>
<td>true, false</td>
</tr>
<tr>
<td>serialized-element</td>
<td>Whether the elements of a collection or array-valued persistent field should be stored serialised into a single column of the join table (where used).</td>
<td>true, false</td>
</tr>
</tbody>
</table>

These are attributes within the `<extension>` tag (jdo/package/class/field/collection/extension).

Extension (JDO) Tags

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache</td>
<td>Whether this SCO collection will be cached by DataNucleus or whether every access of the collection will go through to the datastore. See also &quot;datanucleus.cache.collections&quot; in the Persistence Properties Guide. This MetaData attribute is used to override the value used by the PersistenceManagerFactory</td>
<td>true</td>
</tr>
<tr>
<td>cache-lazy-loading</td>
<td>Whether objects from this SCO collection will be lazy loaded (loaded when required) or whether they should be loaded at initialisation. See also &quot;datanucleus.cache.collections.lazy&quot; in the Persistence Properties Guide. This MetaData attribute is used to override the value used by the PersistenceManagerFactory</td>
<td>true</td>
</tr>
</tbody>
</table>
**comparator-name**
Defines the name of the comparator to use with SortedSet, TreeSet collections. The specified name is the name of the comparator class, which must have a default constructor. This extension is only used by SortedSet, TreeSet fields.

---

### 70.1.23 Metadata for map tag

These are attributes within the `<map>` tag (jdo/package/class/field/map). This is used to define the persistence of a Map.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard (JDO) Tags</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>key-type</td>
<td>The type of key stored in this Map (fully qualified class). This is not required when the Map is defined using JDK 1.5 generics.</td>
<td></td>
</tr>
<tr>
<td>embedded-key</td>
<td>Whether the elements of a Map key field should be stored embedded or as first-class objects.</td>
<td>true, false</td>
</tr>
<tr>
<td>value-type</td>
<td>The type of value stored in this Map (fully qualified class). This is not required when the Map is defined using JDK 1.5 generics.</td>
<td>true, false</td>
</tr>
<tr>
<td>embedded-value</td>
<td>Whether the elements of a Map value field should be stored embedded or as first-class objects.</td>
<td>true, false</td>
</tr>
<tr>
<td>dependent-key</td>
<td>Whether the keys of the map are to be considered dependent on the owner object.</td>
<td>true, false</td>
</tr>
<tr>
<td>dependent-value</td>
<td>Whether the value of the map are to be considered dependent on the owner object.</td>
<td>true, false</td>
</tr>
<tr>
<td>serialized-key</td>
<td>Whether the keys of a map-valued persistent field should be stored serialised into a single column of the join table (where used).</td>
<td>true, false</td>
</tr>
<tr>
<td>serialized-value</td>
<td>Whether the values of a map-valued persistent field should be stored serialised into a single column of the join table (where used).</td>
<td>true, false</td>
</tr>
</tbody>
</table>

These are attributes within the `<extension>` tag (jdo/package/class/field/map/extension).
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Extension (JDO) Tags</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cache</td>
<td>Whether this SCO map will be cached by DataNucleus or whether every access of the map will go through to the datastore. See also &quot;datanucleus.cache.collections&quot; in the Persistence Properties Guide. This MetaData attribute is used to override the value used by the PersistenceManagerFactory</td>
<td>true</td>
</tr>
<tr>
<td>cache-lazy-loading</td>
<td>Whether objects from this SCO map will be lazy loaded (loaded when required) or whether they should be loaded at initialisation. See also &quot;datanucleus.cache.collections.lazy&quot; in the Persistence Properties Guide. This MetaData attribute is used to override the value used by the PersistenceManagerFactory</td>
<td>true</td>
</tr>
<tr>
<td>comparator-name</td>
<td>Defines the name of the comparator to use with SortedMap, TreeMap maps. The specified name is the name of the comparator class, which must have a default constructor. This extension is only used by SortedMap, TreeMap fields.</td>
<td>Fully-qualified class name</td>
</tr>
</tbody>
</table>

### 70.1.24 Metadata for array tag

This is used to define the persistence of an array. DataNucleus provides support for many types of arrays, either serialised into a single column, using a join table, or via a foreign-key (for arrays of PC objects).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard (JDO) Tags</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>embedded-element</td>
<td>Whether the array elements should be stored embedded (default = true for primitives, wrappers etc and false for PersistenceCapable objects).</td>
<td>true, false</td>
</tr>
<tr>
<td>serialized-element</td>
<td>Whether the array elements should be stored serialised into a single column in the join table.</td>
<td>true, false</td>
</tr>
<tr>
<td>dependent-element</td>
<td>Whether the elements of the array are to be considered dependent on the owner object.</td>
<td>true, false</td>
</tr>
</tbody>
</table>
70.1.25 Metadata for sequence tag

These are attributes within the `<sequence>` tag. This is used to denote a JDO datastore sequence.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard (JDO) Tags</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>Symbolic name for the sequence for this package</td>
<td></td>
</tr>
<tr>
<td>datastore-sequence</td>
<td>Name of the sequence in the datastore</td>
<td></td>
</tr>
<tr>
<td>factory-class</td>
<td>Factory class for creating the sequence. Please refer to the Sequence guide</td>
<td></td>
</tr>
<tr>
<td>strategy</td>
<td>Strategy to use for application of this sequence.</td>
<td>nontransactional</td>
</tr>
<tr>
<td>allocation-size</td>
<td>Allocation size for the sequence for this package</td>
<td>50</td>
</tr>
<tr>
<td>initial-value</td>
<td>Initial value for the sequence for this package</td>
<td>1</td>
</tr>
</tbody>
</table>

These are attributes within the `<extension>` tag (jdo/package/class/sequence/extension). These are for controlling the datastore sequences created by DataNucleus. Please refer to the documentation for the value generator being used for applicability.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Extension (JDO) Tags</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sequence-catalog-name</td>
<td>The catalog used to store sequences for use by value generators. See Value Generation. Default catalog for the datastore will be used if not specified.</td>
<td></td>
</tr>
<tr>
<td>sequence-schema-name</td>
<td>The schema used to store sequences for use by value generators. See Value Generation. Default schema for the datastore will be used if not specified.</td>
<td></td>
</tr>
<tr>
<td>sequence-table-name</td>
<td>The table used to store sequences for use by value generators. See Value Generation.</td>
<td><strong>SEQUENCE_TABLE</strong></td>
</tr>
<tr>
<td>sequence-name-column-name</td>
<td>The column name in the sequence-table used to store the name of the sequence for use by value generators. See Value Generation.</td>
<td><strong>SEQUENCE_NAME</strong></td>
</tr>
</tbody>
</table>
sequence-nextval-column-name: The column name in the sequence
  table used to store the next value
  in the sequence for use by value
  generators. See Value Generation.

key-min-value: The minimum key value for use
  by value generators. Keys lower
  than this will not be generated. See
  Value Generation.

key-max-value: The maximum key value for use
  by value generators. Keys higher
  than this will not be generated. See
  Value Generation.

key-initial-value: The starting value for use by value
  generators. Keys will start from this
  value when being generated. See
  Value Generation.

key-cache-size: The cache size for keys for use
  by value generators. The cache
  of keys will be constrained by this
  value. See Value Generation.

key-database-cache-size: The database cache size for
  keys for use by value generators.
  The cache of keys will be
  constrained by this value. See
  Value Generation.

70.1.26 Metadata for fetch-plan tag
These are attributes within the <fetch-plan> tag (jdo/fetch-plan). This element is used to define fetch
plans that are utilised at runtime, and are of particular use with queries. This element contains fetch-
group sub-elements.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard (JDO) Tags</td>
<td></td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>Name of the fetch plan.</td>
<td></td>
</tr>
<tr>
<td>maxFetchDepth</td>
<td>Max depth to fetch with this fetch plan</td>
<td>1</td>
</tr>
<tr>
<td>fetchSize</td>
<td>Size to fetch with this fetch plan (for use with query result sets)</td>
<td>0</td>
</tr>
</tbody>
</table>

70.1.27 Metadata for class extension tag
These are attributes within the <extension> tag (jdo/package/class/extension). These are for
controlling the class definition

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
</table>
### Extension (JDO) Tags

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>requires-table</td>
<td>This is for use with a &quot;nondurable&quot; identity case and specifies whether the class requires a table/view in the datastore.</td>
<td>true</td>
</tr>
<tr>
<td>ddl-definition</td>
<td>Definition of the TABLE SCHEMA to be used by the class.</td>
<td>true</td>
</tr>
<tr>
<td>ddl-imports</td>
<td>Classes imported resolve macro identifiers in the definition of a RDBMS Table.</td>
<td></td>
</tr>
<tr>
<td>mysql-engine-type</td>
<td>&quot;Engine Type&quot; to use when creating the table for this class in MySQL. Refer to the MySQL documentation for ENGINE type (e.g INNODB, MEMORY, ISAM)</td>
<td></td>
</tr>
<tr>
<td>view-definition</td>
<td>Definition of the VIEW to be used by the class. Please refer to the RDBMS Views Guide for details. If your view already exists, then specify this as &quot; &quot; and have the autoStart flags set to false.</td>
<td></td>
</tr>
<tr>
<td>view-imports</td>
<td>Classes imported resolve macro identifiers in the definition of a RDBMS View. Please refer to the RDBMS Views Guide for details.</td>
<td></td>
</tr>
<tr>
<td>read-only</td>
<td>Whether objects of this type are read-only. Setting this to true will prevent any insert/update/delete of this type</td>
<td>true</td>
</tr>
</tbody>
</table>

### 70.1.28 Metadata for extension tag

These are attributes within the `<extension>` tag. This is used to denote a DataNucleus extension to JDO.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vendor-name</td>
<td>Name of the vendor. For DataNucleus we use the name &quot;datanucleus&quot; (lowercase).</td>
</tr>
<tr>
<td>key</td>
<td>Key of the extension property</td>
</tr>
<tr>
<td>value</td>
<td>Value of the extension property</td>
</tr>
</tbody>
</table>
71 Annotations

71.1 JDO : Annotations

One of the things that JDK 1.5 provides that can be of some use is annotations, and JDO provides its own set. When selecting to use annotations please bear in mind the following:

- You must have the datanucleus-api-jdo jar available in your CLASSPATH.
- You must have the jdo-api jar in your CLASSPATH since this provides the annotations
- Annotations should really only be used for attributes of persistence that you won't be changing at deployment. Things such as table and column names shouldn't really be specified using annotations although it is permitted. Instead it would be better to put such information in an ORM MetaData file.
- Annotations can be added in two places - for the class as a whole, or for a field in particular.
- You can annotate fields or getters with field-level information. If you annotate fields then the fields are processed for persistence. If you annotate the methods (getters) then the methods (properties) are processed for persistence.
- Annotations are prefixed by the @ symbol and can take properties (in brackets after the name, comma-separated)

Annotations supported by DataNucleus are shown below. The annotations/attributes coloured in pink are ORM and really should be placed in XML rather than directly in the class using annotations.

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Class/Field/Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@PersistenceCapable</td>
<td>Class</td>
<td>Specifies that the class/interface is persistent. In the case of an interface this would utilise JDO2's &quot;persistent-interface&quot; capabilities</td>
</tr>
<tr>
<td>@PersistenceAware</td>
<td>Class</td>
<td>Specifies that the class is not persistent but needs to be able to access fields of persistent classes</td>
</tr>
<tr>
<td>@Cacheable</td>
<td>Class</td>
<td>Specifies whether this class can be cached in a Level 2 cache or not.</td>
</tr>
<tr>
<td>@EmbeddedOnly</td>
<td>Class</td>
<td>Specifies that the class is persistent and can only be persisted embedded in another persistent class</td>
</tr>
<tr>
<td>@DatastoreIdentity</td>
<td>Class</td>
<td>Specifies the details for generating datastore-identifier for this class</td>
</tr>
<tr>
<td>@Version</td>
<td>Class</td>
<td>Specifies any versioning process for objects of this class</td>
</tr>
<tr>
<td>@FetchPlans</td>
<td>Class</td>
<td>Defines a series of fetch plans</td>
</tr>
<tr>
<td>@FetchPlan</td>
<td>Class</td>
<td>Defines a fetch plan</td>
</tr>
<tr>
<td>@FetchGroups</td>
<td>Class</td>
<td>Defines a series of fetch groups for this class</td>
</tr>
<tr>
<td>@FetchGroup</td>
<td>Class</td>
<td>Defines a fetch group for this class</td>
</tr>
<tr>
<td>@Sequence</td>
<td>Class</td>
<td>Defines a sequence for use by this class</td>
</tr>
<tr>
<td>Annotation</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>@Queries</td>
<td>Class</td>
<td>Defines a series of named queries for this class</td>
</tr>
<tr>
<td>@Query</td>
<td>Class</td>
<td>Defines a named query for this class</td>
</tr>
<tr>
<td>@Inheritance</td>
<td>Class</td>
<td>Specifies the inheritance model for persisting this class</td>
</tr>
<tr>
<td>@Discriminator</td>
<td>Class</td>
<td>Specifies any discriminator for this class to be used for determining object types</td>
</tr>
<tr>
<td>@PrimaryKey</td>
<td>ORM Class</td>
<td>Defines the primary key constraint for this class</td>
</tr>
<tr>
<td>@Indices</td>
<td>ORM Class</td>
<td>Defines a series of indices for this class</td>
</tr>
<tr>
<td>@Index</td>
<td>ORM Class</td>
<td>Defines an index for the class as a whole (typically a composite)</td>
</tr>
<tr>
<td>@Uniques</td>
<td>ORM Class</td>
<td>Defines a series of unique constraints for this class</td>
</tr>
<tr>
<td>@Unique</td>
<td>ORM Class</td>
<td>Defines a unique constraint for the class as a whole (typically a composite)</td>
</tr>
<tr>
<td>@ForeignKeys</td>
<td>ORM Class</td>
<td>Defines a series of foreign-keys (typically for non-mapped columns/tables)</td>
</tr>
<tr>
<td>@ForeignKey</td>
<td>ORM Class</td>
<td>Defines a foreign-key for the class as a whole (typically for non-mapped columns/tables)</td>
</tr>
<tr>
<td>@Joins</td>
<td>ORM Class</td>
<td>Defines a series of joins to secondary tables from this table</td>
</tr>
<tr>
<td>@Join</td>
<td>ORM Class</td>
<td>Defines a join to a secondary table from this table</td>
</tr>
<tr>
<td>@Columns</td>
<td>ORM Class</td>
<td>Defines a series of columns that don't have associated fields (&quot;unmapped columns&quot;)</td>
</tr>
<tr>
<td>@Persistent</td>
<td>Field/Method</td>
<td>Defines the persistence for a field/property of the class</td>
</tr>
<tr>
<td>@Serialized</td>
<td>Field/Method</td>
<td>Defines this field as being stored serialised</td>
</tr>
<tr>
<td>@NotPersistent</td>
<td>Field/Method</td>
<td>Defines this field as being not persisted</td>
</tr>
<tr>
<td>@Transactional</td>
<td>Field/Method</td>
<td>Defines this field as being transactional (not persisted, but managed)</td>
</tr>
<tr>
<td>@Cacheable</td>
<td>Field/Method</td>
<td>Specifies whether this field/property can be cached in a Level 2 cache or not.</td>
</tr>
<tr>
<td>@PrimaryKey</td>
<td>Field/Method</td>
<td>Defines this field as being (part of) the primary key</td>
</tr>
</tbody>
</table>
71.1.1 @PersistenceCapable

This annotation is used when you want to mark a class as persistent. It equates to the <class> MetaData element (though with only some of its attributes). Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>requiresExtent</td>
<td>String</td>
<td>Whether an extent is required for this class</td>
<td>true</td>
</tr>
<tr>
<td>embeddedOnly</td>
<td>String</td>
<td>Whether objects of this class can only be stored embedded in other objects</td>
<td>false</td>
</tr>
<tr>
<td>detachable</td>
<td>String</td>
<td>Whether objects of this class can be detached</td>
<td>false</td>
</tr>
<tr>
<td>identityType</td>
<td>IdentityType</td>
<td>Type of identity (APPLICATION, DATASTORE, NONDURABLE)</td>
<td>DATASTORE</td>
</tr>
<tr>
<td>objectIdClass</td>
<td>Class</td>
<td>Object-id class</td>
<td></td>
</tr>
</tbody>
</table>
### Table Mapping

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>String</td>
<td>ORM: Name of the table where this class is persisted</td>
<td></td>
</tr>
<tr>
<td>catalog</td>
<td>String</td>
<td>ORM: Name of the catalog where this table is persisted</td>
<td></td>
</tr>
<tr>
<td>schema</td>
<td>String</td>
<td>ORM: Name of the schema where this table is persisted</td>
<td></td>
</tr>
<tr>
<td>cacheable</td>
<td>String</td>
<td>Whether the class can be L2 cached. <strong>From JDO2.2</strong></td>
<td>true</td>
</tr>
<tr>
<td>serializeRead</td>
<td>String</td>
<td>Whether to default reads of this object type to lock the object</td>
<td>false</td>
</tr>
<tr>
<td>extensions</td>
<td>Extension[]</td>
<td>Vendor extensions</td>
<td></td>
</tr>
</tbody>
</table>

```java
@PersistenceCapable(identityType=IdentityType.APPLICATION)
public class MyClass
{
    ...
}
```

#### 71.1.2 @PersistenceAware

This annotation is used when you want to mark a class as being used in persistence but not being persistable. That is "persistence-aware" in JDO2 terminology. It has no attributes. Specified on the class.

```java
@PersistenceAware
public class MyClass
{
    ...
}
```

See the documentation for [Class Mapping](#).

#### 71.1.3 @Cacheable

This annotation is a shortcut for `@PersistenceCapable(cacheable={value})` specifying whether the class can be cached in a Level 2 cache. Specified on the class. The default

```java
public class MyClass
{
    ...
}
```
71.1.4 @EmbeddedOnly

This annotation is a shortcut for @PersistenceCapable(embeddedOnly="true") meaning that the class can only be persisted embedded into another class. It has no attributes. Specified on the class.

```java
@EmbeddedOnly
public class MyClass {
    ...
}
```

71.1.5 @Inheritance

Annotation used to define the inheritance for a class. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>strategy</td>
<td>InheritanceStrategy</td>
<td>The inheritance strategy (NEW_TABLE, SUBCLASS_TABLE, SUPERCLASS_TABLE)</td>
<td></td>
</tr>
<tr>
<td>customStrategy</td>
<td>String</td>
<td>Name of a custom inheritance strategy (DataNucleus supports &quot;complete-table&quot;)</td>
<td></td>
</tr>
</tbody>
</table>

```java
@PersistenceCapable
@Inheritance(strategy=InheritanceStrategy.NEW_TABLE)
public class MyClass {
    ...
}
```

See the documentation for Inheritance
71.1.6 @Discriminator

Annotation used to define a discriminator to be stored with instances of this class and is used to determine the types of the objects being stored. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>strategy</td>
<td>DiscriminatorStrategy</td>
<td>The discriminator strategy (VALUE_MAP, CLASS_NAME, NONE)</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>String</td>
<td>Value to use for instances of this type when using strategy of VALUE_MAP</td>
<td></td>
</tr>
<tr>
<td>column</td>
<td>String</td>
<td>ORM : Name of the column to use to store the discriminator</td>
<td></td>
</tr>
<tr>
<td>indexed</td>
<td>String</td>
<td>ORM : Whether the discriminator column is to be indexed</td>
<td></td>
</tr>
<tr>
<td>columns</td>
<td>Column[]</td>
<td>ORM : Column definitions used for storing the discriminator</td>
<td></td>
</tr>
</tbody>
</table>

```java
@PersistenceCapable
@Inheritance(strategy=InheritanceStrategy.NEW_TABLE)
@Discriminator(strategy=DiscriminatorStrategy.CLASS_NAME)
public class MyClass
{
    ...
}
```

71.1.7 @DatastoreIdentity

Annotation used to define the identity when using datastore-identity for the class. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>strategy</td>
<td>IdGeneratorStrategy</td>
<td>The inheritance strategy (NATIVE, SEQUENCE, IDENTITY, INCREMENT, UUIDSTRING, UUIDHEX)</td>
<td></td>
</tr>
<tr>
<td>customStrategy</td>
<td>String</td>
<td>Name of a custom id generation strategy (e.g &quot;max&quot;, &quot;auid&quot;). This overrides the value of &quot;strategy&quot;</td>
<td></td>
</tr>
</tbody>
</table>
### 71.1.8 @Version

Annotation used to define the versioning details for use with optimistic transactions. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>strategy</td>
<td>VersionStrategy</td>
<td>The version strategy (NONE, STATE_IMAGE, DATE_TIME, VERSION_NUMBER)</td>
<td></td>
</tr>
<tr>
<td>indexed</td>
<td>String</td>
<td>Whether the version column(s) is indexed</td>
<td></td>
</tr>
<tr>
<td>column</td>
<td>String</td>
<td>ORM : Name of the column for the version</td>
<td></td>
</tr>
<tr>
<td>columns</td>
<td>Column[]</td>
<td>ORM : Column definition for the column(s) for the version</td>
<td></td>
</tr>
<tr>
<td>extensions</td>
<td>Extension[]</td>
<td>Vendor extensions</td>
<td></td>
</tr>
</tbody>
</table>

```java
@PersistenceCapable
@Version(strategy=VersionStrategy.VERSION_NUMBER)
public class MyClass
{
    ...
}
```

See the documentation for [Datastore Identity](#)
See the documentation for Optimistic Transactions

### 71.1.9 @PrimaryKey
Annotation used to define the primary key constraint for a class. Maps across to the `<primary-key>` MetaData element. Specified on the `class`.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>ORM: Name of the primary key constraint</td>
<td></td>
</tr>
<tr>
<td>column</td>
<td>String</td>
<td>ORM: Name of the column for this key</td>
<td></td>
</tr>
<tr>
<td>columns</td>
<td>Column[]</td>
<td>ORM: Column definition for the column(s) of this key</td>
<td></td>
</tr>
</tbody>
</table>

```java
@PersistenceCapable
@PrimaryKey(name="MYCLASS_PK")
public class MyClass
{
    ...
}
```

### 71.1.10 @FetchPlans
Annotation used to define a set of fetch plans. Specified on the `class`. Used by named queries

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>FetchPlan[]</td>
<td>Array of fetch plans - see @FetchPlan annotation</td>
<td></td>
</tr>
</tbody>
</table>

```java
@PersistenceCapable
@FetchPlans({@FetchPlan(name="plan_3", maxFetchDepth=3, fetchGroups={"group1", "group4"}),
             @FetchPlan(name="plan_4", maxFetchDepth=2, fetchGroups={"group1", "group2"}))
public class MyClass
{
    ...
}
```

See the documentation for FetchGroups
**71.1.11 @FetchPlan**

Annotation used to define a fetch plan is equivalent to the `<fetch-plan>` metadata element. Specified on the `class`. Used by named queries.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name of the FetchPlan</td>
<td></td>
</tr>
<tr>
<td>maxFetchDepth</td>
<td>int</td>
<td>Maximum fetch depth</td>
<td>1</td>
</tr>
<tr>
<td>fetchSize</td>
<td>int</td>
<td>Size hint for fetching query result sets</td>
<td>0</td>
</tr>
<tr>
<td>fetchGroups</td>
<td>String[]</td>
<td>Names of the fetch groups included in this FetchPlan.</td>
<td></td>
</tr>
</tbody>
</table>

```java
@PersistenceCapable
@FetchPlan(name="plan_3", maxFetchDepth=3, fetchGroups={"group1", "group4"})
public class MyClass {
    ...
}
```

See the documentation for `FetchGroups`.

**71.1.12 @FetchGroups**

Annotation used to define a set of fetch groups for a class. Specified on the `class`.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>FetchGroup[]</td>
<td>Array of fetch groups</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- see @FetchGroup annotation</td>
<td></td>
</tr>
</tbody>
</table>
@PersistenceCapable
@FetchGroups({@FetchGroup(name="one_two", members=[@Persistent(name="field1"), @Persistent(name="field2")]),
@FetchGroup(name="three", members=[@Persistent(name="field3")])})
public class MyClass {
    @Persistent
    String field1;

    @Persistent
    String field2;

    @Persistent
    String field3;
    ...
}

See the documentation for FetchGroups

71.1.13 @FetchGroup
Annotation used to define a fetch group. Is equivalent to the <fetch-group> metadata element. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name of the fetch group</td>
<td></td>
</tr>
<tr>
<td>postLoad</td>
<td>String</td>
<td>Whether to call jdoPostLoad after loading this fetch group</td>
<td></td>
</tr>
<tr>
<td>members</td>
<td>Persistent[]</td>
<td>Definitions of the fields/properties to include in this fetch group</td>
<td></td>
</tr>
</tbody>
</table>

@PersistenceCapable
@FetchGroup(name="one_two", members=[@Persistent(name="field1"), @Persistent(name="field2"), @Persistent(name="field3")])
public class MyClass {
    @Persistent
    String field1;

    @Persistent
    String field2;
    ...
}

See the documentation for FetchGroups
71.1.14 @Sequence

Annotation used to define a sequence generator. Is equivalent to the `<sequence>` metadata element. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name of the sequence</td>
<td></td>
</tr>
<tr>
<td>strategy</td>
<td>SequenceStrategy</td>
<td>Strategy for the sequence (NONTRANSACTIONAL, CONTIGUOUS, NONCONTIGUOUS)</td>
<td></td>
</tr>
<tr>
<td>datastoreSequence</td>
<td>String</td>
<td>Name of a datastore sequence that this maps to</td>
<td></td>
</tr>
<tr>
<td>factoryClass</td>
<td>Class</td>
<td>Factory class to use to generate the sequence</td>
<td></td>
</tr>
<tr>
<td>initialValue</td>
<td>int</td>
<td>Initial value of the sequence</td>
<td>1</td>
</tr>
<tr>
<td>allocationSize</td>
<td>int</td>
<td>Allocation size of the sequence</td>
<td>50</td>
</tr>
<tr>
<td>extensions</td>
<td>Extension[]</td>
<td>Vendor extensions</td>
<td></td>
</tr>
</tbody>
</table>

See the documentation for Sequences

71.1.15 @Queries

Annotation used to define a set of named queries for a class. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Query[]</td>
<td>Array of queries - see @Query annotation</td>
<td></td>
</tr>
</tbody>
</table>

```java
@PersistenceCapable
@Queries({
    @Query(name="PeopleCalledSmith", language="JDOQL",
    value="SELECT FROM org.datanucleus.samples.Person WHERE surname == \"Smith\""),
    @Query(name="PeopleCalledJones", language="JDOQL",
    value="SELECT FROM org.datanucleus.samples.Person WHERE surname == \"Jones\""))
}
public class Person {
    @Persistent
    String surname;
    ...
}
```

See the documentation for Named Queries
### 71.1.16 @Query

Annotation used to define a named query. Is equivalent to the `<query>` metadata element. Specified on the **class**.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name of the query</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>String</td>
<td>The query string itself</td>
<td></td>
</tr>
<tr>
<td>language</td>
<td>String</td>
<td>Language of the query</td>
<td>JDOQL</td>
</tr>
<tr>
<td>unmodifiable</td>
<td>String</td>
<td>Whether the query is not modifiable at runtime</td>
<td></td>
</tr>
<tr>
<td>unique</td>
<td>String</td>
<td>Whether the query returns unique results (for SQL queries only)</td>
<td></td>
</tr>
<tr>
<td>resultClass</td>
<td>Class</td>
<td>Result class to use (for SQL queries only)</td>
<td></td>
</tr>
<tr>
<td>fetchPlan</td>
<td>String</td>
<td>Name of a named FetchPlan to use with this query</td>
<td></td>
</tr>
<tr>
<td>extensions</td>
<td>Extension[]</td>
<td>Vendor extensions</td>
<td></td>
</tr>
</tbody>
</table>

```java
@PersistenceCapable
@Query(name="PeopleCalledSmith", language="JDOQL",
       value="SELECT FROM org.datanucleus.samples.Person WHERE surname == \"Smith\""
)
public class Person
{
    @Persistent
    String surname;
    ...
}
```

See the documentation for **Named Queries**

### 71.1.17 @Indices

Annotation used to define a set of indices for a class. Specified on the **class**.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Index[]</td>
<td>Array of indices - see @Index annotation</td>
<td></td>
</tr>
</tbody>
</table>
@PersistenceCapable
@Indices({@Index(name="MYINDEX_1", members={"field1","field2"}), @Index(name="MYINDEX_2", members={"field3"})})
public class Person
{
    ...
}

See the documentation for Schema Constraints

71.1.18 @Index
Annotation used to define an index for the class as a whole typically being a composite index across multiple columns or fields/properties. Is equivalent to the <index> metadata element when specified under class. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>ORM : Name of the index</td>
<td></td>
</tr>
<tr>
<td>table</td>
<td>String</td>
<td>ORM : Name of the tablefor the index</td>
<td></td>
</tr>
<tr>
<td>unique</td>
<td>String</td>
<td>ORM : Whether the indexis unique</td>
<td></td>
</tr>
<tr>
<td>members</td>
<td>String[]</td>
<td>ORM : Names of thefields/properties thatmake up this index</td>
<td></td>
</tr>
<tr>
<td>columns</td>
<td>Column[]</td>
<td>ORM : Columns thatmake up this index</td>
<td></td>
</tr>
</tbody>
</table>

@PersistenceCapable
@Index(name="MY_COMPOSITE_IDX", members={"field1", "field2"})
public class MyClass
{
    @Persistent
    String field1;

    @Persistent
    String field2;

    ...
}

See the documentation for Schema Constraints

71.1.19 @Uniques
Annotation used to define a set of unique constraints for a class. Specified on the class.
### @Unique

Annotation used to define a unique constraint for the class as a whole typically being a composite constraint across multiple columns or fields/properties. Is equivalent to the `<unique>` metadata element when specified under class. Specified on the `class`.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>ORM: Name of the constraint</td>
<td></td>
</tr>
<tr>
<td>table</td>
<td>String</td>
<td>ORM: Name of the table for the constraint</td>
<td></td>
</tr>
<tr>
<td>deferred</td>
<td>String</td>
<td>ORM: Whether the constraint is deferred</td>
<td></td>
</tr>
<tr>
<td>members</td>
<td>String[]</td>
<td>ORM: Names of the fields/properties that make up this constraint</td>
<td></td>
</tr>
<tr>
<td>columns</td>
<td>Column[]</td>
<td>ORM: Columns that make up this constraint</td>
<td></td>
</tr>
</tbody>
</table>

```java
@PersistenceCapable
@Unique(name="MY_COMPOSITE_IDX", members="field1", "field2")
public class MyClass
{
    @Persistent
    String field1;

    @Persistent
    String field2;

    ...
}
```
See the documentation for Schema Constraints

71.1.21 @ForeignKeys
Annotation used to define a set of foreign-key constraints for a class. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>ForeignKey[]</td>
<td>Array of FK constraints</td>
<td>- see @ForeignKey annotation</td>
</tr>
</tbody>
</table>

See the documentation for Schema Constraints

71.1.22 @ForeignKey
Annotation used to define a foreign-key constraint for the class. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>ORM : Name of the constraint</td>
<td></td>
</tr>
<tr>
<td>table</td>
<td>String</td>
<td>ORM : Name of the table that the FK is to</td>
<td></td>
</tr>
<tr>
<td>deferred</td>
<td>String</td>
<td>ORM : Whether the constraint is deferred</td>
<td></td>
</tr>
<tr>
<td>unique</td>
<td>String</td>
<td>ORM : Whether the constraint is unique</td>
<td></td>
</tr>
<tr>
<td>deleteAction</td>
<td>ForeignKeyAction</td>
<td>ORM : Action to apply to the FK to be used on deleting</td>
<td>ForeignKeyAction.RESTRICT</td>
</tr>
<tr>
<td>updateAction</td>
<td>ForeignKeyAction</td>
<td>ORM : Action to apply to the FK to be used on updating</td>
<td>ForeignKeyAction.RESTRICT</td>
</tr>
<tr>
<td>members</td>
<td>String[]</td>
<td>ORM : Names of the fields/properties that compose this FK.</td>
<td></td>
</tr>
<tr>
<td>columns</td>
<td>Column[]</td>
<td>ORM : Columns that compose this FK.</td>
<td></td>
</tr>
</tbody>
</table>

See the documentation for Schema Constraints

71.1.23 @Joins
Annotation used to define a set of joins (to secondary tables) for a class. Specified on the class.
### 71.1.24 @Join

Annotation used to specify a join for a secondary table. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>String</td>
<td>ORM : Table name used when joining the PK of a FCO class table to a secondary table.</td>
<td></td>
</tr>
<tr>
<td>column</td>
<td>String</td>
<td>ORM : Name of the column used to join to the PK of the primary table (when only one column used)</td>
<td></td>
</tr>
<tr>
<td>outer</td>
<td>String</td>
<td>ORM : Whether to use an outer join when retrieving fields/properties stored in the secondary table</td>
<td></td>
</tr>
<tr>
<td>columns</td>
<td>Column[]</td>
<td>ORM : Name of the columns used to join to the PK of the primary table (when multiple columns used)</td>
<td></td>
</tr>
<tr>
<td>extensions</td>
<td>Extension[]</td>
<td>Vendor extensions</td>
<td></td>
</tr>
</tbody>
</table>
71.1.25 @Columns
Annotation used to define the columns which have no associated field in the class. User should specify a minimum of @Column "name", "jdbcType", and "insertValue". Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Column[]</td>
<td>Array of columns - see @Column annotation</td>
<td></td>
</tr>
</tbody>
</table>

```java
@PersistenceCapable
@Columns(@Column(name="MY_OTHER_COL", jdbcType="VARCHAR", insertValue="N/A")
public class MyClass
{
    ...}
```

71.1.26 @Persistent
Annotation used to define the fields/properties to be persisted. Is equivalent to the <field> and <property> metadata elements. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>persistenceModifier</td>
<td>PersistenceModifier</td>
<td>Whether the field is persistent (PERSISTENT, TRANSACTIONAL, NONE)</td>
<td>[depends on field type]</td>
</tr>
<tr>
<td>defaultFetchGroup</td>
<td>String</td>
<td>Whether the field is part of the DFG</td>
<td></td>
</tr>
<tr>
<td>nullValue</td>
<td>NullValue</td>
<td>Required behaviour when inserting a null value for this field (NONE, EXCEPTION, DEFAULT)</td>
<td>NONE</td>
</tr>
<tr>
<td>Field</td>
<td>Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>embedded</td>
<td>String</td>
<td>Whether this field as a whole is embedded. Use @Embedded to specify details.</td>
<td></td>
</tr>
<tr>
<td>embeddedElement</td>
<td>String</td>
<td>Whether the element stored in this collection/array field/property is embedded</td>
<td></td>
</tr>
<tr>
<td>embeddedKey</td>
<td>String</td>
<td>Whether the key stored in this map field/property is embedded</td>
<td></td>
</tr>
<tr>
<td>embeddedValue</td>
<td>String</td>
<td>Whether the value stored in this map field/property is embedded</td>
<td></td>
</tr>
<tr>
<td>serialized</td>
<td>String</td>
<td>Whether this field/property as a whole is serialised</td>
<td></td>
</tr>
<tr>
<td>serializedElement</td>
<td>String</td>
<td>Whether the element stored in this collection/array field/property is serialised</td>
<td></td>
</tr>
<tr>
<td>serializedKey</td>
<td>String</td>
<td>Whether the key stored in this map field/property is serialised</td>
<td></td>
</tr>
<tr>
<td>serializedValue</td>
<td>String</td>
<td>Whether the value stored in this map field/property is serialised</td>
<td></td>
</tr>
<tr>
<td>dependent</td>
<td>String</td>
<td>Whether this field is dependent, deleting the related object when deleting this object</td>
<td></td>
</tr>
<tr>
<td>dependentElement</td>
<td>String</td>
<td>Whether the element stored in this field/property is dependent</td>
<td></td>
</tr>
<tr>
<td>dependentKey</td>
<td>String</td>
<td>Whether the key stored in this field/property is dependent</td>
<td></td>
</tr>
<tr>
<td>dependentValue</td>
<td>String</td>
<td>Whether the value stored in this field/property is dependent</td>
<td></td>
</tr>
<tr>
<td>primaryKey</td>
<td>String</td>
<td>Whether this field is (part of) the primary key</td>
<td></td>
</tr>
<tr>
<td>valueStrategy</td>
<td>IdGeneratorStrategy</td>
<td>Strategy to use when generating values for the field</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(NATIVE, SEQUENCE, IDENTITY, INCREMENT, UUIDSTRING, UUIDHEX)</td>
<td></td>
</tr>
<tr>
<td>customValueStrategy</td>
<td>String</td>
<td>Name of a custom id generation strategy (e.g “max”, “auid”). This overrides the value of “valueStrategy”</td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td>Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>sequence</td>
<td>String</td>
<td>Name of the sequence when using valueStrategy of SEQUENCE - refer to @Sequence</td>
<td></td>
</tr>
<tr>
<td>types</td>
<td>Class[]</td>
<td>Type(s) of field (when using interfaces/reference types). DataNucleus currently only supports the first value although in the future it is hoped to support multiple.</td>
<td></td>
</tr>
<tr>
<td>mappedBy</td>
<td>String</td>
<td>Field in other class when the relation is bidirectional to signify the owner of the relation</td>
<td></td>
</tr>
<tr>
<td>table</td>
<td>String</td>
<td>ORM : Name of the table where this field is persisted. If this field is a collection/map/array then the table refers to a join table, otherwise this refers to a secondary table.</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>Name of the field when defining an embedded field.</td>
<td></td>
</tr>
<tr>
<td>columns</td>
<td>Column[]</td>
<td>ORM : Column definition(s) for the columns into which this field is persisted. This is only typically used when specifying columns of a field of an embedded class.</td>
<td></td>
</tr>
<tr>
<td>cacheable</td>
<td>String</td>
<td>Whether the field/property can be L2 cached. From JDO2.2</td>
<td></td>
</tr>
<tr>
<td>extensions</td>
<td>Extension[]</td>
<td>Vendor extensions</td>
<td></td>
</tr>
<tr>
<td>recursionDepth</td>
<td>int</td>
<td>Recursion depth for this field when fetching. Only applicable when specified within @FetchGroup</td>
<td></td>
</tr>
<tr>
<td>loadFetchGroup</td>
<td>String</td>
<td>Name of a fetch group to activate when a load of this field is initiated (due to it being currently unloaded). Not used for getObjectById, queries, extents etc. Better to use @FetchGroup and define your groups</td>
<td></td>
</tr>
</tbody>
</table>
71 Annotations

```java
@PersistenceCapable
public class MyClass {
    @Persistent(primaryKey="true")
    String myField;
    ...
}
```

See the documentation for [Fields/Properties](#)

### 71.1.27 @Serialized

This annotation is a shortcut for `@Persistent(serialized="true")` meaning that the field is stored serialized. It has no attributes. Specified on the field/method.

```java
@PersistenceCapable
public class MyClass {
    @Serialized
    Object myField;
    ...
}
```

See the documentation for [Serialising](#)

### 71.1.28 @NotPersistent

This annotation is a shortcut for `@Persistent(persistenceModifier=PersistenceModifier.NONE)` meaning that the field/property is not persisted. It has no attributes. Specified on the field/method.

```java
@PersistenceCapable
public class MyClass {
    @NotPersistent
    String myOtherField;
    ...
}
```

See the documentation for [Fields/Properties](#)

### 71.1.29 @Transactional

This annotation is a shortcut for `@Persistent(persistenceModifier=PersistenceModifier.TRANSACTIONAL)` meaning that the field/property is not persisted yet managed. It has no attributes. Specified on the field/method.
See the documentation for Fields/Properties

### 71.1.30 @Cacheable

This annotation is a shortcut for `@Persistent(cacheable={value})` specifying whether the field/property can be cached in a Level 2 cache. Specified on the field/property. The default

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>String</td>
<td>Whether the field/property is cacheable</td>
<td>true</td>
</tr>
</tbody>
</table>

```java
public class MyClass {
    @Cacheable("false")
    Collection elements;
    ...
}
```

See the documentation for L2 Caching

### 71.1.31 @PrimaryKey

This annotation is a shortcut for `@Persistent(primaryKey="true")` meaning that the field/property is part of the primary key for the class. No attributes are needed when specified like this. Specified on the field/method.

```java
@PersistenceCapable
public class MyClass {
    @PrimaryKey
    String myOtherField;
    ...
}
```

See the documentation for Schema Constraints
### 71.1.32 @Element

Annotation used to define the element for any collection/array to be persisted. Maps across to the `<collection>`, `<array>` and `<element>` MetaData elements. Specified on the `field/method`.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>types</td>
<td>Class[]</td>
<td>Type(s) of element. While the attribute allows multiple values</td>
<td>When using an array is not needed. When using a collection will</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DataNucleus currently only supports the first type value</td>
<td>be taken from the collection definition if using generics, otherwise</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>must be specified.</td>
</tr>
<tr>
<td>embedded</td>
<td>String</td>
<td>Whether the element is embedded into a join table</td>
<td></td>
</tr>
<tr>
<td>serialized</td>
<td>String</td>
<td>Whether the element is serialised into the join table</td>
<td></td>
</tr>
<tr>
<td>dependent</td>
<td>String</td>
<td>Whether the element objects are dependent when deleting the owner</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>collection/array</td>
<td></td>
</tr>
<tr>
<td>mappedBy</td>
<td>String</td>
<td>Field in the element class that represents this object (when the relation is</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bidirectional)</td>
<td></td>
</tr>
<tr>
<td>embeddedMapping</td>
<td>Embedded[]</td>
<td>Definition of any embedding of the (persistable) element. Only 1 &quot;Embedded&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>should be provided</td>
<td></td>
</tr>
<tr>
<td>table</td>
<td>String</td>
<td>ORM : Name of the table for this element</td>
<td></td>
</tr>
<tr>
<td>column</td>
<td>String</td>
<td>ORM : Name of the column for this element</td>
<td></td>
</tr>
<tr>
<td>foreignKey</td>
<td>String</td>
<td>ORM : Name of any foreign-key constraint to add</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>generateForeignKey</td>
<td>String</td>
<td>ORM : Whether to generate a FK constraint for the element (when not</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>specifying the name)</td>
<td></td>
</tr>
<tr>
<td>deleteAction</td>
<td>ForeignKeyAction</td>
<td>ORM : Action to be applied to the foreign key for this element for action</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>upon deletion</td>
<td></td>
</tr>
<tr>
<td>updateAction</td>
<td>ForeignKeyAction</td>
<td>ORM : Action to be applied to the foreign key for this element for action</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>upon update</td>
<td></td>
</tr>
</tbody>
</table>
@PersistenceCapable
public class MyClass
{
    @Element(types=org.datanucleus.samples.MyElementClass.class, dependent="true")
    Collection myField;
    ...
}

71.1.33 @Order
Annotation used to define the ordering of an order-based Collection/array to be persisted. Maps across
to the <order> MetaData element. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mappedBy</td>
<td>String</td>
<td>ORM : Field in the element class that represents the ordering of the collection/array</td>
</tr>
<tr>
<td>column</td>
<td>String</td>
<td>ORM : Name of the column for this order</td>
</tr>
<tr>
<td>columns</td>
<td>Column[]</td>
<td>ORM : Column definition for the column(s) of this order</td>
</tr>
<tr>
<td>extensions</td>
<td>Extension[]</td>
<td>Vendor extensions</td>
</tr>
</tbody>
</table>
71.1.34 @Key

Annotation used to define the key for any map to be persisted. Maps across to the `<map>` and `<key>` MetaData elements. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>types</td>
<td>Class[]</td>
<td>Type(s) of key. While the attribute allows multiple values DataNucleus currently only supports the first type value</td>
<td>When using generics will be taken from the Map definition, otherwise must be specified</td>
</tr>
<tr>
<td>embedded</td>
<td>String</td>
<td>Whether the key is embedded into a join table</td>
<td></td>
</tr>
<tr>
<td>serialized</td>
<td>String</td>
<td>Whether the key is serialised into the join table</td>
<td></td>
</tr>
<tr>
<td>dependent</td>
<td>String</td>
<td>Whether the key objects are dependent when deleting the owner map</td>
<td></td>
</tr>
<tr>
<td>mappedBy</td>
<td>String</td>
<td>Used to specify the field in the value class where the key is stored (optional).</td>
<td></td>
</tr>
<tr>
<td>embeddedMapping</td>
<td>Embedded[]</td>
<td>Definition of any embedding of the (persistable) key. Only 1 &quot;Embedded&quot; should be provided</td>
<td></td>
</tr>
<tr>
<td>table</td>
<td>String</td>
<td>ORM : Name of the table for this key</td>
<td></td>
</tr>
<tr>
<td>column</td>
<td>String</td>
<td>ORM : Name of the column for this key</td>
<td></td>
</tr>
<tr>
<td>foreignKey</td>
<td>String</td>
<td>ORM : Name of any foreign-key constraint to add</td>
<td></td>
</tr>
<tr>
<td>generateForeignKey</td>
<td>String</td>
<td>ORM : Whether to generate a FK constraint for the key (when not specifying the name)</td>
<td></td>
</tr>
</tbody>
</table>
deleteAction | ForeignKeyAction | ORM: Action to be applied to the foreign key for this key for action upon deletion
updateAction | ForeignKeyAction | ORM: Action to be applied to the foreign key for this key for action upon update
index | String | ORM: Name of any index constraint to add
indexed | String | ORM: Whether this key column is indexed
uniqueKey | String | ORM: Name of any unique key constraint to add
unique | String | ORM: Whether this key column is unique
columns | Column[] | ORM: Column definition for the column(s) of this key
extensions | Extension[] | Vendor extensions

```java
@PersistenceCapable
class MyClass {
    @Key(types=java.lang.String.class)
    Map myField;
    ...
}
```

### 71.1.35 @Value
Annotation used to define the value for any map to be persisted. Maps across to the `<map>` and `<value>` MetaData elements. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>types</td>
<td>Class[]</td>
<td>Type(s) of value. While the attribute allows multiple values DataNucleus currently only supports the first type value</td>
<td>When using generics will be taken from the Map definition, otherwise must be specified</td>
</tr>
<tr>
<td>embedded</td>
<td>String</td>
<td>Whether the value is embedded into a join table</td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td>Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------</td>
<td>---------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>serialized</td>
<td>String</td>
<td>Whether the value is serialised into the join table</td>
<td></td>
</tr>
<tr>
<td>dependent</td>
<td>String</td>
<td>Whether the value objects are dependent when deleting the owner map</td>
<td></td>
</tr>
<tr>
<td>mappedBy</td>
<td>String</td>
<td>Used to specify the field in the key class where the value is stored (optional).</td>
<td></td>
</tr>
<tr>
<td>embeddedMapping</td>
<td>Embedded[]</td>
<td>Definition of any embedding of the (persistable) value. Only 1 &quot;Embedded&quot; should be provided.</td>
<td></td>
</tr>
<tr>
<td>table</td>
<td>String</td>
<td>ORM : Name of the table for this value</td>
<td></td>
</tr>
<tr>
<td>column</td>
<td>String</td>
<td>ORM : Name of the column for this value</td>
<td></td>
</tr>
<tr>
<td>foreignKey</td>
<td>String</td>
<td>ORM : Name of any foreign-key constraint to add</td>
<td></td>
</tr>
<tr>
<td>deleteAction</td>
<td>ForeignKeyAction</td>
<td>ORM : Action to be applied to the foreign key for this value for action upon deletion</td>
<td></td>
</tr>
<tr>
<td>generateForeignKey</td>
<td>String</td>
<td>ORM : Whether to generate a FK constraint for the value (when not specifying the name)</td>
<td></td>
</tr>
<tr>
<td>updateAction</td>
<td>ForeignKeyAction</td>
<td>ORM : Action to be applied to the foreign key for this value for action upon update</td>
<td></td>
</tr>
<tr>
<td>index</td>
<td>String</td>
<td>ORM : Name of any index constraint to add</td>
<td></td>
</tr>
<tr>
<td>indexed</td>
<td>String</td>
<td>ORM : Whether this value column is indexed</td>
<td></td>
</tr>
<tr>
<td>uniqueKey</td>
<td>String</td>
<td>ORM : Name of any unique key constraint to add</td>
<td></td>
</tr>
<tr>
<td>unique</td>
<td>String</td>
<td>ORM : Whether this value column is unique</td>
<td></td>
</tr>
<tr>
<td>columns</td>
<td>Column[]</td>
<td>ORM : Column definition for the column(s) of this value</td>
<td></td>
</tr>
<tr>
<td>extensions</td>
<td>Extension[]</td>
<td>Vendor extensions</td>
<td></td>
</tr>
</tbody>
</table>
```java
@PersistenceCapable
public class MyClass {
    @Key(types=java.lang.String.class)
    @Value(types=org.datanucleus.samples.MyValueClass.class, dependent="true")
    Map myField;
    ...
}
```

### 71.1.36 @Join
Annotation used to specify a join to a join table for a collection/array/map. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>String</td>
<td>ORM : Name of the table</td>
<td></td>
</tr>
<tr>
<td>column</td>
<td>String</td>
<td>ORM : Name of the column to join our PK to in the join table (when only one column used)</td>
<td></td>
</tr>
<tr>
<td>primaryKey</td>
<td>String</td>
<td>ORM : Name of any primary key constraint to add for the join table</td>
<td></td>
</tr>
<tr>
<td>generatePrimaryKey</td>
<td>String</td>
<td>ORM : Whether to generate a PK constraint on the join table (when not specifying the name)</td>
<td></td>
</tr>
<tr>
<td>foreignKey</td>
<td>String</td>
<td>ORM : Name of any foreign-key constraint to add</td>
<td></td>
</tr>
<tr>
<td>generateForeignKey</td>
<td>String</td>
<td>ORM : Whether to generate a FK constraint on the join table (when not specifying the name)</td>
<td></td>
</tr>
<tr>
<td>index</td>
<td>String</td>
<td>ORM : Name of any index constraint to add</td>
<td></td>
</tr>
<tr>
<td>indexed</td>
<td>String</td>
<td>ORM : Whether the join column(s) is indexed</td>
<td></td>
</tr>
<tr>
<td>uniqueKey</td>
<td>String</td>
<td>ORM : Name of any unique constraint to add</td>
<td></td>
</tr>
<tr>
<td>unique</td>
<td>String</td>
<td>ORM : Whether the join column(s) has a unique constraint</td>
<td></td>
</tr>
<tr>
<td>columns</td>
<td>Column[]</td>
<td>ORM : Name of the columns to join our PK to in the join table (when multiple columns used)</td>
<td></td>
</tr>
</tbody>
</table>
@PersistenceCapable
public class MyClass
{
    @Persistent
    @Element(types=org.datanucleus.samples.MyElement.class)
    @Join(table="MYCLASS_ELEMENTS", column="MYCLASS_ELEMENTS_PK")
    Collection myField;
    ...
}

71.1.37 @Embedded

Annotation used to define that the field contents is embedded into the same table as this field Maps across to the <embedded> MetaData element. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>ownerMember</td>
<td>String</td>
<td>ORM : The field/property in the embedded object that links back to the owning object (where it has a bidirectional relation)</td>
<td></td>
</tr>
<tr>
<td>nullIndicatorColumn</td>
<td>String</td>
<td>ORM : The column in the embedded object used to judge if the embedded object is null.</td>
<td></td>
</tr>
<tr>
<td>nullIndicatorValue</td>
<td>String</td>
<td>ORM : The value in the null column to interpret the object as being null.</td>
<td></td>
</tr>
<tr>
<td>members</td>
<td>Persistent[]</td>
<td>ORM : Field/property definitions for this embedding.</td>
<td></td>
</tr>
</tbody>
</table>
@PersistenceCapable
public class MyClass
{
    @Embedded(members={
        @Persistent(name="field1", columns=@Column(name="OTHER_FLD_1")),
        @Persistent(name="field2", columns=@Column(name="OTHER_FLD_2"))
    })
    MyOtherClass myField;
    ...
}

@PersistenceCapable
@EmbeddedOnly
public class MyOtherClass
{
    @Persistent
    String field1;
    ...
    @Persistent
    String field2;
}

### 71.1.38 @Columns
Annotation used to define the columns into which a field is persisted. If the field is persisted into a single column then @Column should be used. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Column[]</td>
<td>Array of columns - see @Columns annotation</td>
<td></td>
</tr>
</tbody>
</table>

@PersistenceCapable
public class MyClass
{
    @Persistent
    @Columns({@Column(name="RED"), @Column(name="GREEN"), @Column(name="BLUE"), @Column(name="ALPHA"))})
    Color myField;
    ...
}

### 71.1.39 @Column
Annotation used to define that the column where a field is persisted. Is equivalent to the <column> metadata element when specified under field. Specified on the field/method (and within other annotations).
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>ORM : Name of the column</td>
<td></td>
</tr>
<tr>
<td>target</td>
<td>String</td>
<td>ORM : Column in the other class that this maps to</td>
<td></td>
</tr>
<tr>
<td>targetMember</td>
<td>String</td>
<td>ORM : Field/Property in the other class that this maps to</td>
<td></td>
</tr>
<tr>
<td>jdbcType</td>
<td>String</td>
<td>ORM : JDBC Type to use for persisting into this column</td>
<td></td>
</tr>
<tr>
<td>sqlType</td>
<td>String</td>
<td>ORM : SQL Type to use for persisting into this column</td>
<td></td>
</tr>
<tr>
<td>length</td>
<td>int</td>
<td>ORM : Max length of data to store in this column</td>
<td></td>
</tr>
<tr>
<td>scale</td>
<td>int</td>
<td>ORM : Max number of floating points of data to store in this column</td>
<td></td>
</tr>
<tr>
<td>allowsNull</td>
<td>String</td>
<td>ORM : Whether null is allowed to be persisted into this column</td>
<td></td>
</tr>
<tr>
<td>defaultValue</td>
<td>String</td>
<td>ORM : Default value to persist into this column. If you want the default to</td>
<td>&quot;#NULL&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>be NULL, then put this as &quot;#NULL&quot;</td>
<td></td>
</tr>
<tr>
<td>insertValue</td>
<td>String</td>
<td>ORM : Value to insert into this column when it is an &quot;unmapped&quot; column. If</td>
<td>&quot;#NULL&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>you want the inserted value to be NULL, then put this as &quot;#NULL&quot;</td>
<td></td>
</tr>
<tr>
<td>position</td>
<td>int</td>
<td>Position of this column in the owning table (0 = first)</td>
<td></td>
</tr>
<tr>
<td>extensions</td>
<td>Extension[]</td>
<td>Vendor extensions</td>
<td></td>
</tr>
</tbody>
</table>

```java
@PersistenceCapable
public class MyClass
{
    @Persistent
    @Column(name="MYCOL", jdbcType="VARCHAR", length=40)
    String field1;

    ...
}
```
71.1.40 @Index
Annotation used to define that this field is indexed. Is equivalent to the <index> metadata element when specified under field. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>ORM: Name of the index</td>
<td></td>
</tr>
<tr>
<td>unique</td>
<td>String</td>
<td>ORM: Whether the index is unique</td>
<td></td>
</tr>
</tbody>
</table>

@PersistenceCapable
public class MyClass
{
    @Persistent
    @Index(name="MYFIELD1_IDX")
    String field1;

    @Persistent
    @Index(name="MYFIELD2_IDX", unique="true")
    String field2;

    ...
}

See the documentation for Schema Constraints

71.1.41 @Unique
Annotation used to define that this field has a unique constraint. Is equivalent to the <unique> metadata element when specified under field. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>ORM: Name of the constraint</td>
<td></td>
</tr>
<tr>
<td>deferred</td>
<td>String</td>
<td>ORM: Whether the constraint is deferred</td>
<td></td>
</tr>
</tbody>
</table>

@PersistenceCapable
public class MyClass
{
    @Persistent
    @Unique(name="MYFIELD1_IDX")
    String field1;

    ...
}
See the documentation for Schema Constraints

### 71.1.42 @ForeignKey
Annotation used to define the foreign key for a relationship field. Is equivalent to the `<foreign-key>` metadata element when specified under field. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>ORM: Name of the constraint</td>
<td></td>
</tr>
<tr>
<td>deferred</td>
<td>String</td>
<td>ORM: Whether the constraint is deferred</td>
<td></td>
</tr>
<tr>
<td>unique</td>
<td>String</td>
<td>ORM: Whether the constraint is unique</td>
<td></td>
</tr>
<tr>
<td>deleteAction</td>
<td>ForeignKeyAction</td>
<td>ORM: Action to apply to the FK to be used on deleting</td>
<td>ForeignKeyAction.RESTRICT</td>
</tr>
<tr>
<td>updateAction</td>
<td>ForeignKeyAction</td>
<td>ORM: Action to apply to the FK to be used on updating</td>
<td>ForeignKeyAction.RESTRICT</td>
</tr>
</tbody>
</table>

```java
@PersistenceCapable
public class MyClass
{
    @Persistent
    @ForeignKey(name="MYFIELD1_FK", deleteAction=ForeignKeyAction.RESTRICT)
    String field1;
    ...
}
```

See the documentation for Schema Constraints

### 71.1.43 @Extensions
Annotation used to define a set of extensions specific to the JDO2 implementation being used. Specified on the class or field.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Extension[]</td>
<td>Array of extensions - see @Extension annotation</td>
<td></td>
</tr>
</tbody>
</table>
@PersistenceCapable
@Extensions({
    @Extension(vendorName="datanucleus", key="firstExtension", value="myValue"),
    @Extension(vendorName="datanucleus", key="secondExtension", value="myValue")
})
public class Person
{
    ...
}

71.1.44 @Extension
Annotation used to define an extension specific to a particular JDO implementation. Is equivalent to the <extension> metadata element. Specified on the class or field.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>vendorName</td>
<td>String</td>
<td>Name of the JDO vendor</td>
<td></td>
</tr>
<tr>
<td>key</td>
<td>String</td>
<td>Key for the extension</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>String</td>
<td>Value of the extension</td>
<td></td>
</tr>
</tbody>
</table>

@PersistenceCapable
@Extension(vendorName="DataNucleus", key="RunFast", value="true")
public class Person
{
    ...
}
72 MetaData API

72.1 JDO : Metadata API
When using JDO you need to define which classes are persistent, and also how they are persisted. JDO has allowed XML metadata since its first revision, and introduced support for annotations in JDO 2.1. JDO 3.0 introduces a programmatic API to do the same task.

72.1.1 Defining Metadata for classes
The basic idea behind the Metadata API is that the developer obtains a metadata object from the PersistenceManagerFactory, and adds the definition to that as required, before registering it for use in the persistence process.

```java
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(propsFile);
...
JDOMetadata md = pmf.newMetadata();
```

So we have a `JDOMetadata` object and want to define the persistence for our class `mydomain.MyClass`, so we do as follows

```java
PackageMetadata pmd = md.newPackageMetadata("mydomain");
ClassMetadata cmd = pmd.newClassMetadata("MyClass");
```

So we follow the same structure of the JDO XML Metadata file adding packages to the top level, and classes to the respective package. Note that we could have achieved this by a simple typesafe invocation

```java
ClassMetadata cmd = md.newClassMetadata(MyClass.class);
```

So now we have the class defined, we need to set its key information

```java
cmd.setTable("CLIENT").setDetachable(true).setIdentityType(IdentityType.DATASTORE);
cmd.setPersistenceModifier(ClassPersistenceModifier.PERSISTENCE_CAPABLE);
InheritanceMetadata inhmd = cmd.newInheritanceMetadata();
inhmd.setStrategy(InheritanceStrategy.NEW_TABLE);
DiscriminatorMetadata dmd = inhmd.newDiscriminatorMetadata();
dmd.setColumn("disc").setValue("Client");
dmd.setStrategy(DiscriminatorStrategy.VALUE_MAP).setIndexed(Indexed.TRUE);
VersionMetadata vermd = cmd.newVersionMetadata();
vermd.setStrategy(VersionStrategy.VERSION_NUMBER);
vermd.setColumn("version").setIndexed(Indexed.TRUE);
```

And we define also define fields/properties via the API in a similar way
FieldMetadata fmd = cmd.newFieldMetadata("name");
fmd.setNullValue(NullValue.DEFAULT).setColumn("client_name");
fmd.setIndexed(true).setUnique(true);

Note that, just like with XML metadata, we don’t need to add information for all fields since they have their own default persistence settings based on the type of the field.

All that remains is to register the metadata with the persistence process

pmf.registerMetadata(md);

72.1.2 Accessing Metadata for classes

Maybe you have a class with its persistence defined in XML or annotations and you want to check its persistence information at runtime. With the JDO Metadata API you can do that

TypeMetadata compmd = pmf.getMetadata("mydomain.MyOtherClass");

and we can now inspect the information, casting the compmd to either javax.jdo.metadata.ClassMetadata or javax.jdo.metadata.InterfaceMetadata.

Please note that you cannot currently change metadata retrieved in this way, only view it...
JDO defines that MetaData (defined in the MetaData guide) can be found in particular locations in the CLASSPATH, and has a particular format. It also defines that you can split your MetaData for Object Relational Mapping (ORM) into separate files if you so wish. So you would define your basic persistence in a file "package.jdo" and then define the MetaData files "package-mysql.orm" (for MySQL), and "package-oracle.orm" (for Oracle). To make use of this JDO 2 Object-Relational Mapping file separation, you must specify the PersistenceManagerFactory property datanucleus.Mapping. If you set this to, for example, mysql DataNucleus would look for files such as package.jdo and package-mysql.orm in the same locations as specified above.

73.1.1 Simple Example

Let us take a sample class and generate MetaData for it. Suppose I have a class as follows:

```java
package mydomain;

public class Person {
    /** Title of the Person. */
    String title=null;

    /** Forename of the Person. */
    String forename=null;

    /** Surname of the Person. */
    String surname=null;

    ...
}
```

and I want to use an existing schema. With this case I need to define the table and column names that it maps to. To do this I need to use JDO 2 ORM tags. So I come up with MetaData as follows in package.jdo:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo PUBLIC
"-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
"http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
    <package name="mydomain">
        <class name="Person" identity-type="datastore">
            <field name="title"/>
            <field name="forename"/>
            <field name="surname"/>
        </class>
    </package>
</jdo>
```
and then I add the ORM information in `package-mysql.orm` as

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE orm PUBLIC
    "-//Sun Microsystems, Inc.//DTD Java Data Objects Mapping Metadata 2.0//EN"
    "http://java.sun.com/dtd/jdo.orm_2_0.dtd">
<orm>
    <package name="mydomain">
        <class name="Person" table="PERSON">
            <field name="title">
                <column name="TITLE"/>
            </field>
            <field name="forename">
                <column name="FORENAME" length="100" jdbc-type="VARCHAR"/>
            </field>
            <field name="surname">
                <column name="SURNAME" length="100" jdbc-type="VARCHAR"/>
            </field>
        </class>
    </package>
</orm>
```

So you see that our class is being mapped across to a table "PERSON" in the datastore, with columns "TITLE", "FORENAME", "SURNAME". We have also specified that the upper size limit on the forename and surname fields is 100.

### 73.1.2 Memory utilisation

The XML files are parsed and populated to memory the first time a persistent operation is executed over a persistent class (e.g. `pm.makePersistent(object)`). If the persistent class has relationships to other persistent classes, the metadata for the classes in the relationships are loaded. In addition to the persistent class and classes in the relationships, all other classes / files that were encountered while searching for the persistent classes are loaded, plus their relationships.

In average, for each persistent class a 3kb of memory is used to hold metadata information. This value will vary according the amount of metadata declared. Although this value can be used as reference in earlier stages of development, you should verify if it corresponds to your persistent classes.

A general formula can be used (with caution) to estimate the amount of memory required:

\[
\text{Amount Required} = (\# \text{ of persistent classes}) \times 3\text{KB}
\]
74 Schema Mapping

74.1 JDO : Schema Mapping
You saw in our basic class mapping guide how you define MetaData for a classes basic persistence, notating which fields are persisted. The next step is to define how it maps to the schema of the datastore (in this case RDBMS). The simplest way of mapping is to map each class to its own table. This is the default model in JDO persistence (with the exception of inheritance). If you don't specify the table and column names, then DataNucleus will generate table and column names for you. **You should specify your table and column names if you have an existing schema.** Failure to do so will mean that DataNucleus uses its own names and these will almost certainly not match what you have in the datastore. There are several aspects to cover here

- Table and column names
- Column for datastore identity
- Column(s) for application identity
- Column nullability and default value
- Column Types
- Columns with no field in the class

74.1.1 Tables and Column names
The main thing that developers want to do when they set up the persistence of their data is to control the names of the tables and columns used for storing the classes and fields. This is an essential step when mapping to an existing schema, because it is necessary to map the classes onto the existing database entities. Let's take an example

```java
public class Hotel {
    private String name;
    private String address;
    private String telephoneNumber;
    private int numberOfRooms;
    ...
}
```

In our case we want to map this class to a table called **ESTABLISHMENT**, and has columns **NAME, DIRECTION, PHONE** and **NUMBER_OF_ROOMS** (amongst other things). So we define our Meta-Data like this
So we have defined the table and the column names. It should be mentioned that if you don’t specify the table and column names then DataNucleus will generate names for the datastore identifiers. The table name will be based on the class name, and the column names will be based on the field names and the role of the field (if part of a relationship).

See also :-

- Identifier Guide - defining the identifiers to use for table/column names
- MetaData reference for <column> element
- MetaData reference for <primary-key> element
- Annotations reference for @Column
- Annotations reference for @PrimaryKey

### 74.1.2 Column names for datastore-identity

When you select `datastore-identity` a surrogate column will be added in the datastore. You need to be able to define the column name if mapping to an existing schema (or wanting to control the schema). So lets say we have the following:

```java
public class MyClass // persisted to table "MYCLASS"
{
    ...
}

public class MySubClass extends MyClass // persisted to table "MYSUBCLASS"
{
    ...
}
```

We want to define the names of the identity column in "MYCLASS" and "MYSUBCLASS". Here’s how we do it:
So we will have a PK column "MY_PK_COLUMN" in the table "MYCLASS", and a PK column "MYSUB_PK_COLUMN" in the table "MYSUBCLASS" (and that corresponds to the "MY_PK_COLUMN" value in "MYCLASS"). We could also do

See also:-

- Inheritance Guide - defining how to use inheritance between classes
- MetaData reference for <column> element
- MetaData reference for <primary-key> element
- Annotations reference for @Column
- Annotations reference for @PrimaryKey

74.1.3 Column names for application-identity

When you select application-identity you have some field(s) that form the "primary-key" of the class. A common situation is that you have inherited classes and each class has its own table, and so the primary-key column names can need defining for each class in the inheritance tree. So lets show an example how to do it
public class MyClass // persisted to table "MYCLASS"
{
    long id; // PK field
    ...
}
public class MySubClass extends MyClass // persisted to table "MYSUBCLASS"
{
    ...
}

Defining the column name for "MyClass.id" is easy since we use the same as shown previously "column" for the field. Obviously the table "MYSUBCLASS" will also need a PK column. Here’s how we define the column mapping

```xml
<class name="MyClass" identity-type="application" table="MYCLASS">
    <field name="myPrimaryKeyField" primary-key="true">
        <column name="MY_PK_COLUMN"/>
    </field>
    ...
</class>
<class name="MySubClass" identity-type="application" table="MYSUBCLASS">
    <inheritance strategy="new-table"/>
    <primary-key>
        <column name="MYSUB_PK_COLUMN" target="MY_PK_COLUMN"/>
    </primary-key>
    ...
</class>
```

So we will have a PK column "MY_PK_COLUMN" in the table "MYCLASS", and a PK column "MYSUB_PK_COLUMN" in the table "MYSUBCLASS" (and that corresponds to the "MY_PK_COLUMN" value in "MYCLASS"). You can also use

```xml
<class name="MyClass" identity-type="application" table="MYCLASS">
    <field name="myPrimaryKeyField" primary-key="true">
        <column name="MY_PK_COLUMN"/>
    </field>
    ...
</class>
<class name="MySubClass" identity-type="application" table="MYSUBCLASS">
    <inheritance strategy="new-table">
        <join>
            <column name="MYSUB_PK_COLUMN" target="MY_PK_COLUMN"/>
        </join>
    </inheritance>
    ...
</class>
```

See also :-
- Inheritance Guide - defining how to use inheritance between classes
- MetaData reference for <inheritance> element
74.1.4 Column nullability and default values

So we've seen how to specify the basic structure of a table, naming the table and its columns, and how to control the types of the columns. We can extend this further to control whether the columns are allowed to contain nulls and to set a default value for a column if we ever have need to insert into it and not specify a particular column. Let's take a related class for our hotel. Here we have a class to model the payments made to the hotel.

```java
public class Payment {
    Customer customer;
    String bankTransferReference;
    String currency;
    double amount;
}
```

In this class we can model payments from a customer of an amount. Where the customer pays by bank transfer we can save the reference number. Since our hotel is in the United Kingdom we want the default currency to be pounds, or to use its ISO4217 currency code "GBP". In addition, since the bank transfer reference is optional we want that column to be nullable. So let's specify the MetaData for the class.

```xml
<class name="Payment">
    <field name="customer" persistence-capable="persistent" column="CUSTOMER_ID"/>
    <field name="bankTransferReference">
        <column name="TRANSFER_REF" allows-null="true"/>
    </field>
    <field name="currency">
        <column name="CURRENCY" default-value="GBP"/>
    </field>
    <field name="amount" column="AMOUNT"/>
</class>
```

So we make use of the `allows-null` and `default-value` attributes. The table, when created by DataNucleus, will then provide the default and nullability that we require.

See also:

- MetaData reference for `<column>` element
- Annotations reference for `@Column`
74.1.5 Column types

DataNucleus will provide a default type for any columns that it creates, but it will allow users to override this default. The default that DataNucleus chooses is always based on the Java type for the field being mapped. For example a Java field of type "int" will be mapped to a column type of INTEGER in RDBMS datastores. Similarly String will be mapped to VARCHAR. To override the default setting (and always the best policy if you are wanting your MetaData to give the same datastore definition with all JDO implementations) you do as follows

```xml
<class name="Payment">
  <field name="customer" persistence-capable="persistent" column="CUSTOMER_ID">
    <column name="TRANSFER_REF" jdbc-type="VARCHAR" length="255" allows-null="true"/>
  </field>
  <field name="bankTransferReference">
    <column name="TRANSFER_REF" jdbc-type="VARCHAR" length="255" allows-null="true"/>
  </field>
  <field name="currency">
    <column name="CURRENCY" jdbc-type="CHAR" length="3" default-value="GBP"/>
  </field>
  <field name="amount">
    <column name="AMOUNT" jdbc-type="DECIMAL" length="10" scale="2"/>
  </field>
</class>
```

So we have defined TRANSFER_REF to use VARCHAR(255) column type, CURRENCY to use CHAR(3) column type, and AMOUNT to use DECIMAL(10,2) column type. Please be aware that DataNucleus only supports persisting particular Java types to particular JDBC/SQL types. We have demonstrated above the jdbc-type attribute, but there is also an sql-type attribute. This is to be used where you want to map to some specific SQL type (and will not be needed in the vast majority of cases - the jdbc-type should generally be used).

See also :

- Types Guide - defining persistence of Java types
- RDBMS Types Guide - defining mapping of Java types to available JDBC/SQL types
- MetaData reference for <column> element
- Annotations reference for @Column

74.1.6 Columns with no field in the class

DataNucleus supports mapping of columns in the datastore that have no associated field in the java class. These are useful where you maybe have a table used by other applications and dont use some of the information in your Java model. DataNucleus needs to know about these columns so that it can validate the schema correctly, and also insert particular values when inserting objects into the table. You could handle this by defining your schema yourself so that the particular columns have "DEFAULT" settings, but this way you allow DataNucleus to know about all information. So to give an example
So in this example our table "ESTABLISHMENT" has the columns associated with the specified fields and also has columns "YEAR_ESTABLISHED" (that is INTEGER-based and will be given a value of "1980" on any inserts) and "MANAGER_NAME" (VARCHAR-based and will be given a value of "N/A" on any inserts).
75 Multitenancy

75.1 JDO : Multitenancy

On occasion you need to share a data model with other user-groups or other applications and where the model is persisted to the same structure of datastore. There are three ways of handling this with DataNucleus.

- **Separate Database per Tenant** - have a different database per user-group/application.
- **Separate Schema per Tenant** - as the first option, except use different schemas.
- **Same Database/Schema but with a Discriminator** - this is described below.

75.1.1 Multitenancy via Discriminator

If you specify the persistence property `datanucleus.tenantId` as an identifier for your user-group/application then DataNucleus will know that it needs to provide a tenancy discriminator to all primary tables of persisted classes. This discriminator is then used to separate the data of the different user-groups.

By default this will add a column **TENANT_ID** to each primary table, of String-based type. You can control this by specifying extension metadata for each persistable class.

```
<class name="MyClass">
  <extension vendor-name="datanucleus" key="multitenancy-column-name" value="TENANT"/>
  <extension vendor-name="datanucleus" key="multitenancy-column-length" value="24"/>
  ...
</class>
```

In all subsequent use of DataNucleus, any "insert" to the primary "table"(s) will also include the TENANT column value. Additionally any query will apply a WHERE clause restricting to a particular value of TENANT column.

If you want to disable multitenancy on a class, just specify the following metadata.

```
<class name="MyClass">
  <extension vendor-name="datanucleus" key="multitenancy-disable" value="true"/>
  ...
</class>
```
76 Datastore Identifiers

76.1 JDO : Datastore Identifiers
A datastore identifier is a simple name of a database object, such as a column, table, index, or view, and is composed of a sequence of letters, digits, and underscores (_ ) that represents it’s name. DataNucleus allows users to specify the names of tables, columns, indexes etc but if the user doesn’t specify these DataNucleus will generate names. Generation of identifier names is controlled by an IdentifierFactory, and DataNucleus provides a default implementation. You can provide your own IdentifierFactory plugin to give your own preferred naming if so desired. You set the IdentifierFactory by setting the PMF property datanucleus.identifierFactory. Set it to the symbolic name of the factory you want to use. JDO doesn’t define what the names of datastore identifiers should be but DataNucleus provides the following factories for your use.

- datanucleus2 IdentifierFactory (default for JDO persistence)
- jpa IdentifierFactory (default for JPA persistence)
- datanucleus1 IdentifierFactory (used in DataNucleus v1)
- jpox IdentifierFactory (compatible with JPOX)

In describing the different possible naming conventions available out of the box with DataNucleus we’ll use the following example

```java
class MyClass
{
    String myField1;
    Collection<MyElement> elements1; // Using join table
    Collection<MyElement> elements2; // Using foreign-key
}
class MyElement
{
    String myElementField;
    MyClass myClass2;
}
```

76.1.1 IdentifierFactory ‘datanucleus2’

This became the default for JDO persistence from DataNucleus v2.x onwards and changes a few things over the previous “datanucleus1” factory, attempting to make the naming more concise and consistent (we retain “datanucleus1” for backwards compatibility).

Using the same example above, the rules in this IdentifierFactory mean that, assuming that the user doesn’t specify any <column> elements :-

- MyClass will be persisted into a table named MYCLASS
- When using datastore identity MYCLASS will have a column called MYCLASS_ID
- MyClass.myField1 will be persisted into a column called MYFIELD1
- MyClass.myField1 will be persisted into a column called MYFIELD1
- MyClass.elements1 will be persisted into a join table called MYCLASS_ELEMENTS1
- **MYCLASS_ELEMENTS1** will have columns called **MYCLASS_ID_OID** (FK to owner table) and **MYELEMENT_ID_EID** (FK to element table)
- **MYCLASS_ELEMENTS1** will have column names like **STRING_ELE**, **STRING_KEY**, **STRING_VAL** for non-PC elements/keys/values of collections/maps
- **MyClass.elements2** will be persisted into a column **ELEMENTS2_MYCLASS_ID_OWN** or **ELEMENTS2_MYCLASS_ID_OID** (FK to owner) table
- Any discriminator column will be called **DISCRIMINATOR**
- Any index column in a List will be called **IDX**
- Any adapter column added to a join table to form part of the primary key will be called **IDX**
- Any version column for a table will be called **VERSION**

76.1.2 IdentifierFactory 'datanucleus1'

This was the default in DataNucleus v1.x for JDO persistence and provided a reasonable default naming of datastore identifiers using the class and field names as its basis.

Using the example above, the rules in this IdentifierFactory mean that, assuming that the user doesn't specify any `<column>` elements :-

- **MyClass** will be persisted into a table named **MYCLASS**
- When using datastore identity **MYCLASS** will have a column called **MYCLASS_ID**
- **MyClass.myField1** will be persisted into a column called **MY_FIELD1**
- **MyElement** will be persisted into a table named **MYELEMENT**
- **MyClass.elements1** will be persisted into a join table called **MYCLASS_ELEMENTS1**
- **MYCLASS_ELEMENTS1** will have columns called **MYCLASS_ID_OID** (FK to owner table) and **MYELEMENT_ID_EID** (FK to element table)
- **MYCLASS_ELEMENTS1** will have column names like **STRING_ELE**, **STRING_KEY**, **STRING_VAL** for non-PC elements/keys/values of collections/maps
- **MyClass.elements2** will be persisted into a column **ELEMENTS2_MYCLASS_ID_OID** or **ELEMENTS2_MYCLASS_ID_OID** (FK to owner) table
- Any discriminator column will be called **DISCRIMINATOR**
- Any index column in a List will be called **INTEGER_IDX**
- Any adapter column added to a join table to form part of the primary key will be called **ADPT_PK_IDX**
- Any version column for a table will be called **OPT_VERSION**

76.1.3 IdentifierFactory 'jpa'

The IdentifierFactory "jpa" aims at providing a naming policy consistent with the "JPA" specification.

Using the same example above, the rules in this IdentifierFactory mean that, assuming that the user doesn't specify any `<column>` elements :-

- **MyClass** will be persisted into a table named **MYCLASS**
- When using datastore identity **MYCLASS** will have a column called **MYCLASS_ID**
- MyClass.myField1 will be persisted into a column called MYFIELD1
- MyElement will be persisted into a table named MYELEMENT
- MyClass.elements1 will be persisted into a join table called MYCLASS_MYELEMENT
- MYCLASS_ELEMENTS1 will have columns called MYCLASS_MYCLASS_ID (FK to owner table) and ELEMENTS1_ELEMENT_ID (FK to element table)
- MyClass.elements2 will be persisted into a column ELEMENTS2_MYCLASS_ID (FK to owner) table
- Any discriminator column will be called DTYPE
- Any index column in a List for field MyClass.myField1 will be called MYFIELD1_ORDER
- Any adapter column added to a join table to form part of the primary key will be called IDX
- Any version column for a table will be called VERSION

76.1.4 IdentifierFactory 'jpox'

This IdentifierFactory exists for backward compatibility with JPOX 1.2.0. If you experience changes of schema identifiers when migrating from JPOX 1.2.0 to datanucleus, you should give this one a try.

Schema compatibility between JPOX 1.2.0 and datanucleus had been broken e.g. by the number of characters used in hash codes when truncating identifiers: this has changed from 2 to 4.

76.1.5 IdentifierFactory - Controlling the Case

The underlying datastore will define what case of identifiers are accepted. By default, DataNucleus will capitalise names (assuming that the datastore supports it). You can however influence the case used for identifiers. This is specifiable with the PMF property datanucleus.identifier.case, having the following values

- UpperCase: identifiers are in upper case
- LowerCase: identifiers are in lower case
- PreserveCase: No case changes are made to the name of the identifier provided by the user (class name or jdo metadata).

Please be aware that some datastores only support UPPERCASE or lowercase identifiers and so setting this parameter may have no effect if your database doesn’t support that option. Please note also that this case control only applies to DataNucleus-generated identifiers. If you provide your own identifiers for things like schema/catalog etc then you need to specify those using the case you wish to use in the datastore (including quoting as necessary)
77 Secondary Tables

77.1 JDO : Secondary Tables

Applicable to RDBMS

The standard JDO persistence strategy is to persist an object of a class into its own table. In some situations you may wish to map the class to a primary table as well as one or more secondary tables. For example when you have a Java class that could have been split up into 2 separate classes yet, for whatever reason, has been written as a single class, however you have a legacy datastore and you need to map objects of this class into 2 tables. JDO allows persistence of fields of a class into secondary tables.

The process for managing this situation is best demonstrated with an example. Let's suppose we have a class that represents a Printer. The Printer class contains within it various attributes of the toner cartridge. So we have:

```java
package com.mydomain.samples.secondarytable;

public class Printer
{
    long id;
    String make;
    String model;

    String tonerModel;
    int tonerLifetime;

    /*
     * Constructor.
     * @param make Make of printer (e.g Hewlett-Packard)
     * @param model Model of Printer (e.g LaserJet 1200L)
     * @param tonerModel Model of toner cartridge
     * @param tonerLifetime lifetime of toner (number of prints)
     */
    public Printer(String make, String model, String tonerModel, int tonerLifetime)
    {
        this.make = make;
        this.model = model;
        this.tonerModel = tonerModel;
        this.tonerLifetime = tonerLifetime;
    }
}
```

Now we have a database schema that has 2 tables (PRINTER and PRINTER_TONER) in which to store objects of this class. So we need to tell DataNucleus to perform this mapping. So we define the MetaData for the Printer class like this:
So here we have defined that objects of the **Printer** class will be stored in the primary table **PRINTER**. In addition we have defined that some fields are stored in the table **PRINTER_TONER**. This is achieved by way of

- We will store *tonerModel* and *tonerLifetime* in the table **PRINTER_TONER**. This is achieved by using `<field table="PRINTER_TONER">`
- The table **PRINTER_TONER** will use a primary key column called **PRINTER_REFID**. This is achieved by using `<join table="PRINTER_TONER" column="PRINTER_REFID"/>

You could equally specify this using annotations

```
@PersistenceCapable
@Join(name="PRINTER_TONER", column="PRINTER_REFID")
public class Printer
{
    @Persistent(primaryKey="true", column="PRINTER_ID")
    long id;
    @Column(name="MAKE")
    String make;
    @Column(name="MODEL")
    String model;

    @Persistent(table="PRINTER_TONER", column="MODEL")
    String tonerModel;
    @Persistent(table="PRINTER_TONER", column="LIFETIME")
    int tonerLifetime;
    ...
}
```

This results in the following database tables:-

```
So we now have our primary and secondary database tables. The primary key of the 
PRINTER_TONER table serves as a foreign key to the primary class. Whenever we persist a **Printer** 
object a row will be inserted into both of these tables.

### 77.1.1 Specifying the primary key

You saw above how we defined the column name that will be the primary key of the secondary table 
(the PRINTER_REFID column). What we didn't show is how to specify the name of the primary key 
constraint to be generated. To do this you change the MetaData to

```xml
<class name="Printer" identity-type="datastore" table="PRINTER">
  <join table="PRINTER_TONER" column="PRINTER_REFID">
    <primary-key name="TONER_PK"/>
  </join>
  <field name="id" primary-key="true">
    <column name="PRINTER_ID"/>
  </field>
  <field name="make">
    <column name="MAKE"/>
  </field>
  <field name="model">
    <column name="MODEL"/>
  </field>
  <field name="tonerModel" table="PRINTER_TONER">
    <column name="MODEL"/>
  </field>
  <field name="tonerLifetime" table="PRINTER_TONER">
    <column name="LIFETIME"/>
  </field>
</class>
```

So this will create the primary key constraint with the name "TONER_PK".

See also :-

- MetaData reference for `<primary-key>` element
- MetaData reference for `<join>` element
- Annotations reference for `@PrimaryKey`
- Annotations reference for `@Join`
- Secondary Table tutorial

### 77.2 Worked Example

The above process can be seen with an example
public class Printer {
    String make;
    String model;
    String tonerModel;
    int tonerLifetime;

    public Printer(String make, String model, String tonerModel, int tonerLife) {
        this.make = make;
        this.model = model;
        this.tonerModel = tonerModel;
        this.tonerLifetime = tonerLife;
    }

    ...
}

We now need to specify which fields we want to store in any secondary tables. To do this we can define the metadata like this

```xml
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC
-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN
 "http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
    <package name="mydomain">
        <class name="Printer" table="PRINTER">
            <datastore-identity>
                <column name="ID"/>
            </datastore-identity>
            <join table="TONER">
                <column name="ID"/>
            </join>
            <field name="make"/>
            <field name="model"/>
            <field name="tonerModel" table="TONER"/>
            <field name="tonerLifetime" table="TONER"/>
        </class>
    </package>
</jdo>
```

With this we have stated that the fields make and model will be stored in the default table, that we named PRINTER, and that tonerModel and tonerLifetime we be stored in the table TONER. The tables will both store the unique identity assigned to the objects we persist, in this case we have specified the column name ID for both tables, though we would usually only do this when working to an existing schema. When we retrieve any of our stored objects the tables will be joined automatically by matching the identities.

We can see how this works in more detail by setting the query logging to DEBUG (set log4j.category.DataNucleus.Query=DEBUG, in your log4j.properties file). We can retrieve all of our stored Printer objects by performing the following query
Query q = pm.newQuery(Printer.class);
List<Printer> list = (List<Printer>)q.execute();

Now if we look in our log file we can see how this has been converted into the appropriate query language for our datastore. With an RDBMS datastore using SQL, for example, we get

```
SELECT FROM mydomain.Printer Query compiled to datastore query
"SELECT 'mydomain.Printer' AS NUCLEUS_TYPE, 'A0'.MAKE, 'A0'.MODEL, 'A1'.TONER_MODEL,
'A1'.TONER_LIFETIME, 'A0'.ID
FROM 'PRINTER' 'A0' INNER JOIN 'TONER' 'A1' ON 'A0'.ID = 'A1'.ID"
```

So we can see that in this case an INNER JOIN was performed using the ID columns as expected.

*This worked example was provided by a DataNucleus user Tom Robson*
78 Constraints

78.1 JDO : Constraints

A datastore often provides ways of constraining the storage of data to maintain relationships and improve performance. These are known as *constraints* and they come in various forms. These are:

- **Indexes** - these are used to mark fields that are referenced often as indexes so that when they are used the performance is optimised.
- **Unique constraints** - these are placed on fields that should have a unique value. That is only one object will have a particular value.
- **Foreign-Keys** - these are used to interrelate objects, and allow the datastore to keep the integrity of the data in the datastore.
- **Primary-Keys** - allow the PK to be set, and also to have a name.

### 78.1.1 Indexes

Applicable to RDBMS, NeoDatis, MongoDB.

Many datastores provide the ability to have indexes defined to give performance benefits. With RDBMS the indexes are specified on the table and the indexes to the rows are stored separately. In the same way an ODBMS typically allows indexes to be specified on the fields of the class, and these are managed by the datastore. JDO provides a mechanism for defining indexes, and hence if a developer knows that a particular field is going to be highly used for querying, they can select that field to be indexed in their (JDO) persistence solution. Let’s take an example class, and show how to specify this

```java
public class Booking
{
    private int bookingType;
    ...
}
```

We decide that our *bookingType* is going to be highly used and we want to index this in the persistence tool. To do this we define the Meta-Data for our class as

```xml
<class name="Booking">
    <field name="bookingType">
        <index name="BOOKING_TYPE_INDEX"/>
    </field>
</class>
```

This will mean that DataNucleus will create an index in the datastore for the field and the index will have the name *BOOKING_TYPE_INDEX* (for datastores that support using named indexes). If we had wanted the index to provide uniqueness, we could have made this

```xml
    <index name="BOOKING_TYPE_INDEX" unique="true"/>
```
This has demonstrated indexing the fields of a class. The above example will index together all columns for that field. In certain circumstances you want to be able to index from the column point of view. So we are thinking more from a database perspective. Here we define our indexes at the <class> level, like this:

```xml
<class name="Booking">
  <index name="MY_BOOKING_INDEX">
    <column name="BOOKING"/>
  </index>

  <extension vendor-name="datanucleus" key="extended-setting" value="USING HASH"/>

</class>
```

This creates an index for the specified column (where the datastore supports columns i.e RDBMS).

Should you have need to tailor the index creation, for example to generate a particular type of index (where the datastore supports it), you can specify extended settings that is appended to the end of any CREATE INDEX statement:

```xml
<class name="Booking">
  <index name="MY_BOOKING_INDEX">
    <extension vendor-name="datanucleus" key="extended-setting" value="USING HASH"/>
  </index>

  ...    
</class>
```

See also:-
- MetaData reference for <index> element
- Annotations reference for @Index
- Annotations reference for @Index (class level)

### 78.1.2 Unique constraints

**Applicable to RDBMS, NeoDatis, MongoDB.**

Some datastores provide the ability to have unique constraints defined on tables to give extra control over data integrity. JDO provides a mechanism for defining such unique constraints. Lets take the previous class, and show how to specify this:

```xml
<class name="Booking">
  <field name="bookingType">
    <unique name="BOOKING_TYPE_CONSTRAINT"/>
  </field>

</class>
```

So in an identical way to the specification of an index. This example specification will result in the column(s) for "bookingType" being enforced as unique in the datastore. In the same way you can specify unique constraints directly to columns - see the example above for indexes.
Again, as for index, you can also specify unique constraints at "class" level in the MetaData file. This is useful to specify where the composite of 2 or more columns or fields are unique. So with this example

```xml
<class name="Booking">
    <unique name="UNIQUE_PERF">
        <field name="performanceDate"/>
        <field name="startTime"/>
    </unique>
    <field name="performanceDate"/>
    <field name="startTime"/>
</class>
```

The table for Booking has a unique constraint on the columns for the fields `performanceDate` and `startTime`

See also: -
- MetaData reference for `<unique>` element
- Annotations reference for @Unique
- Annotations reference for @Unique (class level)

### 78.1.3 Foreign Keys

**Applicable to RDBMS**

When objects have relationships with one object containing, for example, a Collection of another object, it is common to store a foreign key in the datastore representation to link the two associated tables. Moreover, it is common to define behaviour about what happens to the dependent object when the owning object is deleted. Should the deletion of the owner cause the deletion of the dependent object maybe? Let's take an example

```java
public class Hotel {
    private Set rooms;
    ...
}

public class Room {
    private int numberOfBeds;
    ...
}
```

We now want to control the relationship so that it is linked by a named foreign key, and that we cascade delete the Room object when we delete the Hotel. We define the Meta-Data like this
So we now have given the datastore control over the cascade deletion strategy for objects stored in these tables. Please be aware that JDO2 provides Dependent Fields as a way of allowing cascade deletion. The difference here is that Dependent Fields is controlled by DataNucleus, whereas foreign key delete actions are controlled by the datastore (assuming the datastore supports it even)

DataNucleus provides an extension that can give significant benefit to users. This is provided via the PersistenceManagerFactory data.nucleus.rdbms.constraintCreateMode. This property has 2 values. The default is DataNucleus which will automatically decide which foreign keys are required to satisfy the relationships that have been specified, whilst utilising the information provided in the MetaData for foreign keys. The other option is JDO2 which will simply create foreign keys that have been specified in the MetaData file(s).

Note that the foreign-key for a 1-N FK relation can be specified as above, or under the element element. Note that the foreign-key for a 1-N Join Table relation is specified under field for the FK from owner to join table, and is specified under element for the FK from join table to element table.

In the special case of application-identity and inheritance there is a foreign-key from subclass to superclass. You can define this as follows

```
<class name="MySubClass">
  <inheritance>
    <join>
      <foreign-key name="ID_FK"/>
    </join>
  </inheritance>
</class>
```

See also :-
- MetaData reference for <foreignkey> element
- Annotations reference for @ForeignKey
- Deletion of related objects using FK constraints

### 78.1.4 Primary Keys

**Applicable to RDBMS**

In RDBMS datastores, it is accepted as good practice to have a primary key on all tables. You specify in other parts of the MetaData which fields are part of the primary key (if using application identity), or you define the name of the column DataNucleus should use for the primary key (if using datastore identity). What these other parts of the MetaData don't allow is specifying the constraint name for the primary key. You can specify this if you wish, like this
When the schema is generated for this table, the primary key will be given the specified name, and will use the column(s) specified by the identity type in use.

In the case where you have a 1-N/M-N relation using a join table you can specify the name of the primary key constraint used as follows:

```
<class name="Hotel">
  <field name="rooms">
    <collection element-type="com.mydomain.samples.hotel.Room"/>
    <join>
      <primary-key name="HOTEL_ROOM_PK"/>
    </join>
  </field>
</class>
```

This creates a PK constraint with name "HOTEL_ROOM_PK".

See also:

- MetaData reference for `<primary-key>` element
- Annotations reference for @PrimaryKey
- Annotations reference for @PrimaryKey (class level)
79 Enhancer

79.1 DataNucleus Enhancer

As is described in the Class Enhancement guide below, DataNucleus utilises the common technique of byte-code manipulation to make your normal Java classes "persistable". The mechanism provided by DataNucleus is to use an "enhancer" process to perform this manipulation before you use your classes at runtime. The process is very quick and easy.

How to use the DataNucleus Enhancer depends on what environment you are using. Below are some typical examples.

- Post-compilation
  - Using Maven via the DataNucleus Maven plugin
  - Using Ant
  - Manual invocation at the command line
  - Using the Eclipse DataNucleus plugin
- At runtime
  - Runtime Enhancement
  - Programmatically via an API

79.1.1 Maven

Maven operates from a series of plugins. There is a DataNucleus plugin for Maven that allows enhancement of classes. Go to the Download section of the website and download this. Once you have the Maven plugin, you then need to set any properties for the plugin in your pom.xml file. Some properties that you may need to change are below

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>metadataDirectory</td>
<td>${project.build.outputDirectory}</td>
<td>Directory to use for enhancement files (classes/mappings)</td>
</tr>
<tr>
<td>metadataIncludes</td>
<td>**/<em>.jdo, **/</em>.class</td>
<td>Fileset to include for enhancement (if not using persistence-unit)</td>
</tr>
<tr>
<td>metadataExcludes</td>
<td></td>
<td>Fileset to exclude for enhancement (if not using persistence-unit)</td>
</tr>
<tr>
<td>persistenceUnitName</td>
<td></td>
<td>Name of the persistence-unit to enhance (if not using metadataIncludes etc)</td>
</tr>
<tr>
<td>log4jConfiguration</td>
<td></td>
<td>Config file location for Log4J (if using it)</td>
</tr>
<tr>
<td>jdkLogConfiguration</td>
<td></td>
<td>Config file location for JDK1.4 logging (if using it)</td>
</tr>
<tr>
<td>alwaysDetachable</td>
<td>false</td>
<td>Whether to enhance all classes as detachable irrespective of metadata</td>
</tr>
<tr>
<td>verbose</td>
<td>false</td>
<td>Verbose output?</td>
</tr>
<tr>
<td>quiet</td>
<td>false</td>
<td>No output?</td>
</tr>
</tbody>
</table>
targetDirectory
fork true
Where the enhanced classes are written (default is to overwrite them)
fork true
Whether to fork the enhancer process. Note that if you are running on Windows and have a large number of classes/mapping-files then this will result in a large command line, so set this option to false to avoid hitting Windows limit on command line length
generatePK true
Generate a PK class (of name {MyClass}_PK) for cases where there are multiple PK fields yet no PK class is defined.
generateConstructor true
Generate a default constructor if not defined for the class being enhanced.
detachListener false
Whether to enhance classes to make use of a detach listener for attempts to access an undetached field (see below)

You will need to add datanucleus-api-jdo.jar into the CLASSPATH (of the plugin, or your project) for the enhancer to operate. Also if using JPA metadata then you also will need datanucleus-api-jpa.jar and persistence-api.jar in the CLASSPATH. You then run the Maven DataNucleus plugin, as follows

```
mvn datanucleus:enhance
```

This will enhance all classes found that correspond to the classes defined in the JDO files in your source tree. If you want to check the current status of enhancement you can also type

```
mvn datanucleus:enhance-check
```

Or alternatively, you could add the following to your POM
<build>
  ...
  <plugins>
  <plugin>
    <groupId>org.datanucleus</groupId>
    <artifactId>datanucleus-maven-plugin</artifactId>
    <version>3.3.0-release</version>
    <configuration>
      <log4jConfiguration>${basedir}/log4j.properties</log4jConfiguration>
      <verbose>true</verbose>
    </configuration>
    <executions>
      <execution>
        <phase>process-classes</phase>
        <goals>
          <goal>enhance</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
  </plugins>
  ...
</build>

So you then get auto-enhancement after each compile. Please refer to the Maven JDO guide for more details.

79.1.2 Ant

Ant provides a powerful framework for performing tasks and DataNucleus provides an Ant task to enhance classes. The DataNucleus Enhancer is in datanucleus-core.jar, and you need to make sure that the datanucleus-core.jar, datanucleus-api-jdo.jar, jdo-api.jar (and optionally log4j.jar) are in your CLASSPATH. If using JPA metadata then you will also need persistence-api.jar and datanucleus-api-jpa.jar in the CLASSPATH. In the DataNucleus Enhancer Ant task, the following parameters are available

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dir</td>
<td>Optional. Directory containing the JDO (class/metadata) files to use for enhancing. Uses ant build file directory if the parameter is not specified.</td>
</tr>
<tr>
<td>destination</td>
<td>Optional. Defining a directory where enhanced classes will be written. If omitted, the original classes are updated.</td>
</tr>
<tr>
<td>alwaysDetachable</td>
<td>Optional. Whether to enhance all classes as detachable irrespective of metadata</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>persistenceUnit</td>
<td>Optional. Defines the &quot;persistence-unit&quot; to enhance.</td>
</tr>
<tr>
<td>checkonly</td>
<td>Whether to just check the classes for enhancement status. Will respond for each class with &quot;ENHANCED&quot; or &quot;NOT ENHANCED&quot;. <strong>This will disable the enhancement process and just perform these checks.</strong></td>
</tr>
<tr>
<td>verbose</td>
<td>Whether to have verbose output.</td>
</tr>
<tr>
<td>quiet</td>
<td>Whether to have no output.</td>
</tr>
<tr>
<td>generatePK</td>
<td>Whether to generate PK classes as required.</td>
</tr>
<tr>
<td>generateConstructor</td>
<td>Whether to generate a default constructor as required.</td>
</tr>
<tr>
<td>detachListener</td>
<td>Whether to enhance classes to make use of a detach listener for attempts to access an undetached field (see below)</td>
</tr>
<tr>
<td>filesuffixes</td>
<td>Optional. Suffixes to accept for the input files. The Enhancer Ant Task will scan for the files having these suffixes under the directory specified by <code>dir</code> option. The value can include comma-separated list of suffixes. If using annotations you can have &quot;class&quot; included as a valid suffix here or use the <code>fileset</code>.</td>
</tr>
<tr>
<td>fileset</td>
<td>Optional. Defines the files to accept as the input files. Fileset enables finer control to which classes / metadata files are accepted to enhanced. If one or more files are found in the fileset, the Enhancer Ant Task will not scan for additional files defined by the option <code>filesuffixes</code>. For more information on defining a fileset, see <a href="#">Apache FileSet Manual</a>.</td>
</tr>
<tr>
<td>if</td>
<td>Optional. The name of a property that must be set in order to the Enhancer Ant Task to execute.</td>
</tr>
</tbody>
</table>

The enhancer task extends the Apache Ant **Java task**, thus all parameters available to the Java task are also available to the enhancer task.

So you could define something like the following, setting up the parameters `enhancer.classpath`, `jdo.file.dir`, and `log4j.config.file` to suit your situation (the `jdo.file.dir` is a directory containing the JDO files defining the classes to be enhanced). The classes specified by the XML Meta-Data files, together with the XML Meta-Data files must be in the CLASSPATH (*Please note that a CLASSPATH should contain a set of JAR's, and a set of directories. It should NOT explicitly include class files, and should NOT include parts of the package names. If in doubt please consult a Java book*).
You can also define the files to be enhanced using a fileset. When a fileset is defined, the Enhancer Ant Task will not scan for additional files, and the option filesuffixes is ignored.

You can disable the enhancement execution upon the existence of a property with the usage of the if parameter.

79.1.3 Manually
DataNucleus provides an Enhancer in datanucleus-core.jar. If you are building your application manually and want to enhance your classes you follow the instructions in this section. You invoke the enhancer as follows
java -cp classpath org.datanucleus.enhancer.DataNucleusEnhancer [options] [mapping-files] [class-files]
where options can be
-pu {persistence-unit-name} : Name of a "persistence-unit" to enhance the classes for
-dir {directory-name} : Name of a directory that contains all model classes/mapping-files to enhance
-d {target-dir-name} : Write the enhanced classes to the specified directory
-checkonly : Just check the classes for enhancement status
-v : verbose output
-q : quiet mode (no output, overrides verbose flag too)
-alwaysDetachable : enhance all classes as detachable irrespective of metadata
-generatePK {flag} : generate any PK classes where needed
  {{flag} should be true or false - default=true}
-generateConstructor {flag} : generate default constructor where needed
  {{flag} should be true or false - default=true}
-detachListener {flag} : see below {set to true if required}

where "mapping-files" and "class-files" are provided when not enhancing a persistence-unit,
and give the paths to the mapping files and class-files that define the classes being enhanced.

where classpath must contain the following
  datanucleus-core.jar
  datanucleus-api-jdo.jar
  jdo-api.jar
  log4j.jar (optional)
  persistence-api.jar (optional - if using JPA metadata)
your classes
your meta-data files

The input to the enhancer should be either a set of MetaData/class files or the name of the "persistence-unit" to enhance. In the first option, if any classes have annotations then they must be specified. All classes and MetaData files should be in the CLASSPATH when enhancing. To give an example of how you would invoke the enhancer

Linux/Unix :
-Dlog4j.configuration=file:log4j.properties
org.datanucleus.enhancer.DataNucleusEnhancer
**/*.jdo

Windows :
java -cp target\classes;lib\datanucleus-core.jar;lib\jdo-api.jar;lib\jdo-api-jdo.jar;lib\log4j.jar
-Dlog4j.configuration=file:log4j.properties
org.datanucleus.enhancer.DataNucleusEnhancer
target/classes/org/mydomain/mypackage1/package.jdo

[should all be on same line. Shown like this for clarity]
So you pass in your JDO MetaData files (and/or the class files which use annotations) as the final argument(s) in the list, and include the respective JAR's in the classpath (-cp). The enhancer responds as follows

```
DataNucleus Enhancer (version 3.2.0.m2) for API "JDO" using JRE "1.6"
DataNucleus Enhancer : Classpath
>> /home/andy/work/myproject/target/classes
>> /home/andy/work/myproject/lib/log4j.jar
>> /home/andy/work/myproject/lib/jdo-api.jar
>> /home/andy/work/myproject/lib/datanucleus-core.jar
>> /home/andy/work/myproject/lib/datanucleus-api-jdo.jar
ENHANCED (PersistenceCapable): org.mydomain.mypackage1.Pack
ENHANCED (PersistenceCapable): org.mydomain.mypackage1.Card
DataNucleus Enhancer completed with success for 2 classes. Timings : input=422 ms, enhance=490 ms, total=912 ms.
... Consult the log for full details
```

If you have errors here relating to "Log4J" then you must fix these first. If you receive no output about which class was ENHANCED then you should look in the DataNucleus enhancer log for errors. The enhancer performs much error checking on the validity of the passed MetaData and the majority of errors are caught at this point. You can also use the DataNucleus Enhancer to check whether classes are enhanced. To invoke the enhancer in this mode you specify the `checkonly` flag. This will return a list of the classes, stating whether each class is enhanced for persistence under JDO or not. The classes need to be in the CLASSPATH (Please note that a CLASSPATH should contain a set of JAR's, and a set of directories. It should NOT explicitly include class files, and should NOT include parts of the package names. If in doubt please consult a Java book).

### 79.1.4 Runtime Enhancement

Enhancement of persistent classes at runtime is possible when using JRE 1.5+. Runtime Enhancement requires the following runtime dependencies: DataNucleus Core library. To enable runtime enhancement, the `javaagent` option must be set in the java command line. For example,

```
java -javaagent:datanucleus-core.jar=-api=JDO Main
```

The statement above will mean that all classes, when being loaded, will be processed by the ClassFileTransformer (except class in packages "java.*", "javax.*", "org.datanucleus.*"). This means that it can be slow since the MetaData search algorithm will be utilised for each. To speed this up you can specify an argument to that command specifying the names of package(s) that should be processed (and all others will be ignored). Like this

```
java -javaagent:datanucleus-core.jar=-api=JDO,mydomain.mypackage1,mydomain.mypackage2 Main
```

so in this case only classes being loaded that are in `mydomain.mypackage1` and `mydomain.mypackage2` will be attempted to be enhanced.

Please take care over the following when using runtime enhancement

- When you have a class with a field of another persistable type make sure that you mark that field as "persistent" (@Persistent, or in XML) since with runtime enhancement at that point the related class is likely not yet enhanced so will likely not be marked as persistent otherwise. **Be explicit**
• If the agent jar is not found make sure it is specified with an absolute path.

### 79.1.5 Programmatic API

You could alternatively programmatically enhance classes from within your application. This is done as follows

```java
import javax.jdo.JDOEnhancer;

JDOEnhancer enhancer = JDOHelper.getEnhancer();
enhancer.setVerbose(true);
enhancer.addPersistenceUnit("MyPersistenceUnit");
 enhancer.enhance();
```

This will look in META-INF/persistence.xml and enhance all classes defined by that unit. Please note that you will need to load the enhanced version of the class into a different ClassLoader after performing this operation to use them. See this guide

### 79.2 Class enhancement

DataNucleus requires that all classes that are persisted implement `PersistenceCapable`, an interface defined by JDO. Why should we do this, Hibernate/TopLink dont need it? Well thats a simple question really

• DataNucleus uses this `PersistenceCapable` interface, and adds it using bytecode enhancement techniques so that you never need to actually change your classes. This means that you get **transparent persistence**, and your classes always remain your classes. ORM tools that use a mix of reflection and/or proxies are not totally transparent.

• DataNucleus' use of `PersistenceCapable` provides transparent change tracking. When any change is made to an object the change creates a notification to DataNucleus allowing it to be optimally persisted. ORM tools that dont have access to such change tracking have to use reflection to detect changes. The performance of this process will break down as soon as you read a large number of objects, but modify just a handful, with these tools having to compare all object states for modification at transaction commit time.

Why not also read this comparison of bytecode enhancement, and proxies. It gives a clear enough comparison of the relative benefits.

In a JDO-enabled application there are 3 categories of classes. These are `PersistenceCapable`, `PersistenceAware` and normal classes. The Meta-Data defines which classes fit into these categories. To give an example for JDO, we have 3 classes. The class A is to be persisted in the datastore. The class B directly updates the fields of class A but doesn't need persisting. The class C is not involved in the persistence process. We would define JDO MetaData for these classes like this
So our MetaData is mainly for those classes that are **PersistenceCapable** and are to be persisted to the datastore (we don't really need the *persistence-modifier* for these classes since this is the default). For **PersistenceAware** classes we simply notate that the class knows about persistence. We don't define MetaData for any class that has no knowledge of persistence.

JDO requires that all classes to be persisted must implement the **PersistenceCapable** interface. Users could manually do this themselves but this would impose work on them. JDO permits the use of a byte-code enhancer that converts the users normal classes to implement this interface. DataNucleus provides its own byte-code enhancer (this can be found in the *datanucleus-core.jar*). This section describes how to use this enhancer with DataNucleus. The DataNucleus enhancer fully implements JDO2 and so is the recommended choice when persisting using the JDO2 API. The enhancement process adds the necessary methods to the users class in order to implement **PersistenceCapable**.

The example above doesn't show all **PersistenceCapable** methods, but demonstrates that all added methods and fields are prefixed with "jdo" to distinguish them from the users own methods and fields. Also each persistent field of the class will be given a jdoGetXXX, jdoSetXXX method so that accesses of these fields are intercepted so that JDO can manage their "dirty" state.

The MetaData defines which classes are required to be persisted, and also defines which aspects of persistence each class requires. For example if a class has the **detachable**
attribute set to \texttt{true}, then that class will be enhanced to also implement \textit{Detachable}

\begin{tikzpicture}
\node[draw, label=left:MyClass] (m) {
\begin{tabular}{l}
field1 \\
field2
\end{tabular}
};
\node[draw, label=right:\texttt{\{PersistenceCapable, Detachable\}}] (c) at (4,0) {
\begin{tabular}{l}
field1 \\
field2 \\
.jdoFieldTypes \\
.jdoFieldNames \\
+jdoDetachedState \\
+jdoProvideField() \\
+jdoReplaceField() \\
+jdoCopyField() \\
+jdoSetfield1() \\
+jdoSetfield2() \\
+jdoGetfield1() \\
+jdoGetfield2() \\
+jdoGetObjectId() \\
+jdoReplaceDetachedState() \\
+jdoGetVersion()
\end{tabular}
};
\draw[->] (m) -- (c);
\end{tikzpicture}

Again, the example above doesn't show all methods added for the Detachable interface but the main thing to know is that the detached state (object id of the datastore object, the version of the datastore object when it was detached, and which fields were detached is stored in "jdoDetachedState"). Please see the JDO spec for more details.

\textbf{If the MetaData is changed in any way during development, the classes should always be recompiled and re-enhanced afterwards.}

\subsection*{79.2.1 Byte-Code Enhancement Myths}

Some groups (e.g Hibernate) perpetuated arguments against "byte-code enhancement" saying that it was somehow 'evil'. The most common were :-

\begin{itemize}
\item \textit{Slows down the code-test cycle}. This is erroneous since you only need to enhance just before test and the provided plugins for Ant, Eclipse and Maven all do the enhancement job automatically and rapidly.
\item \textit{Is less "lazy" than the proxy approach since you have to load the object as soon as you get a pointer to it}. In a 1-1 relation you \textbf{have to load} the object then since you would cause issues with null pointers otherwise. With 1-N relations you load the elements of the collection/map only when you access them and not the collection/map. Hardly an issue then is it!
\item \textit{Fail to detect changes to public fields unless you enhance your client code}. Firstly very few people will be writing code with public fields since it is bad practice in an OO design, and secondly, this is why we have "PersistenceAware" classes.
\end{itemize}

So as you can see, there are no valid reasons against byte-code enhancement, and the pluses are that runtime detection of dirty events on objects is much quicker, hence your persistence layer operates faster without any need for iterative reflection-based checks. The fact is that Hibernate itself also now has a mode whereby you can do bytecode enhancement although not the default mode of Hibernate. So maybe it wasn't so evil after all?
79.2.2 Decompilation

Many people will wonder what actually happens to a class upon bytecode enhancement. In simple terms the necessary methods and fields are added so as to implement PersistenceCapable. If you want to check this, just use a Java decompiler such as JD. It has a nice GUI allowing you to just select your class to decompile and shows you the source.

79.2.3 Detach Listener

By default when you access the field of a detached object the bytecode enhanced class will check if that field is detached and throw a JDODetachedFieldAccessException if it was not detached. An alternative to this is to register a listener for such exceptions, and enable use of this listener when enhancing your classes. To enhance your classes to do this set the detachListener to true and then register the listener like this

```java
org.datanucleus.util.DetachListener.setInstance(myListener);
```

where myListener is an instance of a class that extends/implements org.datanucleus.util.DetachListener
80 Datastore Schema

80.1 JDO : Datastore Schema

Some datastores have a well-defined structure and when persisting/retrieving from these datastores you have to have this schema in place. DataNucleus provides various controls for creation of any necessary schema components. This creation can be performed as follows:

- One off task before running your application using SchemaTool. This is the recommended option since it separates schema from operation.
- At runtime, auto-generating tables as it requires them
- At runtime, as a one-off generate-schema step

The thing to remember when using DataNucleus is that the schema is under your control. DataNucleus does not impose anything on you as such, and you have the power to turn on/off all schema components. Some Java persistence tools add various types of information to the tables for persisted classes, such as special columns, or meta information. DataNucleus is very unobtrusive as far as the datastore schema is concerned. It minimises the addition of any implementation artifacts to the datastore, and adds nothing (other than any datastore identities, and version columns where requested) to any schema tables.

80.1.1 Schema Auto-Generation at runtime

If you want to create the schema ("tables"+"columns"+"constraints") during the persistence process, the property datanucleus.autoCreateScheme provides a way of telling DataNucleus to do this. It's a shortcut to setting the other 3 properties to true. Thereafter, during calls to DataNucleus to persist classes or performs queries of persisted data, whenever it encounters a new class to persist that it has no information about, it will use the MetaData to check the datastore for presence of the "table", and if it doesn't exist, will create it. In addition it will validate the correctness of the table (compared to the MetaData for the class), and any other constraints that it requires (to manage any relationships). If any constraints are missing it will create them.

- If you wanted to only create the "tables" required, and none of the "constraints" the property datanucleus.autoCreateTables provides this, simply performing the tables part of the above.
- If you want to create any missing "columns" that are required, the property datanucleus.autoCreateColumns provides this, validating and adding any missing columns.
- If you wanted to only create the "constraints" required, and none of the "tables" the property datanucleus.autoCreateConstraints provides this, simply performing the "constraints" part of the above.
- If you want to keep your schema fixed (i.e don't allow any modifications at runtime) then the property datanucleus.fixedDatastore should be set to true and this has the effect of setting the "table"+"columns"+"constraints" autoCreate values to false

80.1.2 Schema Generation for persistence-unit

DataNucleus allows you to generate the schema for your persistence-unit when creating a PMF. You enable this by specifying the persistence property datanucleus.generateSchema.mode to create (other values drop which deletes the schema, and drop-and-create which deletes and recreates the schema). When you create your PMF it will generate the schema before it returns your PMF.
You have some additional control over whether to actually create the schema, or whether to just output DDL for the schema (which you could apply yourself) - see the persistence property `datanucleus.generateSchema.target` which can be set to `scripts` to just create the DDL.

For RDBMS there are a few extensions here that are also worthy of mention, in that you can define scripts that are run during this schema generation phase. These are controlled by the following persistence properties:

- `datanucleus.generateSchema.scripts.create.source` - set this to an SQL script of your own that will create some tables (prior to any schema generation from the persistable objects)
- `datanucleus.generateSchema.scripts.drop.source` - set this to an SQL script of your own that will drop some tables (prior to any schema generation from the persistable objects)
- `datanucleus.generateSchema.scripts.load` - set this to an SQL script of your own that will insert any data that you require to be available when your PMF is initialised

### 80.1.3 Schema Generation : Validation

DataNucleus can check any existing schema against what is implied by the MetaData.

The property `datanucleus.validateTables` provides a way of telling DataNucleus to validate any tables that it needs against their current definition in the datastore. If the user already has a schema, and want to make sure that their tables match what DataNucleus requires (from the MetaData definition) they would set this property to `true`. This can be useful for example where you are trying to map to an existing schema and want to verify that you’ve got the correct MetaData definition.

The property `datanucleus.validateColumns` provides a way of telling DataNucleus to validate any columns of the tables that it needs against their current definition in the datastore. If the user already has a schema, and want to make sure that their tables match what DataNucleus requires (from the MetaData definition) they would set this property to `true`. This will validate the precise column types and widths etc, including defaultability/nullability settings. **Please be aware that many JDBC drivers contain bugs that return incorrect column detail information and so having this turned off is sometimes the only option (dependent on the JDBC driver quality).**

The property `datanucleus.validateConstraints` provides a way of telling DataNucleus to validate any constraints (primary keys, foreign keys, indexes) that it needs against their current definition in the datastore. If the user already has a schema, and want to make sure that their table constraints match what DataNucleus requires (from the MetaData definition) they would set this property to `true`.

### 80.1.4 Schema Generation : Naming Issues

Some datastores allow access to multiple "schemas" (such as with most RDBMS). DataNucleus will, by default, use the "default" database schema for the Connection URL and user supplied. This may cause issues where the user has been set up and in some databases (e.g Oracle) you want to write to a different schema (which that user has access to). To achieve this in DataNucleus you would set the persistence properties:

```
  datanucleus.mapping.Catalog={the_catalog_name}
  datanucleus.mapping.Schema={the_schema_name}
```

This will mean that all RDBMS DDL and SQL statements will prefix table names with the necessary catalog and schema names (specify which ones your datastore supports).
80.1.5 Schema Generation : Column Ordering

By default all tables are generated with columns in alphabetical order, starting with root class fields followed by subclass fields (if present in the same table) etc. There is a JDO3.1 attribute that allows you to specify the order of columns for schema generation. This is not present in JPA. It is achieved by specifying the metadata attribute position against the column.

```xml
<column position="1"/>
```

Note that the values of the position start at 0, and should be specified completely for all columns of all fields.

80.1.6 Read-Only

If your datastore is read-only (you can't add/update/delete any data in it), obviously you could just configure your application to not perform these operations. An alternative is to set the PMF as "read-only". You do this by setting the persistence property `javax.jdo.option.ReadOnly` to true.

From now on, whenever you perform a persistence operation that implies a change in datastore data, the operation will throw a `JDOReadOnlyException`.

DataNucleus provides an additional control over the behaviour when an attempt is made to change a read-only datastore. The default behaviour is to throw an exception. You can change this using the persistence property `datanucleus.readOnlyDatastoreAction` with values of "EXCEPTION" (default), and "IGNORE". "IGNORE" has the effect of simply ignoring all attempted updates to readonly objects.

You can take this read-only control further and specify it just on specific classes. Like this

```java
@Extension(vendorName="datanucleus", key="read-only", value="true")
public class MyClass {...}
```

80.2 SchemaTool

DataNucleus SchemaTool currently works with RDBMS, HBase, Excel, OOXML, ODF, MongoDB datastores and is very simple to operate. It has the following modes of operation :

- **create** - create all database tables required for the classes defined by the input data.
- **delete** - delete all database tables required for the classes defined by the input data.
- **deletecreate** - delete all database tables required for the classes defined by the input data, then create the tables.
- **validate** - validate all database tables required for the classes defined by the input data.
- **dbinfo** - provide detailed information about the database, it's limits and datatypes support. Only for RDBMS currently.
- **schemainfo** - provide detailed information about the database schema. Only for RDBMS currently.

Note that for RDBMS, the **create/ delete** modes can also be used by adding "--ddlFile {filename}" and this will then not create/delete the schema, but instead output the DDL for the tables/constraints into the specified file.
For the **create**, **delete** and **validate** modes DataNucleus SchemaTool accepts either of the following types of input:

- A set of MetaData and class files. The MetaData files define the persistence of the classes they contain. The class files are provided when the classes have annotations.
- The name of a **persistence-unit**. The **persistence-unit** name defines all classes, metadata files, and jars that make up that unit. Consequently, running DataNucleus SchemaTool with a persistence unit name will create the schema for all classes that are part of that unit.

Here we provide many different ways to invoke **DataNucleus SchemaTool**

- **Invoke it using Maven**, with the DataNucleus Maven plugin
- **Invoke it using Ant**, using the provided DataNucleus SchemaTool Ant task
- **Invoke it manually from the command line**
- **Invoke it using the DataNucleus Eclipse plugin**
- **Invoke it programmatically from within an application**

### 80.2.1 Maven

If you are using Maven to build your system, you will need the DataNucleus Maven plugin. This provides 5 goals representing the different modes of **DataNucleus SchemaTool**. You can use the goals **datanucleus:schema-create**, **datanucleus:schema-delete**, **datanucleus:schema-validate** depending on whether you want to create, delete or validate the database tables. To use the DataNucleus Maven plugin you will may need to set properties for the plugin (in your `pom.xml`). For example

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>metadataDirectory</td>
<td><code>${project.build.outputDirectory}</code></td>
<td>Directory to use for schema generation files (classes/mappings)</td>
</tr>
<tr>
<td>metadataIncludes</td>
<td>**/<em>.jdo, **/</em>.class</td>
<td>Fileset to include for schema generation</td>
</tr>
<tr>
<td>metadataExcludes</td>
<td></td>
<td>Fileset to exclude for schema generation</td>
</tr>
<tr>
<td>persistenceUnitName</td>
<td></td>
<td>Name of the persistence-unit to generate the schema for (defines the classes and the properties defining the datastore)</td>
</tr>
<tr>
<td>props</td>
<td></td>
<td>Name of a properties file for the datastore (PMF)</td>
</tr>
<tr>
<td>log4jConfiguration</td>
<td></td>
<td>Config file location for Log4J (if using it)</td>
</tr>
<tr>
<td>jdkLogConfiguration</td>
<td></td>
<td>Config file location for JDK1.4 logging (if using it)</td>
</tr>
<tr>
<td>api</td>
<td>JDO</td>
<td>API in use for metadata (JDO, JPA)</td>
</tr>
<tr>
<td>verbose</td>
<td>false</td>
<td>Verbose output?</td>
</tr>
</tbody>
</table>

©2015, DataNucleus • ALL RIGHTS RESERVED.
Whether to fork the SchemaTool process. Note that if you don’t fork the process, DataNucleus will likely struggle to determine class names from the input filenames, so you need to use a persistence.xml file defining the class names directly.

Name of an output file to dump any DDL to (for RDBMS)

Whether to generate DDL including things that already exist? (for RDBMS)

Whether to include auto-start mechanisms in SchemaTool usage

So to give an example, I add the following to my pom.xml (note: in 3.2.0.m1 it is "maven-datanucleus-plugin")

```xml
<build>
  ...
  <plugins>
    <plugin>
      <groupId>org.datanucleus</groupId>
      <artifactId>datanucleus-maven-plugin</artifactId>
      <version>3.2.0-m3</version>
      <configuration>
        <props>${basedir}/datanucleus.properties</props>
        <log4jConfiguration>${basedir}/log4j.properties</log4jConfiguration>
        <verbose>true</verbose>
      </configuration>
    </plugin>
    ... 
  </plugins>
  ...
</build>
```

So with these properties when I run SchemaTool it uses properties from the file datanucleus.properties at the root of the Maven project. I am also specifying a log4j configuration file defining the logging for the SchemaTool process. I then can invoke any of the Maven goals

```
mvn datanucleus:schema-create  Create the Schema
mvn datanucleus:schema-delete  Delete the schema
mvn datanucleus:schema-deletecreate Delete and create the schema
mvn datanucleus:schema-validate Validate the Schema
mvn datanucleus:schema-info     Output info for the Schema
mvn datanucleus:schema-dbinfo   Output info for the datastore
```

### 80.2.2 Ant

An Ant task is provided for using DataNucleus SchemaTool. It has classname org.datanucleus.store.schema.SchemaToolTask, and accepts the following parameters
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>Mode of operation.</td>
<td>create, delete, validate, dbinfo, schemainfo</td>
</tr>
<tr>
<td>verbose</td>
<td>Whether to give verbose output.</td>
<td>true, false</td>
</tr>
<tr>
<td>props</td>
<td>The filename to use for PMF properties</td>
<td></td>
</tr>
<tr>
<td>ddlFile</td>
<td>The filename where SchemaTool should output the DDL (for RDBMS).</td>
<td></td>
</tr>
<tr>
<td>completeDdl</td>
<td>Whether to output complete DDL (instead of just missing tables). Only used with ddlFile</td>
<td>true, false</td>
</tr>
<tr>
<td>includeAutoStart</td>
<td>Whether to include any auto-start mechanism in SchemaTool usage</td>
<td>true, false</td>
</tr>
<tr>
<td>api</td>
<td>API that we are using for metadata</td>
<td>JDO</td>
</tr>
<tr>
<td>persistenceUnit</td>
<td>Name of the persistence-unit that we should manage the schema for (defines the classes and the properties defining the datastore).</td>
<td></td>
</tr>
</tbody>
</table>

The SchemaTool task extends the Apache Ant Java task, thus all parameters available to the Java task are also available to the SchemaTool task.

In addition to the parameters that the Ant task accepts, you will need to set up your CLASSPATH to include the classes and MetaData files, and to define the following system properties via the sysproperty parameter (not required when specifying the persistence props via the properties file, or when providing the persistence-unit)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>datanucleus.ConnectionDriverName</td>
<td>Name of JDBC driver class</td>
<td>Mandatory</td>
</tr>
<tr>
<td>datanucleus.ConnectionURL</td>
<td>URL for the database</td>
<td>Mandatory</td>
</tr>
<tr>
<td>datanucleus.ConnectionUserName</td>
<td>User name for the database</td>
<td>Mandatory</td>
</tr>
<tr>
<td>datanucleus.ConnectionPassword</td>
<td>Password for the database</td>
<td>Mandatory</td>
</tr>
<tr>
<td>datanucleus.Mapping</td>
<td>ORM Mapping name</td>
<td>Optional</td>
</tr>
<tr>
<td>log4j.configuration</td>
<td>Log4J configuration file, for SchemaTool's Log</td>
<td>Optional</td>
</tr>
</tbody>
</table>

So you could define something like the following, setting up the parameters schematool.classpath, datanucleus.ConnectionDriverName, datanucleus.ConnectionURL, datanucleus.ConnectionUserName, and datanucleus.ConnectionPassword to suit your situation.

You define the jdo files to create the tables using filesset.
80.2.3 Manual Usage

If you wish to call DataNucleus SchemaTool manually, it can be called as follows
java [-cp classpath] [system_props] org.datanucleus.store.schema.SchemaTool [modes] [options] [props] [mapping-files] [class-files]

where system_props (when specified) should include
-Ddatanucleus.ConnectionDriverName=db_driver_name
-Ddatanucleus.ConnectionURL=db_url
-Ddatanucleus.ConnectionUserName=db_username
-Ddatanucleus.ConnectionPassword=db_password
-Ddatanucleus.Mapping=orm_mapping_name (optional)
-Dlog4j.configuration=file:(log4j.properties) (optional)

where modes can be
-create : Create the tables specified by the mapping-files/class-files
-delete : Delete the tables specified by the mapping-files/class-files
-deletecreate : Delete the tables specified by the mapping-files/class-files and then create them
-validate : Validate the tables specified by the mapping-files/class-files
-dbinfo : Detailed information about the database
-schemainfo : Detailed information about the database schema

where options can be
-ddlFile {filename} : RDBMS - only for use with "create"/"delete" mode to dump the DDL to the specified file
-completeDdl : RDBMS - when using "ddlFile" in "create" mode to get all DDL output and not just missing tables/constraints
-includeAutoStart : whether to include any auto-start mechanism in SchemaTool usage
-api : The API that is being used (default is JDO, but can be set to JPA)
-pu {persistence-unit-name} : Name of the persistence unit to manage the schema for
-v : verbose output

where props can be
-props {propsfilename} : PMF properties to use in place of the "system_props"

All classes, MetaData files, "persistence.xml" files must be present in the CLASSPATH. In terms of the schema to use, you either specify the "props" file (recommended), or you specify the System properties defining the database connection, or the properties in the "persistence-unit". You should only specify one of the [modes] above. Let's make a specific example and see the output from SchemaTool. So we have the following files in our application

src/java/... (source files and MetaData files)
target/classes/... (enhanced classes, and MetaData files)
lib/log4j.jar (optional, for Log4J logging)
lib/datanucleus-core.jar
lib/datanucleus-api-jdo.jar
lib/datanucleus-rdbms.jar, lib/datanucleus-hbase.jar, etc
lib/jdo-api.jar
lib/mysql-connector-java.jar (driver for the datastore, whether RDBMS, HBase etc)
log4j.properties

So we want to create the schema for our persistent classes. So let's invoke DataNucleus SchemaTool to do this, from the top level of our project. In this example we're using Linux (change the CLASSPATH definition to suit for Windows)
   -Dlog4j.configuration=file:log4j.properties
   org.datanucleus.store.schema.SchemaTool -create
   -props datanucleus.properties
   target/classes/org/datanucleus/examples/normal/package.jdo
   target/classes/org/datanucleus/examples/inverse/package.jdo

DataNucleus SchemaTool (version 3.2.0.m1) : Creation of the schema

DataNucleus SchemaTool : Classpath
>> /home/andy/work/DataNucleus/samples/packofcards/target/classes
>> /home/andy/work/DataNucleus/samples/packofcards/lib/log4j.jar
>> /home/andy/work/DataNucleus/samples/packofcards/lib/datanucleus-core.jar
>> /home/andy/work/DataNucleus/samples/packofcards/lib/datanucleus-api-jdo.jar
>> /home/andy/work/DataNucleus/samples/packofcards/lib/datanucleus-rdbms.jar
>> /home/andy/work/DataNucleus/samples/packofcards/lib/jdo-api.jar
>> /home/andy/work/DataNucleus/samples/packofcards/lib/mysql-connector-java.jar

DataNucleus SchemaTool : Input Files
>> /home/andy/work/DataNucleus/samples/packofcards/target/classes/org/datanucleus/examples/inverse/package.jdo
>> /home/andy/work/DataNucleus/samples/packofcards/target/classes/org/datanucleus/examples/normal/package.jdo

DataNucleus SchemaTool : Taking JDO properties from file "datanucleus.properties"

SchemaTool completed successfully

So as you see, **DataNucleus SchemaTool** prints out our input, the properties used, and finally a success message. If an error occurs, then something will be printed to the screen, and more information will be written to the log.

### 80.2.4 SchemaTool API

DataNucleus SchemaTool can also be called programmatically from an application. You need to get hold of the StoreManager and cast it to *SchemaAwareStoreManager*. The API is shown below.

```java
package org.datanucleus.store.schema;

public interface SchemaAwareStoreManager {
    public int createSchema(Set<String> classNames, Properties props);
    public int deleteSchema(Set<String> classNames, Properties props);
    public int validateSchema(Set<String> classNames, Properties props);
}
```

So for example to create the schema for classes `mydomain.A` and `mydomain.B` you would do something like this
JDOPersistenceManagerFactory pmf =
        (JDOPersistenceManagerFactory)JDOHelper.getPersistenceManagerFactory("datanucleus.properties");
NucleusContext ctx = pmf.getNucleusContext();
...
List classNames = new ArrayList();
classNames.add("mydomain.A");
classNames.add("mydomain.B");
try
{
    Properties props = new Properties();
    // Set any properties for schema generation
    ((SchemaAwareStoreManager)ctx.getStoreManager()).createSchema(classNames, props);
}
catch(Exception e)
{
    ...
}
81 Bean Validation

81.1 JDO : Bean Validation

The Bean Validation API (JSR0303) can be hooked up with JDO (DataNucleus extension) so that you have validation of an objects values prior to persistence, update and deletion. To do this

- Put the `javax.validation` "validation-api" jar in your CLASSPATH, along with the Bean Validation implementation jar of your choice (Apache BVAL, Hibernate Validator, etc)
- Set the persistence property `datanucleus.validation.mode` to one of `auto`, `none` (default), or `callback`
- Optionally set the persistence property(s) `datanucleus.validation.group.pre-persist`, `datanucleus.validation.group.pre-update`, `datanucleus.validation.group.pre-remove` to fine tune the behaviour (the default is to run validation on pre-persist and pre-update if you don't specify these).
- Use JDO as you normally would for persisting objects

To give a simple example of what you can do with the Bean Validation API

```java
@PersistenceCapable
public class Person {
    @PrimaryKey
    @NotNull
    private Long id;

    @NotNull
    @Size(min = 3, max = 80)
    private String name;

    ...
}
```

So we are validating that instances of the `Person` class will have an "id" that is not null and that the "name" field is not null and between 3 and 80 characters. If it doesn't validate then at persist/update an exception will be thrown.

Note that this was added to DataNucleus AccessPlatform in version 3.0.3
82 PersistenceManagerFactory

82.1 JDO : PersistenceManagerFactory

Any JDO-enabled application will require at least one PersistenceManagerFactory (PMF). Typically applications create one per datastore being utilised. A PersistenceManagerFactory provides access to PersistenceManagers which allow objects to be persisted, and retrieved. The PersistenceManagerFactory can be configured to provide particular behaviour.

Important: A PersistenceManagerFactory is designed to be thread-safe. A PersistenceManager is not.

There are many ways of creating a PersistenceManagerFactory

```
Properties properties = new Properties();
properties.setProperty("javax.jdo.PersistenceManagerFactoryClass", "org.datanucleus.api.jdo.JDOPersistenceManagerFactory");
properties.setProperty("javax.jdo.option.ConnectionURL", "jdbc:mysql://localhost/myDB");
properties.setProperty("javax.jdo.option.ConnectionDriverName", "com.mysql.jdbc.Driver");
properties.setProperty("javax.jdo.option.ConnectionUserName", "login");
properties.setProperty("javax.jdo.option.ConnectionPassword", "password");
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);
```

A slight variation on this, is to have a file to specify these properties like this

```
javax.jdo.PersistenceManagerFactoryClass=org.datanucleus.api.jdo.JDOPersistenceManagerFactory
javax.jdo.option.ConnectionURL=jdbc:mysql://localhost/myDB
javax.jdo.option.ConnectionDriverName=com.mysql.jdbc.Driver
javax.jdo.option.ConnectionUserName=login
javax.jdo.option.ConnectionPassword=password
```

and then to create the PersistenceManagerFactory using this file

```
File propsFile = new File(filename);
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(propsFile);
```

or if the above file is in the CLASSPATH (at "datanucleus.properties" in the root of the CLASSPATH), then

```
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory("datanucleus.properties");
```

If using a named PMF file, you can create the PMF by providing the name of the PMF like this

```
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory("myNamedPMF");
```

If using a META-INF/persistence.xml file, you can simply specify the persistence-unit name as

```
```
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory("myPersistenceUnit");

Another alternative, when specifying your datastore via JNDI, would be to call
JDOHelper.getPersistenceManagerFactory(jndiLocation, context);, and then set the other persistence properties on the received PMF.

Whichever way we wish to obtain the PersistenceManagerFactory we have defined a series of properties to give the behaviour of the PersistenceManagerFactory. The first property specifies to use the DataNucleus implementation, and the following 4 properties define the datastore that it should connect to. There are many properties available. Some of these are standard JDO properties, and some are DataNucleus extensions.

82.1.1 Specifying the datastore properties

With JDO you have 3 ways of specifying the datastore via persistence properties

- Specify the connection URL/driverName/username/password and it will internally create a DataSource for this URL (with optional connection pooling). This is achieved by specifying javax.jdo.option.ConnectionDriverName, javax.jdo.option.ConnectionURL, javax.jdo.option.ConnectionUserName, and javax.jdo.option.ConnectionPassword
- Specify the JNDI name of the connectionFactory This is achieved by specifying javax.jdo.option.ConnectionFactoryName, and javax.jdo.option.ConnectionFactory2Name (for secondary operations)
- Specify the DataSource of the connectionFactory This is achieved by specifying javax.jdo.option.ConnectionFactory, and javax.jdo.option.ConnectionFactory2 (for secondary operations)

82.1.2 JDO Persistence Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.jdo.PersistenceManagerFactory</td>
<td></td>
<td>The name of the PMF implementation. Only required if you have more than one JDO implementation in the CLASSPATH</td>
</tr>
<tr>
<td>javax.jdo.option.ConnectionFactory</td>
<td>Alias for datanucleus.ConnectionFactory</td>
<td></td>
</tr>
<tr>
<td>javax.jdo.option.ConnectionFactory2</td>
<td>Alias for datanucleus.ConnectionFactory2</td>
<td></td>
</tr>
<tr>
<td>javax.jdo.option.ConnectionFactoryName</td>
<td>Alias for datanucleus.ConnectionFactoryName</td>
<td></td>
</tr>
<tr>
<td>javax.jdo.option.ConnectionFactory2Name</td>
<td>Alias for datanucleus.ConnectionFactory2Name</td>
<td></td>
</tr>
<tr>
<td>javax.jdo.option.ConnectionDriverName</td>
<td>Alias for datanucleus.ConnectionDriverName</td>
<td></td>
</tr>
<tr>
<td>javax.jdo.option.ConnectionURL</td>
<td>Alias for datanucleus.ConnectionURL</td>
<td></td>
</tr>
<tr>
<td>javax.jdo.option.ConnectionUserName</td>
<td>Alias for datanucleus.ConnectionUserName</td>
<td></td>
</tr>
</tbody>
</table>
DataNucleus provides many properties to extend the control that JDO gives you. These can be used alongside the above standard JDO properties, but will only work with DataNucleus. Please consult the Persistence Properties Guide for full details.
82.2 PersistenceManagerFactory for Persistence-Unit

When designing an application you can usually nicely separate your persistable objects into independent groupings that can be treated separately, perhaps within a different DAO object, if using DAOs. JDO uses the (JPA) idea of a persistence-unit. A persistence-unit provides a convenient way of specifying a set of metadata files, and classes, and jars that contain all classes to be persisted in a grouping. The persistence-unit is named, and the name is used for identifying it. Consequently this name can then be used when defining what classes are to be enhanced, for example.

To define a persistence-unit you first need to add a file persistence.xml to the META-INF/ directory of the CLASSPATH (this may mean WEB-INF/classes/META-INF when using a web-application). This file will be used to define your persistence-units. Lets show an example

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
 http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd" version="2.1">
  <!-- Online Store -->
  <persistence-unit name="OnlineStore">
    <class>org.datanucleus.samples.metadata.store.Product</class>
    <class>org.datanucleus.samples.metadata.store.Book</class>
    <class>org.datanucleus.samples.metadata.store.CompactDisc</class>
    <class>org.datanucleus.samples.metadata.store.Customer</class>
    <class>org.datanucleus.samples.metadata.store.Supplier</class>
    <exclude-unlisted-classes/>
    <properties>
      <property name="datanucleus.ConnectionDriverName" value="org.h2.Driver"/>
      <property name="datanucleus.ConnectionURL" value="jdbc:h2:mem:datanucleus"/>
      <property name="datanucleus.ConnectionUserName" value="sa"/>
      <property name="datanucleus.ConnectionPassword" value=""/>
    </properties>
  </persistence-unit>

  <!-- Accounting -->
  <persistence-unit name="Accounting">
    <mapping-file>/com/datanucleus/samples/metadata/accounts/package.jdo</mapping-file>
    <properties>
      <property name="datanucleus.ConnectionDriverName" value="org.h2.Driver"/>
      <property name="datanucleus.ConnectionURL" value="jdbc:h2:mem:datanucleus"/>
      <property name="datanucleus.ConnectionUserName" value="sa"/>
      <property name="datanucleus.ConnectionPassword" value=""/>
    </properties>
  </persistence-unit>
</persistence>
```

In this example we have defined 2 persistence-units. The first has the name "OnlineStore" and contains 5 classes (annotated). The second has the name "Accounting" and contains a metadata file called "package.jdo" in a particular package (which will define the classes being part of that unit). This means that once we have defined this we can reference these persistence-units in our persistence operations. You can find the XSD for persistence.xml here.
There are several sub-elements of this persistence.xml file

- **provider** - Not used by JDO
- **jta-data-source** - JNDI name for JTA connections
- **non-jta-data-source** - JNDI name for non-JTA connections
- **jar-file** - name of a JAR file to scan for annotated classes to include in this persistence-unit.
- **mapping-file** - name of an XML "mapping" file containing persistence information to be included in this persistence-unit. This is the "JDO" mapping file (not the ORM)
- **class** - name of an annotated class to include in this persistence-unit
- **properties** - properties defining the persistence factory to be used.

### 82.2.1 Use with JDO

JDO accepts the "persistence-unit" name to be specified when creating the PersistenceManagerFactory, like this

```java
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory("MyPersistenceUnit");
```

### 82.2.2 Dynamically generated Persistence-Unit

DataNucleus allows an extension to JDO to dynamically create persistence-units at runtime. Use the following code sample as a guide. Obviously any classes defined in the persistence-unit need to have been enhanced.

```java
import org.datanucleus.metadata.PersistenceUnitMetaData;
import org.datanucleus.api.jdo.JDOPersistenceManagerFactory;
PersistenceUnitMetaData pumd = new PersistenceUnitMetaData("dynamic-unit", "RESOURCE_LOCAL", null);
pumd.addClassName("org.datanucleus.test.A");
pumd.setExcludeUnlistedClasses();
pumd.addProperty("javax.jdo.ConnectionDriverName", "org.hsqldb.jdbcDriver");
pumd.addProperty("javax.jdo.ConnectionUserName", "sa");
pumd.addProperty("javax.jdo.ConnectionPassword", "");
pumd.addProperty("datanucleus.autoCreateSchema", "true");
PersistenceManagerFactory pmf = new JDOPersistenceManagerFactory(pumd, null);
```

It should be noted that if you call `pumd.toString();` then this returns the text that would have been found in a persistence.xml file.

### 82.3 Named PersistenceManagerFactory

Typically applications create one PMF per datastore being utilised. An alternate to persistence-unit is to use a named PMF, defined in a file META-INF/jdoconfig.xml at the root of the CLASSPATH (this

©2015, DataNucleus • ALL RIGHTS RESERVED.
may mean WEB-INF/classes/META-INF when using a web-application). Let's see an example of a
\textit{jdoconfig.xml}

\begin{verbatim}
<?xml version="1.0" encoding="utf-8"?>
<jdoconfig xmlns="http://xmlns.jcp.org/xml/ns/jdo/jdoconfig"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="http://xmlns.jcp.org/xml/ns/jdo/jdoconfig">
  <!-- Datastore Txn PMF -->
  <persistence-manager-factory name="Datastore">
    <property name="javax.jdo.PersistenceManagerFactoryClass" value="org.datanucleus.api.jdo.JDOPersistenceManagerFactory"/>
    <property name="javax.jdo.option.ConnectionDriverName" value="com.mysql.jdbc.Driver"/>
    <property name="javax.jdo.option.ConnectionUserName" value="datanucleus"/>
    <property name="javax.jdo.option.ConnectionPassword" value=""/>
    <property name="javax.jdo.option.Optimistic" value="false"/>
    <property name="datanucleus.autoCreateSchema" value="true"/>
  </persistence-manager-factory>

  <!-- Optimistic Txn PMF -->
  <persistence-manager-factory name="Optimistic">
    <property name="javax.jdo.PersistenceManagerFactoryClass" value="org.datanucleus.api.jdo.JDOPersistenceManagerFactory"/>
    <property name="javax.jdo.option.ConnectionDriverName" value="com.mysql.jdbc.Driver"/>
    <property name="javax.jdo.option.ConnectionUserName" value="datanucleus"/>
    <property name="javax.jdo.option.ConnectionPassword" value=""/>
    <property name="javax.jdo.option.Optimistic" value="true"/>
    <property name="datanucleus.autoCreateSchema" value="true"/>
  </persistence-manager-factory>
</jdoconfig>
\end{verbatim}

So in this example we have 2 named PMFs. The first is known by the name "Datastore" and utilises
datastore transactions. The second is known by the name "Optimistic" and utilises optimistic
transactions. You simply define all properties for the particular PMF within its specification block.
And finally we instantiate our PMF like this

\begin{verbatim}
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory("Optimistic");
\end{verbatim}

That's it. The PMF we are returned from JDOHelper will have all of the properties defined in \textit{META-}
\textit{INF/jdoconfig.xml} under the name of "Optimistic".
83 L2 Cache

83.1 JDO : Caching

Caching is an essential mechanism in providing efficient usage of resources in many systems. Data management using JDO is no different and provides a definition of caching at 2 levels. Caching allows objects to be retained and returned rapidly without having to make an extra call to the datastore. The 2 levels of caching available with DataNucleus are

- **Level 1 Cache** - mandated by the JDO specification, and represents the caching of instances within a PersistenceManager
- **Level 2 Cache** - represents the caching of instances within a PersistenceManagerFactory (across multiple PersistenceManager's)

You can think of a cache as a Map, with values referred to by keys. In the case of JDO, the key is the object identity (identity is unique in JDO).

83.1.1 Level 2 Cache

By default the Level 2 Cache is enabled. The user can configure the Level 2 Cache if they so wish. This is controlled by use of the persistence property `datanucleus.cache.level2.type`. You set this to "type" of cache required. With the Level 2 Cache you currently have the following options.

- **none** - turn OFF Level 2 caching.
- **weak** - use the internal (weak reference based) L2 cache. Provides support for the JDO 2 interface of being able to pin objects into the cache, and unpin them when required. This option does not support distributed caching, solely running within the JVM of the client application. Weak references are held to non pinned objects.
- **soft** - use the internal (soft reference based) L2 cache. Provides support for the JDO 2 interface of being able to pin objects into the cache, and unpin them when required. This option does not support distributed caching, solely running within the JVM of the client application. Soft references are held to non pinned objects.
- **EHCache** - a simple wrapper to EHCache's caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.
- **EHCacheClassBased** - similar to the EHCache option but class-based.
- **OSCache** - a simple wrapper to OSCache's caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.
- **SwarmCache** - a simple wrapper to SwarmCache's caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.
- **Oracle Coherence** - a simple wrapper to Oracle's Coherence caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning. Oracle's caches support distributed caching, so you could, in principle, use DataNucleus in a distributed environment with this option.
- **javax.cache** - a simple wrapper to the standard `javax.cache`'s caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.
- **JCache** - a simple wrapper to the old version of `javax.cache`'s caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.
83 L2 Cache

- spymemcached - a simple wrapper to the "spymemcached" client for memcached caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.
- xmemcached - a simple wrapper to the "xmemcached" client for memcached caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.
- cacheonix - a simple wrapper to the Cacheonix distributed caching software. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.

The javax.cache cache is available in the datanucleus-core plugin. The EHCache, OSCache, SwarmCache, Coherence, JCache, Cacheonix, and Memcache caches are available in the datanucleus-cache plugin.

In addition you can control the mode of operation of the L2 cache. You do this using the persistence property datanucleus.cache.level2.mode. The default is UNSPECIFIED which means that DataNucleus will cache all objects of entities unless the entity is explicitly marked as not cacheable. The other options are NONE (don't cache ever), ALL (cache all entities regardless of annotations), ENABLE_SELECTIVE (cache entities explicitly marked as cacheable), or DISABLE_SELECTIVE (cache entities unless explicitly marked as not cacheable - i.e same as our default).

Objects are placed in the L2 cache when you commit() the transaction of a PersistenceManager. This means that you only have datastore-persisted objects in that cache. Also, if an object is deleted during a transaction then at commit it will be removed from the L2 cache if it is present.

The Level 2 cache is a DataNucleus plugin point allowing you to provide your own cache where you require it. Use the examples of the EHCache, Coherence caches etc as reference.

Note that you can have a PMF with L2 caching enabled yet have a PM with it disabled. This is achieved by creating the PM as you would normally, and then call

```
pm.setProperty("datanucleus.cache.level2.type", "none");
```

83.1.2 Controlling the Level 2 Cache

The majority of times when using a JDO-enabled system you will not have to take control over any aspect of the caching other than specification of whether to use a Level 2 Cache or not. With JDO and DataNucleus you have the ability to control which objects remain in the cache. This is available via a method on the PersistenceManagerFactory.

```
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(props);
DataStoreCache cache = pmf.getDataStoreCache();
```

The DataStoreCache interface

provides methods to control the retention of objects in the cache. You have 3 groups of methods

- **evict** - used to remove objects from the Level 2 Cache
- **pin** - used to pin objects into the cache, meaning that they will not get removed by garbage collection, and will remain in the Level 2 cache until removed.
- **unpin** - used to reverse the effects of pinning an object in the Level 2 cache. This will mean that the object can thereafter be garbage collected if not being used.
These methods can be called to pin objects into the cache that will be much used. Clearly this will be very much application dependent, but it provides a mechanism for users to exploit the caching features of JDO. If an object is not "pinned” into the L2 cache then it can typically be garbage collected at any time, so you should utilise the pinning capability for objects that you wish to retain access to during your application lifetime. For example, if you have an object that you want to be found from the cache you can do

```java
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(props);
DataStoreCache cache = pmf.getDataStoreCache();
cache.pinAll(MyClass.class, false); // Pin all objects of type MyClass from now on
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try
{
    tx.begin();
    pm.makePersistent(myObject);
    // *myObject* will now be pinned since we are pinning all objects of type MyClass.
    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.close();
    }
}
```

Thereafter, whenever something refers to *myObject*, it will find it in the Level 2 cache. To turn this behaviour off, the user can either unpin it or evict it.

JDO allows control over which classes are put into a Level 2 cache. You do this by specifying the cacheable attribute to *false* (defaults to true). So with the following specification, no objects of type *MyClass* will be put in the L2 cache.

Using XML:

```xml
<class name="MyClass" cacheable="false">
    ...
</class>
```

Using Annotations:

```java
@Cacheable("false")
public class MyClass
{
    ...
}
```

JDO allows you control over which fields of an object are put in the Level 2 cache. You do this by specifying the cacheable attribute to *false* (defaults to true). This setting is only required for fields that are relationships to other persistable objects. Like this
Using XML:
```xml
<class name="MyClass">
  <field name="values"/>
  <field name="elements" cacheable="false"/>
  ...
</class>
```

Using Annotations:
```java
public class MyClass
{
  ...
  Collection values;
  @Cacheable("false")
  Collection elements;
}
```

So in this example we will cache "values" but not "elements". If a field is cacheable then
- If it is a persistable object, the "identity" of the related object will be stored in the Level 2 cache
  for this field of this object
- If it is a Collection of persistable elements, the "identity" of the elements will be stored in the
  Level 2 cache for this field of this object
- If it is a Map of persistable keys/values, the "identity" of the keys/values will be stored in the
  Level 2 cache for this field of this object

When pulling an object in from the Level 2 cache and it has a reference to another object Access
Platform uses the "identity" to find that object in the Level 1 or Level 2 caches to re-relate the objects.

DataNucleus has an extension in metadata allowing the user to define that all instances of a class are
automatically pinned in the Level 2 cache.

```java
@PersistenceCapable
@Extension(vendorName="datanucleus", key="cache-pin", value="true")
public class MyClass
{
  ...
}
```

### 83.1.3 L2 Cache using javax.cache
DataNucleus provides a simple wrapper to `javax.cache`'s caches. To enable this you should set the
 persistence properties
```properties
datanucleus.cache.level2.type=javax.cache
datanucleus.cache.level2.cacheName={cache name}
datanucleus.cache.level2.timeout={expiration time in millis - optional}
```
83.1.4 L2 Cache using JCache
DataNucleus provides a simple wrapper to JCache's caches. This is an old version of what will become javax.cache (separate option). To enable this you should set the persistence properties

```
datanucleus.cache.level2.type=jcache
datanucleus.cache.level2.cacheName={cache name}
datanucleus.cache.level2.timeout={expiration time in millis - optional}
```

83.1.5 L2 Cache using Oracle Coherence
DataNucleus provides a simple wrapper to Oracle's Coherence caches. This currently takes the NamedCache interface in Coherence and instantiates a cache of a user provided name. To enabled this you should set the following persistence properties

```
datanucleus.cache.level2.type=coherence
datanucleus.cache.level2.cacheName={coherence cache name}
```

The Coherence cache name is the name that you would normally put into a call to CacheFactory.getCache(name). As mentioned earlier, this cache does not support the pin/unpin operations found in the standard JDO interface. However you do have the benefits of Oracle's distributed/serialized caching. If you require more control over the Coherence cache whilst using it with DataNucleus, you can just access the cache directly via

```
DataStoreCache cache = pmf.getDataStoreCache();
NamedCache coherenceCache = ((CoherenceLevel2Cache)cache).getCoherenceCache();
```

83.1.6 L2 Cache using EHCache
DataNucleus provides a simple wrapper to EHCache's caches. To enable this you should set the persistence properties

```
datanucleus.cache.level2.type=ehcache
datanucleus.cache.level2.cacheName={cache name}
datanucleus.cache.level2.configurationFile={EHCache configuration file (in classpath)}
```

The EHCache plugin also provides an alternative L2 Cache that is class-based. To use this you would need to replace "ehcache" above with "ehcacheclassbased".

83.1.7 L2 Cache using OSCache
DataNucleus provides a simple wrapper to OSCache's caches. To enable this you should set the persistence properties
83.1.8 L2 Cache using SwarmCache

DataNucleus provides a simple wrapper to SwarmCache's caches. To enable this you should set the persistence properties

```
datanucleus.cache.level2.type=swarmcache
datanucleus.cache.level2.cacheName={cache name}
```

83.1.9 L2 Cache using Spymemcached/Xmemcached

DataNucleus provides a simple wrapper to Spymemcached caches and Xmemcached caches. To enable this you should set the persistence properties

```
datanucleus.cache.level2.type=spymemcached       [or "xmemcached"]
datanucleus.cache.level2.cacheName={prefix for keys, to avoid clashes with other memcached objects}
datanucleus.cache.level2.memcached.servers=...
datanucleus.cache.level2.memcached.expireSeconds=...
```

`datanucleus.cache.level2.memcached.servers` is a space separated list of memcached hosts/ports, e.g. host:port host2:port. `datanucleus.cache.level2.memcached.expireSeconds` if not set or set to 0 then no expire

83.1.10 L2 Cache using Cacheonix

DataNucleus provides a simple wrapper to Cacheonix. To enable this you should set the persistence properties

```
datanucleus.cache.level2.type=cacheonix

datanucleus.cache.level2.cacheName={cache name}
```

Note that you can optionally also specify

```
datanucleus.cache.level2.timeout={timeout-in-millis (default=60)}
datanucleus.cache.level2.configurationFile={Cacheonix configuration file (in classpath)}
```

and define a `cacheonix-config.xml` like
<?xml version="1.0"?>
<cacheonix>
  <local>
    <!-- One cache per class being stored. -->
    <localCache name="mydomain.MyClass">
      <store>
        <lru maxElements="1000" maxBytes="#mb"/>
        <expiration timeToLive="60s"/>
      </store>
    </localCache>
    <!-- Fallback cache for classes indeterminable from their id. -->
    <localCache name="datanucleus">
      <store>
        <lru maxElements="1000" maxBytes="10mb"/>
        <expiration timeToLive="60s"/>
      </store>
    </localCache>
    <localCache name="default" template="true">
      <store>
        <lru maxElements="10" maxBytes="10mb"/>
        <overflowToDisk maxOverflowBytes="#mb"/>
        <expiration timeToLive="1s"/>
      </store>
    </localCache>
  </local>
</cacheonix>
84 Auto-Start

84.1 JDO : Automatic Startup

By default with JDO implementations when you open a PersistenceManagerFactory and obtain a PersistenceManager DataNucleus knows nothing about which classes are to be persisted to that datastore. JDO implementations only load the Meta-Data for any class when the class is first enlisted in a PersistenceManager operation. For example you call makePersistent on an object. The first time a particular class is encountered DataNucleus will dynamically load the Meta-Data for that class. This typically works well since in an application in a particular operation the PersistenceManagerFactory may well not encounter all classes that are persistable to that datastore. The reason for this dynamic loading is that JDO implementations can't be expected to scan through the whole Java CLASSPATH for classes that could be persisted there. That would be inefficient.

There are situations however where it is desirable for DataNucleus to have knowledge about what is to be persisted, or what subclasses of a candidate are possible on executing a query, so that it can load the Meta-Data at initialisation of the persistence factory and hence when the classes are encountered for the first time nothing needs doing. There are several ways of achieving this

- Define your classes/MetaData in a Persistence Unit and when the PersistenceManagerFactory is initialised it loads the persistence unit, and hence the MetaData for the defined classes and mapping files. This is described on the linked page
- Put the package.jdo at the root of the CLASSPATH, containing all classes, and when the first class is encountered it searches for its metadata, encounters and parses the root package.jdo, and consequently loads the metadata for all classes
- Use a DataNucleus extension known as Auto-Start Mechanism. This is set with the persistence property datanucleus.autoStartMechanism. This can be set to None, XML, Classes, MetaData. In addition we have SchemaTable for RDBMS datastores. These are described below.

84.1.1 AutoStartMechanism : None

With this property set to "None" DataNucleus will have no knowledge about classes that are to be persisted into that datastore and so will add the classes when the user utilises them in calls to the various PersistenceManager methods.

84.1.2 AutoStartMechanism : XML

With XML, DataNucleus stores the information for starting up DataNucleus in an XML file. This is, by default, located in datanucleusAutoStart.xml in the current working directory. The file name can be configured using the persistence factory property datanucleus.autoStartMechanismXmlFile. The file is read at startup and DataNucleus loads the classes using this information.

If the user changes their persistence definition a problem can occur when starting up DataNucleus. DataNucleus loads up its existing data from the XML configuration file and finds that a table/class required by the this file data no longer exists. There are 3 options for what DataNucleus will do in this situation. The property datanucleus.autoStartMechanismMode defines the behaviour of DataNucleus for this situation.

- **Checked** will mean that DataNucleus will throw an exception and the user will be expected to manually fix their database mismatch (perhaps by removing the existing tables).
- **Quiet** (the default) will simply remove the entry from the XML file and continue without exception.
84.1.3 AutoStartMechanism : Classes

With Classes, the user provides to the persistence factory the list of classes to use as the initial list of classes to be persisted. They specify this via the persistence property `datanucleus.autoStartClassNames`, specifying the list of classes as comma-separated. This gives DataNucleus a head start meaning that it will not need to "discover" these classes later.

84.1.4 AutoStartMechanism : MetaData

With MetaData, the user provides to the persistence factory the list of metadata files to use as the initial list of classes to be persisted. They specify this via the persistence property `datanucleus.autoStartMetaDataFiles`, specifying the list of metadata files as comma-separated. This gives DataNucleus a head start meaning that it will not need to "discover" these classes later.

84.1.5 AutoStartMechanism : SchemaTable (RDBMS only)

When using an RDBMS datastore the SchemaTable auto-start mechanism stores the list of classes (and their tables, types and version of DataNucleus) in a datastore table NUCLEUS_TABLES. This table is read at startup of DataNucleus, and provides DataNucleus with the necessary knowledge it needs to continue persisting these classes. This table is continuously updated during a session of a DataNucleus-enabled application.

If the user changes their persistence definition a problem can occur when starting up DataNucleus. DataNucleus loads up its existing data from NUCLEUS_TABLES and finds that a table/class required by the NUCLEUS_TABLES data no longer exists. There are 3 options for what DataNucleus will do in this situation. The property `datanucleus.autoStartMechanismMode` defines the behaviour of DataNucleus for this situation.

- **Checked** will mean that DataNucleus will throw an exception and the user will be expected to manually fix their database mismatch (perhaps by removing the existing tables).
- **Quiet** (the default) will simply remove the entry from NUCLEUS_TABLES and continue without exception.
- **Ignored** will simply continue without doing anything.

The default database schema used the SchemaTable is described below:

```xml
<datanucleus_autostart>
  <class name="mydomain.MyClass" table="MY_TABLE_1" type="FCO" version="3.1.1"/>
</datanucleus_autostart>
```
TABLE : NUCLEUS_TABLES
{
    COLUMN : CLASS_NAME VARCHAR(128) PRIMARY KEY, -- Fully qualified persistent Class name
    COLUMN : TABLE_NAME VARCHAR(128), -- Table name
    COLUMN : TYPE VARCHAR(4), -- FCO | SCO
    COLUMN : OWNER VARCHAR(2), -- 1 | 0
    COLUMN : VERSION VARCHAR(20), -- DataNucleus version
    COLUMN : INTERFACE_NAME VARCHAR(255) -- Fully qualified persistent Class type
        -- of the persistent Interface implemented
}

If you want to change the table name (from NUCLEUS_TABLES) you can set the persistence property datanucleus.rdbms.schemaTable.tableName
85 Data Federation

85.1 JDO : Data Federation

By default JDO provides a PersistenceManagerFactory (PMF) to represent a datastore. DataNucleus extends this to allow for a PMF to represent multiple datastores. This is intended for use where you have a data model for an application and maybe some classes are persisted into a different datastore. Note that this is work-in-progress and only tested for basic persist/retrieve operations using different schemas of the same datastore. Obviously if you have relations between one object in one datastore and another object in another datastore you cannot have foreign-keys (or equivalent).

85.1.1 Defining Primary and Secondary Datastores

You could specify the datastores to be used for the PMF like this. Here we have datanucleus.properties defining the primary datastore.

```java
javax.jdo.option.ConnectionDriverName=com.mysql.jdbc.Driver
javax.jdo.option.ConnectionUserName=mysql
javax.jdo.option.ConnectionPassword=

datanucleus.datastore.store2=datanucleus2.properties
```

You note that this refers to a store2, which is defined by datanucleus2.properties. So the secondary datastore is defined by

```java
javax.jdo.option.ConnectionURL=mongodb:/nucleus
```

85.1.2 Defining which class is persisted to which datastore

So now we need to notate which class is persisted to primary and which is persisted to secondary datastores. We do it like this, for the classes persisted to the secondary datastore.

```java
@PersistenceCapable
@Extension(vendorName="datanucleus", key="datastore", value="store2")
public class MyOtherClass
{
    ...
}
```

So for any persistence of objects of type MyOtherClass, they will be persisted into the MongoDB secondary datastore.
86 PersistenceManager

86.1 JDO: Persistence Manager

As you read in the guide for PersistenceManagerFactory, to control the persistence of your objects you will require at least one PersistenceManagerFactory. Once you have obtained this object you then use this to obtain a PersistenceManager (PM). A PersistenceManager provides access to the operations for persistence of your objects. This short guide will demonstrate some of the more common operations. For example with a web application you would have one PMF representing the datastore, present for the duration of the application, and then have a PM per request that comes in, closing it before responding.

Important: A PersistenceManagerFactory is designed to be thread-safe. A PersistenceManager is not.

You obtain a PersistenceManager as follows

```java
PersistenceManager pm = pmf.getPersistenceManager();
```

You likely will be performing all operations on a PersistenceManager within a transaction, whether your transactions are controlled by your JavaEE container, by a framework such as Spring, or by locally defined transactions. Alternatively you can perform your operations non-transactional. In the examples below we will omit the transaction demarcation for clarity.

86.1.1 Persisting an Object

The main thing that you will want to do with the data layer of a JDO-enabled application is persist your objects into the datastore. As we mentioned earlier, a PersistenceManagerFactory represents the datastore where the objects will be persisted. So you create a normal Java object in your application, and you then persist this as follows

```java
pm.makePersistent(obj);
```

This will result in the object being persisted into the datastore, though clearly it will not be persistent until you commit the transaction. The LifecycleState of the object changes from Transient to PersistentClean (after makePersistent), to Hollow (at commit).

86.1.2 Finding an object by its identity

Once you have persisted an object, it has an "identity". This is a unique way of identifying it. You can obtain the identity by calling

```java
Object id = pm.getObjectId(obj);
```

Alternatively by calling

```java
Object id = pm.newObjectIdInstance(cls, key);
```

So what? Well the identity can be used to retrieve the object again at some other part in your application. So you pass the identity into your application, and the user clicks on some button on a
web page and that button corresponds to a particular object identity. You can then go back to your
data layer and retrieve the object as follows

```java
Object obj = pm.getObjectById(id);
```

A DataNucleus extension is to pass in a String form of the identity to the above method. It accepts
identity strings of the form

- `{fully-qualified-class-name}:{key}`
- `{discriminator-name}:{key}`

where the key is the identity value (datastore-identity) or the result of PK.toString() (application-
identity). So for example we could input

```java
obj = pm.getObjectById("mydomain.MyClass:3");
```

There is, of course, a bulk load variant too

```java
Object[] objs = pm.getObjectsById(ids);
```

When you call the method `getObjectById` if an object with that identity is found in the cache then a
call is, by default, made to validate it still exists. You can avoid this call to the datastore by setting the
persistence property `datanucleus.findObject.validateWhenCached` to `false`.

### 86.1.3 Finding an object by its class and primary-key value

An alternate form of the `getObjectById` method is taking in the class of the object, and the "identity". This is for use where you have a single field that is primary key. Like this

```java
Object id = pm.getObjectId(MyClass.class, 123);
```

where 123 is the value of the primary key field (numeric). Note that the first argument could be a base
class and the real object could be an instance of a subclass of that.

### 86.1.4 Deleting an Object

When you need to delete an object that you had previous persisted, deleting it is simple. Firstly you
need to get the object itself, and then delete it as follows

```java
Object obj = pm.getObjectById(id); // Retrieves the object to delete
pm.deletePersistent(obj);
```

Don’t forget that you can also use deletion by query to delete objects. Alternatively use bulk deletion.

### 86.1.5 Modifying a persisted Object

To modify a previously persisted object you need to retrieve it (`getObjectById`, `query`, `getExtent`) and
then modify it and its changes will be propagated to the datastore at commit of the transaction.

Don’t forget that you can also use bulk update to update a group of objects of a type.
86.1.6 Detaching a persisted Object

You often have a previously persisted object and you want to use it away from the data-access layer of your application. In this case you want to *detach* the object (and its related objects) so that they can be passed across to the part of the application that requires it. To do this you do

```java
Object detachedObj = pm.detachCopy(obj); // Returns a copy of the persisted object, in detached state
```

The detached object is like the original object except that it has no StateManager connected, and it stores its JDO identity and version. It retains a list of all fields that are modified while it is detached. This means that when you want to "attach" it to the data-access layer it knows what to update.

As an alternative, to make the detachment process transparent, you can set the PMF property `datanucleus.DetachAllOnCommit` to true and when you commit your transaction all objects enlisted in the transaction will be detached.

86.1.7 Attaching a persisted Object

You've detached an object (shown above), and have modified it in your application, and you now want to attach it back to the persistence layer. You do this as follows

```java
Object attachedObj = pm.makePersistent(obj); // Returns a copy of the detached object, in attached state
```

86.1.8 Refresh of objects

In the situation where you have an object and you think that its values may have changed in the datastore you can update its values to the latest using the following

```java
pm.refresh(obj);
```

What this will do is as follows

- Refresh the values of all FetchPlan fields in the object
- Unload all non-FetchPlan fields in the object

If the object had any changes they will be thrown away by this step, and replaced by the latest datastore values.

86.1.9 Level 1 Cache

Each PersistenceManager maintains a cache of the objects that it has encountered (or have been "enlisted") during its lifetime. This is termed the **Level 1 Cache**. It is enabled by default and you should only ever disable it if you really know what you are doing. There are inbuilt types for the Level 1 (L1) Cache available for selection. DataNucleus supports the following types of L1 Cache:

- **weak** - uses a weak reference backing map. If JVM garbage collection clears the reference, then the object is removed from the cache.
- **soft** - uses a soft reference backing map. If the map entry value object is not being actively used, then garbage collection *may* garbage collect the reference, in which case the object is removed from the cache.
- **strong** - uses a normal HashMap backing. With this option all references are strong meaning that objects stay in the cache until they are explicitly removed by calling remove() on the cache.
• *none* - will turn off L1 caching. **Only ever use this where the cache is of no use and you are performing bulk operations and not requiring objects returned**

You can specify the type of L1 cache by providing the persistence property `datanucleus.cache.level1.type`. You set this to the value of the type required. If you want to remove objects from the L1 cache programmatically you should use the `pm.evict` or `pm.evictAll` methods.

Objects are placed in the L1 cache (and updated there) during the course of the transaction. This provides rapid access to the objects in use in the users application and is used to guarantee that there is only one object with a particular identity at any one time for that PersistenceManager. When the PersistenceManager is closed the L1 cache is cleared.

The L1 cache is a DataNucleus plugin point allowing you to provide your own cache where you require it.
87 PM Proxy

87.1 JDO : PersistenceManager Proxies

As you read in the guide for PersistenceManager, you perform all operations using a PersistenceManager. This means that you need to obtain this when you want to start datastore operations. In some architectures (e.g. in a web environment) it can be convenient to maintain a single PersistenceManager for use in a servlet init() method to initialise a static variable. Alternatively for use in a SessionBean to initialise a static variable. Thereafter you just refer to the proxy. The proxy isn’t the actual PersistenceManager just a proxy, delegating to the real object. If you call close() on the proxy the real PM will be closed, and when you next invoke an operation on the proxy it will create a new PM delegate and work with that.

To create a PM proxy is simple

```java
PersistenceManager pm = pmf.getPersistenceManagerProxy();
```

So we have our proxy, and now we can perform operations
88 Object Lifecycle

88.1 JDO: Object Lifecycle

During the persistence process, an object goes through lifecycle changes. Below we demonstrate the primary object lifecycle changes for JDO.

JDO has a very high degree of flexibility and so can be configured to operate in different modes. The mode most consistent with JPA is shown below (this has the PMF property `DetachAllOnCommit` set to true).

So a newly created object is transient. You then persist it and it becomes persistent. You then commit the transaction and it is detached for use elsewhere in the application. You then attach any changes back to persistence and it becomes persistent again. Finally when you delete the object from persistence and commit that transaction it is in transient state.

An alternative JDO lifecycle occurs when you have `DetachAllOnCommit` as false. Now at commit the object moves into hollow state (still has its identity, but its field values are optionally unloaded). Set the persistence property `datanucleus.RetainValues` to not unset the values of any non-primary-key fields when migrating to hollow state.

With JDO there are actually some additional lifecycle states, notably when an object has a field changed, becoming dirty, so you get an object in "persistent-dirty", "detached-dirty" states for example. The average user doesn't need to know about these so we don't cover them here. To inspect the lifecycle state of an object, simply call...
See also :-
  • Attach/Detach of objects

### 88.1.1 Helper Methods

In addition to the JDOHelper method above, JDO provides a series of other helper methods for lifecycle operations. These are documented on the [Apache JDO site](https://www.apache.org/jdo/).

Further to this DataNucleus provides yet more helper methods

```java
String[] fieldNames = NucleusJDOHelper.getDirtyFields(pc, pm);
String[] fieldNames = NucleusJDOHelper.getLoadedFields(pc, pm);
```

These methods returns the names of the dirty/loaded fields in the supplied object. The `pm` argument is only required if the object is detached

```java
Boolean dirty = NucleusJDOHelper.isDirty(pc, "fieldName", pm);
Boolean loaded = NucleusJDOHelper.isLoaded(pc, "fieldName", pm);
```

These methods returns whether the specified field in the supplied object is dirty/loaded. The `pm` argument is only required if the object is detached

### 88.1.2 State Transition Lookup

The JDO spec defines all lifecycles transitions. This table provides a summary of some of the common ones. Please refer to the JDO spec for details. Key: T-Clean = Transient Clean, T-Dirty = Transient Dirty, P-New = Persistent New, P-Clean = Persistent Clean, P-Dirty = Persistent Dirty, P-New-Deleted = Persistent New Deleted, P-Deleted = Persistent Deleted, P-Nontrans = Persistent Nontransactional

<table>
<thead>
<tr>
<th>Method / Current State</th>
<th>T-Clean</th>
<th>T-Dirty</th>
<th>P-New</th>
<th>P-Clean</th>
<th>P-Dirty</th>
<th>Hollow</th>
<th>P-New-Deleted</th>
<th>P-Deleted</th>
<th>P-Nontrans</th>
</tr>
</thead>
<tbody>
<tr>
<td>pm.makePersistent</td>
<td>P-New</td>
<td>no change</td>
<td>P-New</td>
<td>no change</td>
<td>P-Dirty</td>
<td>no change</td>
<td>no change</td>
<td>no change</td>
<td>no change</td>
</tr>
<tr>
<td>pm.delete</td>
<td>error</td>
<td>no change</td>
<td>P-New</td>
<td>P-New-Deleted</td>
<td>P-Del</td>
<td>no change</td>
<td>no change</td>
<td>no change</td>
<td>P-Del</td>
</tr>
<tr>
<td>pm.makeTransactional</td>
<td>no change</td>
<td>no change</td>
<td>P-New</td>
<td>no change</td>
<td>P-Dirty</td>
<td>no change</td>
<td>no change</td>
<td>no change</td>
<td>no change</td>
</tr>
<tr>
<td>pm.makeNontransaction</td>
<td>no change</td>
<td>no change</td>
<td>P-New</td>
<td>no change</td>
<td>P-Dirty</td>
<td>no change</td>
<td>no change</td>
<td>no change</td>
<td>no change</td>
</tr>
<tr>
<td>tx.commit, retainValue</td>
<td>T-Clean</td>
<td>Hollow</td>
<td>Hollow</td>
<td>Hollow</td>
<td>no change</td>
<td>T-Clean</td>
<td>T-Clean</td>
<td>no change</td>
<td>no change</td>
</tr>
<tr>
<td>tx.commit, non retainValue</td>
<td>T-Clean</td>
<td>P-Nontrans</td>
<td>P-Nontrans</td>
<td>P-Nontrans</td>
<td>no change</td>
<td>T-Clean</td>
<td>T-Clean</td>
<td>no change</td>
<td>no change</td>
</tr>
<tr>
<td>tx.commit</td>
<td>DetachAllOnCommit=true</td>
<td>no change</td>
<td>T-Clean</td>
<td>Detached-Clean</td>
<td>Detached-Clean</td>
<td>Detached-Clean</td>
<td>Detached-Clean</td>
<td>T-Clean</td>
<td>T-Clean</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------</td>
<td>-----------</td>
<td>--------</td>
<td>---------------</td>
<td>---------------</td>
<td>---------------</td>
<td>---------------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>tx.rollback</td>
<td>no change</td>
<td>T-Clean</td>
<td>T-Clean</td>
<td>Hollow</td>
<td>Hollow</td>
<td>no change</td>
<td>T-Clean</td>
<td>Hollow</td>
<td>no change</td>
</tr>
<tr>
<td>tx.rollback</td>
<td>no change</td>
<td>T-Clean</td>
<td>Transient</td>
<td>P-Nontrans</td>
<td>P-Nontrans</td>
<td>no change</td>
<td>Transient</td>
<td>P-Nontrans</td>
<td>no change</td>
</tr>
<tr>
<td>pm.refresh</td>
<td>no change</td>
<td>no change</td>
<td>P-Clean</td>
<td>no change</td>
<td>no change</td>
<td>no change</td>
<td>T-Clean</td>
<td>no change</td>
<td></td>
</tr>
<tr>
<td>pm.refresh</td>
<td>no change</td>
<td>no change</td>
<td>P-Clean</td>
<td>no change</td>
<td>no change</td>
<td>no change</td>
<td>P-Clean</td>
<td>no change</td>
<td></td>
</tr>
<tr>
<td>pm.evict</td>
<td>no change</td>
<td>no change</td>
<td>Hollow</td>
<td>no change</td>
<td>no change</td>
<td>no change</td>
<td>Hollow</td>
<td>no change</td>
<td></td>
</tr>
<tr>
<td>read field outside txn</td>
<td>no change</td>
<td>no change</td>
<td>no change</td>
<td>P-Nontrans</td>
<td>no change</td>
<td>no change</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read field active Datastore txn</td>
<td>no change</td>
<td>no change</td>
<td>no change</td>
<td>no change</td>
<td>P-Clean</td>
<td>error</td>
<td>error</td>
<td>P-Clean</td>
<td></td>
</tr>
<tr>
<td>read field active Optimistic txn</td>
<td>no change</td>
<td>no change</td>
<td>no change</td>
<td>no change</td>
<td>P-Nontrans</td>
<td>error</td>
<td>error</td>
<td>no change</td>
<td></td>
</tr>
</tbody>
</table>
| write field/makeDirty outside txn | no change | P-Nontrans | P-Nontrans-
<p>| read field active Datastore txn | no change | no change | P-Dirty | no change | P-Dirty | error | error | P-Dirty |
| retrieve() outside txn or with active Optimistic txn | no change | no change | no change | no change | no change | P-Nontrans | no change | no change |
| pm.retrieve with active Datastore txn | no change | no change | no change | no change | no change | no change | P-Clean | no change | P-Clean |</p>
<table>
<thead>
<tr>
<th>pm.detach error outside txn, Nontx-read=true</th>
<th>Detached-Clean</th>
<th>Detached-Clean</th>
</tr>
</thead>
<tbody>
<tr>
<td>pm.detach error outside txn, Nontx-read=false</td>
<td>error</td>
<td>Detached-Clean</td>
</tr>
<tr>
<td>pm.detach active txn</td>
<td>Detached-Clean</td>
<td>Detached-Clean</td>
</tr>
</tbody>
</table>
89 Lifecycle Callbacks

89.1 JDO: Lifecycle Callbacks
JDO defines a mechanism whereby a persistable class can be marked as a listener for lifecycle events. Alternatively a separate listener class can be defined to receive these events. Thereafter when entities of the particular class go through lifecycle changes events are passed to the provided methods. Let's look at the two different mechanisms.

89.1.1 Instance Callbacks
JDO defines an interface for PersistenceCapable classes so that they can be notified of events in their own lifecycle and perform any additional operations that are needed at these checkpoints. This is a complement to the Lifecycle Listeners interface which provides listeners for all objects of particular classes, with the events sent to a listener. With InstanceCallbacks the PersistenceCapable class is the destination of the lifecycle events. As a result the Instance Callbacks method is more intrusive than the method of Lifecycle Listeners in that it requires methods adding to each class that wishes to receive the callbacks.

DataNucleus supports the InstanceCallbacks interface.

To give an example of this capability, let us define a class that needs to perform some operation just before it's object is deleted.

```java
public class MyClass implements InstanceCallbacks
{
    String name;
    ...
    public void jdoPostLoad() {}
    public void jdoPreClear() {}
    public void jdoPreStore() {}
    public void jdoPreDelete()
    {
        // Perform some operation just before being deleted.
    }
}
```

So we have implemented InstanceCallbacks and have defined the 4 required methods. Only one of these is of importance in this example.

These methods will be called just before storage in the data store (jdoPreStore), just before clearing (jdoPreClear), just after being loaded from the datastore (jdoPostLoad) and just before being deleted (jdoPreDelete).
JDO2 adds 2 new callbacks to complement InstanceCallbacks. These are AttachCallback and DetachCallback. If you want to intercept attach/detach events your class can implement these interfaces. You will then need to implement the following methods

```java
public interface AttachCallback {
    public void jdoPreAttach();
    public void jdoPostAttach(Object attached);
}

public interface DetachCallback {
    public void jdoPreDetach();
    public void jdoPostDetach(Object detached);
}
```

89.1.2 Lifecycle Listeners

JDO defines an interface for the PersistenceManager and PersistenceManagerFactory whereby a user can register a listener for persistence events. The user provides a listener for either all classes, or a set of defined classes, and the JDO implementation calls methods on the listener when the required events occur. This provides the user application with the power to monitor the persistence process and, where necessary, append related behaviour. Specifying the listeners on the PersistenceManagerFactory has the benefits that these listeners will be added to all PersistenceManagers created by that factory, and so is for convenience really. This facility is a complement to the Instance Callbacks facility which allows interception of events on an instance by instance basis. The Lifecycle Listener process is much less intrusive than the process provided by Instance Callbacks, allowing a class external to the persistence process to perform the listening.

DataNucleus supports the InstanceLifecycleListener interface.

To give an example of this capability, let us define a Listener for our persistence process.
public class LoggingLifecycleListener implements CreateLifecycleListener, DeleteLifecycleListener, LoadLifecycleListener, StoreLifecycleListener {
    public void postCreate(InstanceLifecycleEvent event) {
        log.info("Lifecycle : create for " +
                ((PersistenceCapable)event.getSource()).jdoGetObjectId());
    }

    public void preDelete(InstanceLifecycleEvent event) {
        log.info("Lifecycle : preDelete for " +
                ((PersistenceCapable)event.getSource()).jdoGetObjectId());
    }

    public void postDelete(InstanceLifecycleEvent event) {
        log.info("Lifecycle : postDelete for " +
                ((PersistenceCapable)event.getSource()).jdoGetObjectId());
    }

    public void postLoad(InstanceLifecycleEvent event) {
        log.info("Lifecycle : load for " +
                ((PersistenceCapable)event.getSource()).jdoGetObjectId());
    }

    public void preStore(InstanceLifecycleEvent event) {
        log.info("Lifecycle : preStore for " +
                ((PersistenceCapable)event.getSource()).jdoGetObjectId());
    }

    public void postStore(InstanceLifecycleEvent event) {
        log.info("Lifecycle : postStore for " +
                ((PersistenceCapable)event.getSource()).jdoGetObjectId());
    }
}

Here we've provided a listener to receive events for CREATE, DELETE, LOAD, and STORE of objects. These are the main event types and in our simple case above we will simply log the event. All that remains is for us to register this listener with the PersistenceManager, or PersistenceManagerFactory

```java
pm.addInstanceLifecycleListener(new LoggingLifecycleListener(), null);
```

When using this interface the user should always remember that the listener is called within the same transaction as the operation being reported and so any changes they then make to the objects in question will be reflected in that objects state.

Register the listener with the PersistenceManager or PersistenceManagerFactory provide different effects. Registering with the PersistenceManagerFactory means that all PersistenceManagers created
by it will have the listeners registered on the PersistenceManagerFactory called. Registering the
listener with the PersistenceManager will only have the listener called only on events raised only by
the PersistenceManager instance.
The above diagram displays the sequence of actions for a listener registered only in the PersistenceManager. Note that a second PersistenceManager will not make calls to the listener registered in the first PersistenceManager.
The above diagram displays the sequence of actions for a listener registered in the PersistenceManagerFactory. All events raised in a PersistenceManager obtained
from the PersistenceManagerFactory will make calls to the listener registered in the PersistenceManagerFactory.

DataNucleus supports the following instance lifecycle listener types

- **AttachLifecycleListener** - all attach events
- **ClearLifecycleListener** - all clear events
- **CreateLifecycleListener** - all object create events
- **DeleteLifecycleListener** - all object delete events
- **DetachLifecycleListener** - all detach events
- **DirtyLifecycleListener** - all dirty events
- **LoadLifecycleListener** - all load events
- **StoreLifecycleListener** - all store events

The default JDO2 lifecycle listener **StoreLifecycleListener** only informs the listener of the object being stored. It doesn't provide information about the fields being stored in that event. DataNucleus extends the JDO2 specification and on the "preStore" event it will return an instance of `org.datanucleus.api.jdo.FieldInstanceLifecycleEvent` (which extends the JDO2 InstanceLifecycleEvent) and provides access to the names of the fields being stored.

```java
public class FieldInstanceLifecycleEvent extends InstanceLifecycleEvent {
    ...
    /**
     * Accessor for the field names affected by this event
     * @return The field names
     */
    public String[] getFieldNames()
    ...
}
```

If the store event is the persistence of the object then this will return all field names. If instead just particular fields are being stored then you just receive those fields in the event. So the only thing to do to utilise this DataNucleus extension is cast the received event to `org.datanucleus.FieldInstanceLifecycleEvent`
90 Attach/Detach

90.1 JDO: Attach/Detach

JDO provides an interface to the persistence of objects. JDO 1.0 doesn't provide a way of taking an object that was just persisted and just work on it and update the persisted object later. The user has to copy the fields manually and copy them back to the persisted object later. JDO 2.0 introduces a new way of handling this situation, by detach an object from the persistence graph, allowing it to be worked on in the users application. It can then be attached to the persistence graph later. Please refer to Object Lifecycle for where this fits in. The first thing to do to use a class with this facility is to tag it as "detachable". This is done by adding the attribute

```xml
<class name="MyClass" detachable="true"/>
```

This acts as an instruction to the enhancement process to add methods necessary to utilise the attach/detach process.

The following code fragment highlights how to use the attach/detach mechanism
Product working_product=null;
Transaction tx=pm.currentTransaction();
try
{
    tx.begin();

    Product prod=new Product(name,description,price);
    pm.makePersistent(prod);

    // Detach the product for use
    working_product = (Product)pm.detachCopy(prod);

    tx.commit();
}
catch (Exception e)
{
    // Handle the exception
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
}

// Work on the detached object in our application
working_product.setPrice(new_price);

...

// Reattach the updated object
tx = pm.currentTransaction();
try
{
    tx.begin();

    Product attached_product = pm.makePersistent(working_product);

    tx.commit();
}
catch (Exception e)
{
    // Handle the exception
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
}
So we now don't need to do any manual copying of object fields just using a simple call to detach the object, and then attach it again later. Here are a few things to note with attach/detach :-

- Calling `detachCopy` on an object that is not detachable will return a transient instance that is a COPY of the original, so use the COPY thereafter.
- Calling `detachCopy` on an object that is detachable will return a detached instance that is a COPY of the original, so use this COPY thereafter.
- A detached object retain the id of its datastore entity. Detached objects should be used where you want to update the objects and attach them later (updating the associated object in the datastore. If you want to create copies of the objects in the datastore with their own identities you should use `makeTransient` instead of `detachCopy`.
- Calling `detachCopy` will detach all fields of that object that are in the current Fetch Group for that class for that PersistenceManager.
- By default the fields of the object that will be detached are those in the Default Fetch Group.
- You should choose your Fetch Group carefully, bearing in mind which object(s) you want to access whilst detached. Detaching a relation field will detach the related object as well.
- If you don't detach a field of an object, you cannot access the value for that field while the object is detached.
- If you don't detach a field of an object, you can update the value for that field while detached, and thereafter you can access the value for that field.
- Calling `makePersistent` will return an (attached) copy of the detached object. It will attach all fields that were originally detached, and will also attach any other fields that were modified whilst detached.

When attaching an object graph (using `makePersistent()`) DataNucleus will, by default, make a check if each detached object has been detached from this datastore (since they could have been detached from a different datastore). This clearly can cause significant numbers of additional datastore activity with a large object graph. Consequently we provide a PMF property `datanucleus.attachSameDatastore` which, when set to true, will omit these checks and assume that we are attaching to the same datastore they were detached from.

To read more about attach/detach and how to use it with fetch-groups you can look at our Tutorial on DAO Layer design.

90.1.1 Detach All On Commit

JDO2 also provides a mechanism whereby all objects that were enlisted in a transaction are automatically detached when the transaction is committed. You can enable this in one of 3 ways. If you want to use this mode globally for all PersistenceManagers (PMs) from a PersistenceManagerFactory (PMF) you could either set the PMF property "datanucleus.DetachAllOnCommit", or you could create your PMF and call the PMF method `setDetachAllOnCommit(true)`. If instead you wanted to use this mode only for a particular PM, or only for a particular transaction for a particular PM, then you can call the PM method `setDetachAllOnCommit(true)` before the commit of the transaction, and it will apply for all transaction commits thereafter, until turned off ( `setDetachAllOnCommit(false)`). Here's an example
// Create a PMF
...

// Create an object
MyObject my = new MyObject();

PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try {
    tx.begin();
    // We want our object to be detached when it's been persisted
    pm.setDetachAllOnCommit(true);
    // Persist the object that we created earlier
    pm.makePersistent(my);
    tx.commit();
    // The object "my" is now in detached state and can be used further
} finally {
    if (tx.isActive) {
        tx.rollback();
    }
}

90.1.2 Copy On Attach

By default when you are attaching a detached object it will return an attached copy of the detached object. JDO2.1 provides a new feature that allows this attachment to just migrate the existing detached object into attached state.

You enable this by setting the PersistenceManagerFactory (PMF) property datanucleus.CopyOnAttach to false. Alternatively you can use the methods PersistenceManagerFactory.setCopyOnAttach(boolean flag) or PersistenceManager.setCopyOnAttach(boolean flag). If we return to the example at the start of this page, this now becomes
Please note that if you try to attach two detached objects representing the same underlying persistent object within the same transaction (i.e. a persistent object with the same identity already exists in the level 1 cache), then a JDOUserException will be thrown.

90.1.3 Detach On Close

A backup to the above programmatic detachment of instances is that when you close your PersistenceManager you can opt to have all instances currently cached in the Level 1 Cache of that PersistenceManager detached automatically. This means that you can persist instances, and then when you close the PM the instances will be detached and ready for further work. This is a DataNucleus extension. It is recommended that you use "detachAllOnCommit" since that is standard JDO and since this option will not work in J2EE environments where the PersistenceManager close is controlled by the J2EE container.

You enable this by setting the PersistenceManagerFactory (PMF) property datanucleus.DetachOnClose when you create the PMF. Let’s give an example
// Create a PMF with the datanucleus.DetachOnClose property set to "true"
...

// Create an object
MyObject my = new MyObject();

PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try {
    tx.begin();
    // Persist the object that we created earlier
    pm.makePersistent(my);
    tx.commit();
    pm.close();
    // The object "my" is now in detached state and can be used further
} finally {
    if (tx.isActive)
    {
        tx.rollback();
    }
}

That is about as close to transparent persistence as you will find. When the PM is closed all instances found in the L1 Cache are detached using the current FetchPlan, and so all fields in that plan for the instances in question will be detached at that time.

90.1.4 Detached Fields

When an object is detached it is typically passed to a different layer of an application and potentially changed. During the course of the operation of the system it may be required to know what is loaded in the object and what is dirty (has been changed since detaching). DataNucleus provides an extension to allow interrogation of the detached object.

```java
String[] loadedFieldNames = NucleusJDOHelper.getLoadedFields(obj, pm);
String[] dirtyFieldNames = NucleusJDOHelper.getDirtyFields(obj, pm);
```

So you have access to the names of the fields that were loaded when detaching the object, and also to the names of the fields that have been updated since detaching.

90.1.5 Serialization of Detachable classes

During enhancement of Detachable classes, a field called jdoDetachedState is added to the class definition. This field allows reading and changing tracking of detached objects while they are not managed by a PersistenceManager.
When serialization occurs on a Detachable object, the `jdoDetachedState` field is written to the serialized object stream. On deserialize, this field is written back to the new deserialized instance. This process occurs transparently to the application. However, if deserialization occurs with an un-enhanced version of the class, the detached state is lost.

Serialization and deserialization of Detachable classes and un-enhanced versions of the same class is only possible if the field `serialVersionUID` is added. It's recommended during development of the class, to define the `serialVersionUID` and make the class to implement the `java.io.Serializable` interface, as the following example:

```java
class MyClass implements java.io.Serializable
{
    private static final long serialVersionUID = 2765740961462495537L; // any random value here

    // .... other fields
}
```
91 Datastore Connection

91.1 JDO : Datastore Connections

DataNucleus utilises datastore connections as follows:

- **PMF**: single connection at any one time for datastore-based value generation. Obtained just for the operation, then released.
- **PMF**: single connection at any one time for schema-generation. Obtained just for the operation, then released.
- **PM**: single connection at any one time. When in a transaction the connection is held from the point of retrieval until the transaction commits or rolls back; the exact point at which the connection is obtained is defined more fully below. When used for non-transactional operations the connection is obtained just for the specific operation (unless configured to retain it).

If you have multiple threads using the same PersistenceManager then you can get “ConnectionInUse” problems where another operation on another thread comes in and tries to perform something while that first operation is still in use. This happens because the JDO spec requires an implementation to use a single datastore connection at any one time. When this situation crops up the user ought to use multiple PersistenceManagers.

Another important aspect is use of queries for Optimistic transactions, or for non-transactional contexts. In these situations it isn't possible to keep the datastore connection open indefinitely and so when the Query is executed the ResultSet is then read into core making the queried objects available thereafter.

91.1.1 Transactional Context

For pessimistic/datastore transactions a connection will be obtained from the datastore when the first persistence operation is initiated. This datastore connection will be held for the duration of the transaction until such time as either `commit()` or `rollback()` are called.

For optimistic transactions the connection is only obtained when `flush()/commit()` is called. When `flush()` is called, or the transaction committed a datastore connection is finally obtained and it is held open until `commit/rollback` completes. When a datastore operation is required. The connection is typically released after performing that operation. So datastore connections, in general, are held for much smaller periods of time. This is complicated slightly by use of the persistence property `java.jdo.option.IgnoreCache`. When this is set to `false`, the connection, once obtained, is not released until the call to `commit()/rollback()`.

Note that for Neo4j/MongoDB a single connection is used for the duration of the PM for all transactional and nontransactional operations.

91.1.2 Nontransactional Context

When performing non-transactional operations, the default behaviour is to obtain a connection when needed, and release it after use. With RDBMS you have the option of retaining this connection ready for the next operation to save the time needed to obtain it; this is enabled by setting the persistence property `datanucleus.connection.nontx.releaseAfterUse` to `false`.

Note that for Neo4j/MongoDB a single connection is used for the duration of the PM for all transactional and nontransactional operations.
91.1.3 User Connection

JDO defines a mechanism for users to access the native connection to the datastore, so that they can perform other operations as necessary. You obtain a connection as follows (for RDBMS):

```java
// Obtain the connection from the JDO implementation
JDOConnection conn = pm.getDataStoreConnection();
try {
    Object native = conn.getNativeConnection();

    // use the "sqlConn" connection to perform some operations.
} finally {
    // Hand the connection back to the JDO implementation
    conn.close();
}
```

For the datastores supported by DataNucleus, the "native" object is of the following types:

- **RDBMS**: java.sql.Connection
- **Excel**: org.apache.poi.hssf.usermodel.HSSFWorkbook
- **OOXML**: org.apache.poi.hssf.usermodel.XSSFWorkbook
- **ODF**: org.odftoolkit.odfdom.doc.OdfDocument
- **LDAP**: javax.naming.ldap.LdapContext
- **MongoDB**: com.mongodb.DB
- **HBase**: NOT SUPPORTED
- **JSON**: NOT SUPPORTED
- **XML**: org.w3c.dom.Document
- **NeoDatis**: org.neodatis.odb.ODB
- **GAE Datastore**: com.google.appengine.api.datastore.DatastoreService
- **Neo4j**: org.neo4j.graphdb.GraphDatabaseService

The "JDOConnection" in the case of DataNucleus is a wrapper to the native connection for the type of datastore being used. You now have a connection allowing direct access to the datastore. Things to bear in mind with this connection:

- You must return the connection back to the PersistenceManager before performing any JDO PM operation. You do this by calling `conn.close()`.
- If you don't return the connection and try to perform a JDO PM operation which requires the connection then a JDOUserException is thrown.

91.2 Connection Pooling: when specifying the connection via URL

When you create a `PersistenceManagerFactory` using a connection URL, driver name, and the username/password, this does not necessarily pool the connections (so they would be efficiently opened/closed when needed to utilise datastore resources in an optimum way). For some of the supported datastores DataNucleus allows you to utilise a connection pool to efficiently manage the
connections to the datastore when specifying the datastore via the URL. We currently provide support for the following:

- **RDBMS**: DBCP we allow use of externally-defined DBCP, but also provide a built-in DBCP v1.4
- **RDBMS**: C3P0
- **RDBMS**: Proxool
- **RDBMS**: BoneCP
- **RDBMS**: Tomcat
- **RDBMS**: Manually creating a DataSource for a third-party software package
- **RDBMS**: Custom Connection Pooling Plugins using the DataNucleus ConnectionPoolFactory interface
- **RDBMS**: Using JNDI, and lookup a connection DataSource.
- **LDAP**: Using JNDI

You need to specify the persistence property `datanucleus.connectionPoolingType` to be whichever of the external pooling libraries you wish to use (or "None" if you explicitly want no pooling). DataNucleus provides two sets of connections to the datastore - one for transactional usage, and one for non-transactional usage. If you want to define a different pooling for non-transactional usage then you can also specify the persistence property `datanucleus.connectionPoolingType.nontx` to whichever is required.

### 91.2.1 RDBMS : JDBC driver properties with connection pool

If using RDBMS and you have a JDBC driver that supports custom properties, you can still use DataNucleus connection pooling and you need to specify the properties in with your normal persistence properties, but add the prefix `datanucleus.connectionPool.driver.` to the property name that the driver requires. For example if an Oracle JDBC driver accepts `defaultRowPrefetch` then you would specify something like

```
datanucleus.connectionPool.driver.defaultRowPrefetch=50
```

and it will pass in `defaultRowPrefetch` as "50" into the driver used by the connection pool.

### 91.2.2 RDBMS : Apache DBCP

DataNucleus provides a built-in version of DBCP to provide pooling. This is automatically selected if using RDBMS, unless you specify otherwise. An alternative is to use an external DBCP (DBCPC). This is accessed by specifying the persistence property `datanucleus.connectionPoolingType` etc like this

```java
// Specify our persistence properties used for creating our PMF
Properties props = new Properties();
props.setProperty("datanucleus.ConnectionDriverName","com.mysql.jdbc.Driver");
props.setProperty("datanucleus.ConnectionURL","jdbc:mysql://localhost/myDB");
props.setProperty("datanucleus.ConnectionUserName","login");
props.setProperty("datanucleus.ConnectionPassword","password");
props.setProperty("datanucleus.connectionPoolingType","DBCP");
```
So the PMF will use connection pooling using DBCP. To do this you will need commons-dbcp, commons-pool and commons-collections JARs to be in the CLASSPATH.

You can also specify persistence properties to control the actual pooling. The currently supported properties for DBCP are shown below

```
# Pooling of Connections
datanucleus.connectionPool.maxIdle=10
datanucleus.connectionPool.minIdle=3
datanucleus.connectionPool.maxActive=5
datanucleus.connectionPool.maxWait=60

# Pooling of PreparedStatements
datanucleus.connectionPool.maxStatements=0

datanucleus.connectionPool.testSQL=SELECT 1

datanucleus.connectionPool.timeBetweenEvictionRunsMillis=2400000

datanucleus.connectionPool.minEvictableIdleTimeMillis=18000000
```

91.2.3 RDBMS : C3P0

DataNucleus allows you to utilise a connection pool using C3P0 to efficiently manage the connections to the datastore. C3P0 is a third-party library providing connection pooling. This is accessed by specifying the persistence property datanucleus.connectionPoolingType. To utilise C3P0-based connection pooling we do this

```
// Specify our persistence properties used for creating our PMF
Properties props = new Properties();
props.setProperty("datanucleus.ConnectionDriverName","com.mysql.jdbc.Driver");
props.setProperty("datanucleus.ConnectionURL","jdbc:mysql://localhost/myDB");
props.setProperty("datanucleus.ConnectionUserName","login");
props.setProperty("datanucleus.ConnectionPassword","password");
props.setProperty("datanucleus.connectionPoolingType", "C3P0");
```

So the PMF will use connection pooling using C3P0. To do this you will need the C3P0 JAR to be in the CLASSPATH. If you want to configure C3P0 further you can include a "c3p0.properties" in your CLASSPATH - see the C3P0 documentation for details.

You can also specify persistence properties to control the actual pooling. The currently supported properties for C3P0 are shown below

```
# Pooling of Connections
datanucleus.connectionPool.maxPoolSize=5
datanucleus.connectionPool.minPoolSize=3
datanucleus.connectionPool.initialPoolSize=3

# Pooling of PreparedStatements
datanucleus.connectionPool.maxStatements=0
```
91.2.4 RDBMS : Proxool

DataNucleus allows you to utilise a connection pool using Proxool to efficiently manage the connections to the datastore. Proxool is a third-party library providing connection pooling. This is accessed by specifying the persistence property `datanucleus.connectionPoolingType`. To utilise Proxool-based connection pooling we do this

```java
// Specify our persistence properties used for creating our PMF
Properties props = new Properties();
props.setProperty("datanucleus.ConnectionDriverName","com.mysql.jdbc.Driver");
props.setProperty("datanucleus.ConnectionURL","jdbc:mysql://localhost/myDB");
props.setProperty("datanucleus.ConnectionUserName","login");
props.setProperty("datanucleus.ConnectionPassword","password");
props.setProperty("datanucleus.connectionPoolingType", "Proxool");
```

So the PMF will use connection pooling using Proxool. To do this you will need the `proxool` and `commons-logging` JARs to be in the CLASSPATH.

You can also specify persistence properties to control the actual pooling. The currently supported properties for Proxool are shown below

```java
datanucleus.connectionPool.maxConnections=10
datanucleus.connectionPool.testSQL=SELECT 1
```

91.2.5 RDBMS : BoneCP

DataNucleus allows you to utilise a connection pool using BoneCP to efficiently manage the connections to the datastore. BoneCP is a third-party library providing connection pooling. This is accessed by specifying the persistence property `datanucleus.connectionPoolingType`. To utilise BoneCP-based connection pooling we do this

```java
// Specify our persistence properties used for creating our PMF
Properties props = new Properties();
props.setProperty("datanucleus.ConnectionDriverName","com.mysql.jdbc.Driver");
props.setProperty("datanucleus.ConnectionURL","jdbc:mysql://localhost/myDB");
props.setProperty("datanucleus.ConnectionUserName","login");
props.setProperty("datanucleus.ConnectionPassword","password");
props.setProperty("datanucleus.connectionPoolingType", "BoneCP");
```

So the PMF will use connection pooling using BoneCP. To do this you will need the `BoneCP` JAR (and SLF4J, google-collections) to be in the CLASSPATH.

You can also specify persistence properties to control the actual pooling. The currently supported properties for BoneCP are shown below
# Pooling of Connections

datanucleus.connectionPool.maxPoolSize=5

datanucleus.connectionPool.minPoolSize=3

# Pooling of PreparedStatements

datanucleus.connectionPool.maxStatements=0

91.2.6 RDBMS : Tomcat

DataNucleus allows you to utilise a connection pool using Tomcats JDBC Pool to efficiently manage the connections to the datastore. This is accessed by specifying the persistence property `datanucleus.connectionPoolingType`. To utilise Tomcat-based connection pooling we do this

```java
// Specify our persistence properties used for creating our PMF
Properties props = new Properties();
props.setProperty("datanucleus.ConnectionDriverName","com.mysql.jdbc.Driver");
props.setProperty("datanucleus.ConnectionURL","jdbc:mysql://localhost/myDB");
props.setProperty("datanucleus.ConnectionUserName","login");
props.setProperty("datanucleus.ConnectionPassword","password");
props.setProperty("datanucleus.connectionPoolingType", "Tomcat");
```

So the PMF will use a DataSource with connection pooling using Tomcat. To do this you will need the `tomcat-jdbc` JAR to be in the CLASSPATH.

You can also specify persistence properties to control the actual pooling, just like other pools.

91.2.7 RDBMS : Manually create a DataSource (e.g DBCP, C3P0, Proxool, etc)

We could have used the built-in DBCP support which internally creates a DataSource ConnectionFactory, alternatively the support for external DBCP, C3P0, Proxool, BoneCP etc, however we can also do this manually if we so wish. Let's demonstrate how to do this with one of the most used pools Apache Commons DBCP

With DBCP you need to generate a `javax.sql.DataSource`, which you will then pass to DataNucleus. You do this as follows
// Load the JDBC driver
Class.forName(dbDriver);

// Create the actual pool of connections
ObjectPool connectionPool = new GenericObjectPool(null);

// Create the factory to be used by the pool to create the connections
ConnectionFactory connectionFactory = new DriverManagerConnectionFactory(dbURL, dbUser, dbPassword);

// Create a factory for caching the PreparedStatements
KeyedObjectPoolFactory kpf = new StackKeyedObjectPoolFactory(null, 20);

// Wrap the connections with pooled variants
PoolableConnectionFactory pcf =
   new PoolableConnectionFactory(connectionFactory, connectionPool, kpf, null, false, true);

// Create the datasource
DataSource ds = new PoolingDataSource(connectionPool);

// Create our PMF
Map properties = new HashMap();
properties.put("javax.jdo.option.ConnectionFactory", ds);
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);

Note that we haven't passed the dbUser and dbPassword to the PMF since we no longer need to specify them - they are defined for the pool so we let it do the work. As you also see, we set the data source for the PMF. Thereafter we can sit back and enjoy the performance benefits. Please refer to the documentation for DBCP for details of its configurability (you will need commons-dbcp, commons-pool, and commons-collections in your CLASSPATH to use this above example).

91.2.8 RDBMS : Lookup a DataSource using JNDI

DataNucleus allows you to use connection pools (java.sql.DataSource) bound to a javax.naming.InitialContext with a JNDI name. You first need to create the DataSource in the container (application server/web server), and secondly you define the datanucleus.ConnectionFactoryName property with the DataSource JNDI name.

The following example uses a properties file that is loaded before creating the PersistenceManagerFactory. The PersistenceManagerFactory is created using the JDOHelper.

datanucleus.ConnectionFactoryName=YOUR_DATASOURCE_JNDI_NAME

Properties properties = new Properties();

// the properties file is in your classpath
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory("/yourpath/yourfile.properties");

Please read more about this in RDBMS DataSources.
91.2.9 LDAP : JNDI

If using an LDAP datastore you can use the following persistence properties to enable connection pooling

```java
datanucleus.connectionPoolingType=JNDI
```

Once you have turned connection pooling on if you want more control over the pooling you can also set the following persistence properties

- `datanucleus.connectionPool.maxPoolSize` : max size of pool
- `datanucleus.connectionPool.initialPoolSize` : initial size of pool

91.3 RDBMS : Data Sources

DataNucleus allows use of a data source that represents the datastore in use. This is often just a URL defining the location of the datastore, but there are in fact several ways of specifying this data source depending on the environment in which you are running.

- Nonmanaged Context - Java Client
- Managed Context - Servlet
- Managed Context - JEE

91.3.1 Java Client Environment : Non-managed Context

DataNucleus permits you to take advantage of using database connection pooling that is available on an application server. The application server could be a full JEE server (e.g WebLogic) or could equally be a servlet engine (e.g Tomcat, Jetty). Here we are in a non-managed context, and we use the following properties when creating our PersistenceManagerFactory, and refer to the JNDI data source of the server.

If the data source is available in WebLogic, the simplest way of using a data source outside the application server is as follows.

```java
Hashtable ht = new Hashtable();
ht.put(Context.INITIAL_CONTEXT_FACTORY,"weblogic.jndi.WLInitialContextFactory");
ht.put(Context.PROVIDER_URL,"t3://localhost:7001");
Context ctx = new InitialContext(ht);
DataSource ds = (DataSource) ctx.lookup("jdbc/datanucleus");

Map properties = new HashMap();
properties.setProperty("datanucleus.ConnectionFactory",ds);
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);
```

If the data source is available in Websphere, the simplest way of using a data source outside the application server is as follows.
Hashtable ht = new Hashtable();
ht.put(Context.INITIAL_CONTEXT_FACTORY, "com.ibm.websphere.naming.WsnInitialContextFactory");
ht.put(Context.PROVIDER_URL, "iiop://server:orb port");

Context ctx = new InitialContext(ht);
DataSource ds = (DataSource) ctx.lookup("jdbc/datanucleus");

Map properties = new HashMap();
properties.setProperty("javax.jdo.PersistenceManagerFactoryClass", "org.datanucleus.api.jdo.JDOPersistenceManagerFactory");
properties.setProperty("datanucleus.ConnectionFactory", ds);
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);

91.3.2 Servlet Environment : Managed Context

As an example of setting up such a JNDI data source for Tomcat 5.0, here we would add the following file to $TOMCAT/conf/Catalina/localhost/ as "datanucleus.xml"
With this Tomcat JNDI data source we would then specify the PMF ConnectionFactoryName as `java:comp/env/jdbc/datanucleus`.

```java
Properties properties = new Properties();
properties.setProperty("datanucleus.ConnectionFactoryName","java:comp/env/jdbc/datanucleus");
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);
```

### 91.3.3 JEE Environment: Managed Context

As in the above example, we can also run in a managed context, in a JEE/Servlet environment, and here we would make a minor change to the specification of the JNDI data source depending on the application server or the scope of the jndi: global or component.

Using JNDI deployed in global environment:
Properties properties = new Properties();
properties.setProperty("datanucleus.ConnectionFactoryName","jdbc/datanucleus");
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);

Using JNDI deployed in component environment:

Properties properties = new Properties();
properties.setProperty("datanucleus.ConnectionFactoryName","java:comp/env/jdbc/datanucleus");
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);

See also: JEE Tutorial for JDO
92 Transactions

92.1 JDO : Transactions

A Transaction forms a unit of work. The Transaction manages what happens within that unit of work, and when an error occurs the Transaction can roll back any changes performed. Transactions can be managed by the users application, or can be managed by a framework (such as Spring), or can be managed by a JEE container. These are described below.

- **Local transactions**: managed using the JDO Transaction API
- **JTA transactions**: managed using the JTA UserTransaction API, or using the JDO Transaction API
- **Container-managed transactions**: managed by a JEE environment
- **Spring-managed transactions**: managed by SpringFramework
- **No transactions**
- **Flushing a Transaction**
- **Controlling transaction isolation level**
- **Read-Only transactions**

92.1.1 Locally-Managed Transactions

When using a JDO implementation such as DataNucleus in a J2SE environment, the transactions are by default **Locally Managed Transactions**. The users code will manage the transactions by starting, and committing the transaction itself. With these transactions with JDO you would do something like

```java
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try
{
    tx.begin();

    // (users code to persist objects)

    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
}
pm.close();
```

The basic idea with **Locally-Managed transactions** is that you are managing the transaction start and end.
92.1.2 JTA Transactions

When using a JDO implementation such as DataNucleus in a J2SE environment, you can also make use of JTA Transactions. You define the persistence property `javax.jdo.option.TransactionType` setting it to "JTA". Then you make use of JTA (or JDO) to demarcate the transactions. So you could do something like

```java
UserTransaction ut = (UserTransaction)
        new InitialContext().lookup("java:comp/UserTransaction");
PersistenceManager pm = pmf.getPersistenceManager();
try {
    ut.begin();
    (users code to persist objects)
    ut.commit();
} finally {
    pm.close();
}
```

So here we used the JTA API to begin/commit the controlling (`javax.transaction.UserTransaction`). An alternative is where you don't have a UserTransaction started and just use the JDO API, which will start the UserTransaction for you.

```java
UserTransaction ut = (UserTransaction)
        new InitialContext().lookup("java:comp/UserTransaction");
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try {
    tx.begin(); // Starts the UserTransaction
    (users code to persist objects)
    tx.commit(); // Commits the UserTransaction
} finally {
    pm.close();
}
```

Important: please note that you need to set both transactional and nontransactional datasources, and the nontransactional cannot be JTA.

92.1.3 Container-Managed Transactions

When using a JEE container you are giving over control of the transactions to the container. Here you have Container-Managed Transactions. In terms of your code, you would do like the previous
example except that you would OMIT the `tx.begin()`, `tx.commit()`, `tx.rollback()` since the JEE container will be doing this for you.

### 92.1.4 Spring-Managed Transactions

When you use a framework like Spring you would not need to specify the `tx.begin()`, `tx.commit()`, `tx.rollback()` since that would be done for you.

### 92.1.5 No Transactions

DataNucleus allows the ability to operate without transactions. With JDO this is enabled by default (see the 2 properties `datanucleus.NontransactionalRead`, `datanucleus.NontransactionalWrite` set to `true`). This means that you can read objects and make updates outside of transactions. This is effectively "auto-commit" mode.

```java
PersistenceManager pm = pmf.getPersistenceManager();

{users code to persist objects}

pm.close();
```

When using non-transactional operations, you need to pay attention to the persistence property `datanucleus.nontx.atomic`. If this is true then any persist/delete/update will be committed to the datastore immediately. If this is false then any persist/delete/update will be queued up until the next transaction (or `pm.close()`) and committed with that.

### 92.1.6 Flushing

During a transaction, depending on the configuration, operations don't necessarily go to the datastore immediately, often waiting until commit. In some situations you need persists/updates/deletes to be in the datastore so that subsequent operations can be performed that rely on those being handled first. In this case you can flush all outstanding changes to the datastore using

```java
pm.flush();
```

A convenient vendor extension is to find which objects are waiting to be flushed at any time, like this

```java
List<ObjectProvider> objs =
    ((JDOPersistenceManager)pm).getExecutionContext().getObjectsToBeFlushed();
```
92.1.7 Transaction Isolation

JDO provides a mechanism for specification of the transaction isolation level. This can be specified globally via the PersistenceManagerFactory property `datanucleus.transactionIsolation` (javax.jdo.option.TransactionIsolationLevel). It accepts the following values:

- **read-uncommitted**: dirty reads, non-repeatable reads and phantom reads can occur
- **read-committed**: dirty reads are prevented; non-repeatable reads and phantom reads can occur
- **repeatable-read**: dirty reads and non-repeatable reads are prevented; phantom reads can occur
- **serializable**: dirty reads, non-repeatable reads and phantom reads are prevented

The default (in DataNucleus) is read-committed. An attempt to set the isolation level to an unsupported value (for the datastore) will throw a JDOUserException. As an alternative you can also specify it on a per-transaction basis as follows (using the names above).

```
Transaction tx = pm.currentTransaction();
...
tx.setIsolationLevel("read-committed");
```

92.1.8 JDO Transaction Synchronisation

There are situations where you may want to get notified that a transaction is in course of being committed or rolling back. To make that happen, you would do something like
```java
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try {
    tx.begin();

    tx.setSynchronization(new javax.transaction.Synchronization()
    {
        public void beforeCompletion()
        {
            // before commit or rollback
        }

        public void afterCompletion(int status)
        {
            if (status == javax.transaction.Status.STATUS_ROLLEDBACK)
            {
                // rollback
            }
            else if (status == javax.transaction.Status.STATUS_COMMITTED)
            {
                // commit
            }
        }
    });
    tx.commit();
} finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
}
pm.close();
```

### 92.1.9 Read-Only Transactions

Obviously transactions are intended for committing changes. If you come across a situation where you don’t want to commit anything under any circumstances you can mark the transaction as “read-only” by calling
Any call to *commit* on the transaction will throw an exception forcing the user to roll it back.

### 92.2 JDO: Transaction Locking

A Transaction forms a unit of work. The Transaction manages what happens within that unit of work, and when an error occurs the Transaction can roll back any changes performed. There are the following types of locking:

- Transactions can lock all records in a datastore and keep them locked until they are ready to commit their changes. These are known as **Pessimistic (or datastore) Locking**.
- Transactions can simply assume that things in the datastore will not change until they are ready to commit, not lock any records and then just before committing make a check for changes. This is known as **Optimistic Locking**.

#### 92.2.1 Pessimistic (Datastore) Locking

**Pessimistic** locking is the default in JDO. It is suitable for short lived operations where no user interaction is taking place and so it is possible to block access to datastore entities for the duration of the transaction.

By default DataNucleus does not currently lock the objects fetched with pessimistic locking, but you can configure this behaviour for RDBMS datastores by setting the persistence property `datanucleus.SerializeRead` to `true`. This will result in all "SELECT ... FROM ..." statements being changed to be "SELECT ... FROM ... FOR UPDATE". This will be applied only where the underlying RDBMS supports the "FOR UPDATE" syntax. This can be done on a transaction-by-transaction basis by doing

```java
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try {
    tx.begin();
    tx.setRollbackOnly();
    (users code to persist objects)
    tx.rollback();
} finally {
    if (tx.isActive()) {
        tx.rollback();
    }
}
pm.close();
```

Alternatively, on a per query basis, you would do
With pessimistic locking DataNucleus will grab a datastore connection at the first operation, and maintain it for the duration of the transaction. A single connection is used for the transaction (with the exception of any Identity Generation operations which need datastore access, so these can use their own connection).

In terms of the process of pessimistic (datastore) locking, we demonstrate this below.

<table>
<thead>
<tr>
<th>Operation</th>
<th>DataNucleus process</th>
<th>Datastore process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start transaction</td>
<td></td>
<td>Open connection.</td>
</tr>
<tr>
<td>Persist object</td>
<td>Prepare object (1) for persistence</td>
<td>Insert the object (1) into the datastore</td>
</tr>
<tr>
<td>Update object</td>
<td>Prepare object (2) for update</td>
<td>Update the object (2) into the datastore</td>
</tr>
<tr>
<td>Persist object</td>
<td>Prepare object (3) for persistence</td>
<td>Insert the object (3) into the datastore</td>
</tr>
<tr>
<td>Update object</td>
<td>Prepare object (4) for update</td>
<td>Update the object (4) into the datastore</td>
</tr>
<tr>
<td>Flush</td>
<td>No outstanding changes so do nothing</td>
<td></td>
</tr>
<tr>
<td>Perform query</td>
<td>Generate query in datastore language</td>
<td>Query the datastore and return selected objects</td>
</tr>
<tr>
<td>Persist object</td>
<td>Prepare object (5) for persistence</td>
<td>Insert the object (5) into the datastore</td>
</tr>
<tr>
<td>Update object</td>
<td>Prepare object (6) for update</td>
<td>Update the object (6) into the datastore</td>
</tr>
<tr>
<td>Commit transaction</td>
<td></td>
<td>Commit connection</td>
</tr>
</tbody>
</table>

So here whenever an operation is performed, DataNucleus pushes it straight to the datastore. Consequently any queries will always reflect the current state of all objects in use. However this mode of operation has no version checking of objects and so if they were updated by external processes in the meantime then they will overwrite those changes.

It should be noted that DataNucleus provides two persistence properties that allow an amount of control over when flushing happens with datastore transactions.

- `datanucleus.flush.mode` when set to MANUAL will try to delay all datastore operations until commit/flush.
- `datanucleus.datastoreTransactionFlushLimit` represents the number of dirty objects before a flush is performed. This defaults to 1.

### 92.2.2 Optimistic Locking

Optimistic locking is the other option in JDO. It is suitable for longer lived operations maybe where user interaction is taking place and where it would be undesirable to block access to datastore entities for the duration of the transaction. The assumption is that data altered in this transaction will...
Transactions

not be updated by other transactions during the duration of this transaction, so the changes are not propagated to the datastore until commit()/flush(). The data is checked just before commit to ensure the integrity in this respect. The most convenient way of checking data for updates is to maintain a column on each table that handles optimistic locking data. The user will decide this when generating their MetaData.

Rather than placing version/timestamp columns on all user datastore tables, JDO2 allows the user to notate particular classes as requiring optimistic treatment. This is performed by specifying in MetaData or annotations the details of the field/column to use for storing the version - see versioning for JDO. With JDO the version is added in a surrogate column, whereas a vendor extension allows you to have a field in your class ready to store the version.

In JDO2 the version is stored in a surrogate column in the datastore so it also provides a method for accessing the version of an object. You can call JDOHelper.getVersion(object) and this returns the version as an Object (typically Long or Timestamp). This will return null for a transient object, and will return the version for a persistent object. If the object is not PersistenceCapable then it will also return null.

In terms of the process of optimistic locking, we demonstrate this below.

<table>
<thead>
<tr>
<th>Operation</th>
<th>DataNucleus process</th>
<th>Datastore process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start transaction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Persist object</td>
<td>Prepare object (1) for persistence</td>
<td></td>
</tr>
<tr>
<td>Update object</td>
<td>Prepare object (2) for update</td>
<td></td>
</tr>
<tr>
<td>Persist object</td>
<td>Prepare object (3) for persistence</td>
<td></td>
</tr>
<tr>
<td>Update object</td>
<td>Prepare object (4) for update</td>
<td></td>
</tr>
<tr>
<td>Flush</td>
<td>Flush all outstanding changes to the datastore</td>
<td></td>
</tr>
</tbody>
</table>

Perform query

<table>
<thead>
<tr>
<th>Operation</th>
<th>DataNucleus process</th>
<th>Datastore process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate query in datastore language</td>
<td>Query the datastore and return selected objects</td>
<td></td>
</tr>
</tbody>
</table>

Persist object

<table>
<thead>
<tr>
<th>Operation</th>
<th>DataNucleus process</th>
<th>Datastore process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update object</td>
<td>Prepare object (6) for update</td>
<td></td>
</tr>
</tbody>
</table>

Commit transaction

<table>
<thead>
<tr>
<th>Operation</th>
<th>DataNucleus process</th>
<th>Datastore process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush</td>
<td>Flush all outstanding changes to the datastore</td>
<td></td>
</tr>
</tbody>
</table>

Here no changes make it to the datastore until the user either commits the transaction, or they invoke flush(). The impact of this is that when performing a query, by default, the results may not contain
the modified objects unless they are flushed to the datastore before invoking the query. Depending on whether you need the modified objects to be reflected in the results of the query governs what you do about that. If you invoke flush() just before running the query the query results will include the changes. The obvious benefit of optimistic locking is that all changes are made in a block and version checking of objects is performed before application of changes, hence this mode copes better with external processes updating the objects.

Please note that for some datastores (e.g RDBMS) the version check followed by update/delete is performed in a single statement.

See also :-

- JDO MetaData reference for <version> element
- JDO Annotations reference for @Version

92.2.3 Persistence-by-Reachability at commit()

When a transaction is committed JDO will, by default, run its reachability algorithm to check if any reachable objects have been persisted and are no longer reachable. If an object is found to be no longer reachable and was only persisted by being reachable (not by an explicit persist operation) then it will be removed from the datastore. You can turn off this reachability check for JDO by setting the persistence property datanucleus.persistenceByReachabilityAtCommit to false.
93 Fetch Groups

93.1 JDO : Fetch Groups

When an object is retrieved from the datastore by JDO typically not all fields are retrieved immediately. This is because for efficiency purposes only particular field types are retrieved in the initial access of the object, and then any other objects are retrieved when accessed (lazy loading). The group of fields that are loaded is called a fetch group. There are 3 types of "fetch groups" to consider

- **Default Fetch Group**: defined in all JDO specs, containing the fields of a class that will be retrieved by default (with no user specification).
- **Named Fetch Groups**: defined by the JDO2 specification, and defined in MetaData (XML/annotations) with the fields of a class that are part of that fetch group. The definition here is static
- **Dynamic Fetch Groups**: Programmatic definition of fetch groups at runtime via an API

The fetch group in use for a class is controled via the FetchPlan interface. To get a handle on the current FetchPlan we do

```
FetchPlan fp = pm.getFetchPlan();
```

93.1.1 Default Fetch Group

JDO provides an initial fetch group, comprising the fields that will be retrieved when an object is retrieved if the user does nothing to define the required behaviour. By default the default fetch group comprises all fields of the following types:

- primitives: boolean, byte, char, double, float, int, long, short
- Object wrappers of primitives: Boolean, Byte, Character, Double, Float, Integer, Long, Short
- java.lang.String, java.lang.Number, java.lang.Enum
- java.math.BigDecimal, java.math.BigInteger
- java.util.Date

If you wish to change the Default Fetch Group for a class you can update the Meta-Data for the class as follows (for XML)

```
<class name="MyClass">
...
  <field name="fieldX" default-fetch-group="true"/>
</class>
```

or using annotations

```
@Persistent(defaultFetchGroup="true")
SomeType fieldX;
```

When a PersistenceManager is created it starts with a FetchPlan of the "default" fetch group. That is, if we call

```
Collection fetchGroups = fp.getGroups();
```
this will have one group, called "default". At runtime, if you have been using other fetch groups and want to revert back to the default fetch group at any time you simply do

\[
fp.setGroup(FetchPlan.DEFAULT);
\]

### 93.1.2 Named Fetch Groups

As mentioned above, JDO2 allows specification of users own fetch groups. These are specified in the MetaData of the class. For example, if we have the following class

```java
class MyClass
{
    String name;
    HashSet col;
    MyOtherClass other;
}
```

and we want to have the other field loaded whenever we load objects of this class, we define our MetaData as

```xml
<package name="mydomain">
    <class name="MyClass">
        <field name="name">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="coll" persistence-modifier="persistent">
            <collection element-type="mydomain.Address"/>
        </field>
        <field name="other" persistence-modifier="persistent"/>
        <fetch-group name="otherfield">
            <field name="other"/>
        </fetch-group>
    </class>
</package>
```

or using annotations

```java
@PersistenceCapable
@FetchGroup(name="otherfield", members={@Persistent(name="other")})
public class MyClass
{
    ...
}
```

So we have defined a fetch group called "otherfield" that just includes the field with name `other`. We can then use this at runtime in our persistence code.
PersistenceManager pm = pmf.getPersistenceManager();
pm.getFetchPlan().addGroup("otherfield");

... (load MyClass object)

By default the FetchPlan will include the default fetch group. We have changed this above by adding the fetch group "otherfield", so when we retrieve an object using this PersistenceManager we will be retrieving the fields name AND other since they are both in the current FetchPlan. We can take the above much further than what is shown by defining nested fetch groups in the MetaData. In addition we can change the FetchPlan just before any PersistenceManager operation to control what is fetched during that operation. The user has full flexibility to add many groups to the current Fetch Plan. This gives much power and control over what will be loaded and when. A big improvement over JDO 1.0

The FetchPlan applies not just to calls to PersistenceManager.getObjectById(), but also to PersistenceManager.newQuery(), PersistenceManager.getExtent(), PersistenceManager.detachCopy and much more besides.

To read more about named fetch-groups and how to use it with attach/detach you can look at our Tutorial on DAO Layer design.

93.1.3 Dynamic Fetch Groups

The mechanism above provides static fetch groups defined in XML or annotations. That is great when you know in advance what fields you want to fetch. In some situations you may want to define your fields to fetch at run time. This became standard in JDO2.2 (was previously a DataNucleus extension). It operates as follows

```java
import org.datanucleus.FetchGroup;

// Create a FetchGroup on the PMF called "TestGroup" for MyClass
FetchGroup grp = myPMF.getFetchGroup(MyClass.class, "TestGroup");
grp.addMember("field1").addMember("field2");

// Add this group to the fetch plan (using its name)
fp.addGroup("TestGroup");
```

So we use the DataNucleus PMF as a way of creating a FetchGroup, and then register that FetchGroup with the PMF for use by all PMs. We then enable our FetchGroup for use in the FetchPlan by using its group name (as we do for a static group). The FetchGroup allows you to add/remove the fields necessary so you have full API control over the fields to be fetched.

93.1.4 Fetch Depth

The basic fetch group defines which fields are to be fetched. It doesn't explicitly define how far down an object graph is to be fetched. JDO2 provides two ways of controlling this.

The first is to set the maxFetchDepth for the FetchPlan. This value specifies how far out from the root object the related objects will be fetched. A positive value means that this number of relationships will be traversed from the root object. A value of -1 means that no limit will be placed on the fetching traversal. The default is 1. Let's take an example
public class MyClass1
{
    MyClass2 field1;
    ...
}

public class MyClass2
{
    MyClass3 field2;
    ...
}

public class MyClass3
{
    MyClass4 field3;
    ...
}

and we want to detach field1 of instances of MyClass1, down 2 levels - so detaching the initial “field1” MyClass2 object, and its "field2" MyClass3 instance. So we define our fetch-groups like this

    <class name="MyClass1">
        ...
        <fetch-group name="includingField1">
            <field name="field1"/>
        </fetch-group>
    </class>

    <class name="MyClass2">
        ...
        <fetch-group name="includingField2">
            <field name="field2"/>
        </fetch-group>
    </class>

and we then define the maxFetchDepth as 2, like this

    pm.getFetchPlan().setMaxFetchDepth(2);

A further refinement to this global fetch depth setting is to control the fetching of recursive fields. This is performed via a MetaData setting "recursion-depth". A value of 1 means that only 1 level of objects will be fetched. A value of -1 means there is no limit on the amount of recursion. The default is 1. Let’s take an example

    public class Directory
    {
        Collection children;
        ...
    }
So when we fetch a Directory, it will fetch 2 levels of the children field, hence fetching the children and the grandchildren.

93.1.5 Fetch Size

A FetchPlan can also be used for defining the fetching policy when using queries. This can be set using

```java
pm.getFetchPlan().setFetchSize(value);
```

The default is `FetchPlan.FETCH_SIZE_OPTIMAL` which leaves it to DataNucleus to optimise the fetching of instances. A positive value defines the number of instances to be fetched. Using `FetchPlan.FETCH_SIZE_GREEDY` means that all instances will be fetched immediately.
94 Query API

94.1 JDO : Query API

Once you have persisted objects you need to query them. For example if you have a web application representing an online store, the user asks to see all products of a particular type, ordered by the price. This requires you to query the datastore for these products. JDO allows support for several query languages using its API. DataNucleus provides querying using

- an object-oriented query language (JDOQL)
- a relational query language (SQL) for RDBMS datastores
- the pseudo-OO query language for JPA (JPQL)
- Stored Procedures for RDBMS datastores

Note that for some datastores additional query languages may be available specific to that datastore - please check the datastores documentation. The query language you choose is your choice, typically dependent on the skillset of the developers of your application.

We recommend using JDOQL for queries wherever possible since it is object-based and datastore agnostic, giving you extra flexibility in the future. If not possible using JDOQL, only then use a language appropriate to the datastore in question.

94.1.1 Creating a query

The principal ways of creating a query are

- Specifying the query language, and using a single-string form of the query

```java
Query q = pm.newQuery("javax.jdo.query.JDOQL", "SELECT FROM mydomain.MyClass WHERE field2 < threshold " + "PARAMETERS java.util.Date threshold");
```

or alternatively

```java
Query q = pm.newQuery("SQL", "SELECT * FROM MYTABLE WHERE COL1 == 25");
```

- A "named" query, (pre-)defined in metadata (refer to metadata docs).

```java
Query q = pm.newNamedQuery(MyClass.class, "MyQuery1");
```

- JDOQL : Use the single-string form of the query

```java
Query q = pm.newQuery("SELECT FROM mydomain.MyClass WHERE field2 < threshold " + "PARAMETERS java.util.Date threshold");
```

- JDOQL : Use the declarative API to define the query

```java
Query q = pm.newQuery(MyClass.class);
q.setFilter("field2 < threshold");
q.declareParameters("java.util.Date threshold");
```

- JDOQL : Use the typesafe API to define the query

```java
Query q = pm.newQuery(MyClass.class);
```
TypesafeQuery<MyClass> q = pm.newTypesafeQuery(MyClass.class);
QMyClass cand = QMyClass.candidate();
List<Product> results =
    q.filter(cand.field2.lt(q.doubleParameter("threshold"))).executeList();

Please note that with the query API you can also specify execution time information for the query, such as whether it executes in memory, or whether to apply a datastore timeout etc.

94.1.2 Compiling a query
An intermediate step once you have your query defined, if you want to check its validity is to compile it. You do this as follows

    q.compile();

If the query is invalid, then a JDO exception will be thrown.

94.1.3 Executing a query
So we have set up our query. We now execute it

    Object result = q.execute();

If we have parameters to pass in we can also do any of

    Object result = q.execute(paramVal1);

    Object result = q.execute(paramVal1, paramVal2);

    Object result = q.executeWithArray(new Object[]{paramVal1, paramVal2});

    Map paramMap = new HashMap();
    paramMap("param1", paramVal1);
    paramMap("param2", paramVal2);
    Object result = q.executeWithMap(paramMap);

By default, when a query is executed, it will execute in the datastore with what is present in the datastore at that time. If there are outstanding changes waiting to be flushed then these will not feature in the results. To flush these changes before execution, set the following query "extension" before calling execute

    q.addExtension("datanucleus.query.flushBeforeExecution","true");
94.1.4 Controlling the execution: Vendor extensions

JDO's query API allows implementations to support extensions and provides a simple interface for enabling the use of such extensions on queries.

```java
q.addExtension("extension_name", "value");
```

```java
HashMap exts = new HashMap();
exts.put("extension1", value1);
exts.put("extension2", value2);
q.setExtensions(exts);
```

94.1.5 Named Query

With the JDO API you can either define a query at runtime, or define it in the MetaData/annotations for a class and refer to it at runtime using a symbolic name. This second option means that the method of invoking the query at runtime is much simplified. To demonstrate the process, let's say we have a class called `Product` (something to sell in a store). We define the JDO Meta-Data for the class in the normal way, but we also have some query that we know we will require, so we define the following in the Meta-Data.

```xml
<jdo>
  <package name="mydomain">
    <class name="Product">
      ...
      <query name="SoldOut" language="javax.jdo.query.JDOQL">
        SELECT FROM mydomain.Product WHERE status == "Sold Out"
      </query>
    </class>
  </package>
</jdo>
```

So we have a JDOQL query called "SoldOut" defined for the class `Product` that returns all Products (and subclasses) that have a `status` of "Sold Out". Out of interest, what we would then do in our application to execute this query would be

```java
Query q = pm.newNamedQuery(mydomain.Product.class, "SoldOut");
Collection results = (Collection)q.execute();
```

The above example was for the JDOQL object-based query language. We can do a similar thing using SQL, so we define the following in our MetaData for our `Product` class
So here we have an SQL query that will return the names of all Products that have a price less than a specified value. This leaves us the flexibility to specify the value at runtime. So here we run our named query, asking for the names of all Products with price below 20 euros.

```java
Query q = pm.newNamedQuery(mydomain.Product.class, "PriceBelowValue");
Collection results = (Collection) q.execute(20.0);
```

All of the examples above have been specified within the `<class>` element of the MetaData. You can, however, specify queries below `<jdo>` in which case the query is not scoped by a particular candidate class. In this case you must put your queries in any of the following MetaData files:

```
/META-INF/package.jdo
/WEB-INF/package.jdo
/package.jdo
/META-INF/package-{mapping}.orm
/WEB-INF/package-{mapping}.orm
/package-{mapping}.orm
/META-INF/package.jdoquery
/WEB-INF/package.jdoquery
/package.jdoquery
```

### 94.1.6 Controlling the execution: FetchPlan

When a Query is executed it executes in the datastore, which returns a set of results. DataNucleus could clearly read all results from this ResultSet in one go and return them all to the user, or could allow control over this fetching process. JDO provides a fetch size on the Fetch Plan to allow this control. You would set this as follows:

```java
Query q = pm.newQuery(...);
q.getFetchPlan().setFetchSize(FetchPlan.FETCH_SIZE_OPTIMAL);
```

fetch size has 3 possible values.

- **FETCH_SIZE_OPTIMAL** - allows DataNucleus full control over the fetching. In this case DataNucleus will fetch each object when they are requested, and then when the owning transaction is committed will retrieve all remaining rows (so that the Query is still usable after the close of the transaction).
• **FETCH_SIZE_GREEDY** - DataNucleus will read all objects in at query execution. This can be efficient for queries with few results, and very inefficient for queries returning large result sets.

• **A positive value** - DataNucleus will read this number of objects at query execution. Thereafter it will read the objects when requested.

In addition to the number of objects fetched, you can also control which fields are fetched for each object of the candidate type. This is controlled via the FetchPlan. For RDBMS any single-valued member will be fetched in the original SQL query, but with multiple-valued members this is not supported. However what will happen is that any collection field will be retrieved in a single SQL query for all candidate objects; this avoids the "N+1" problem, resulting in 1 original SQL query plus 1 SQL query per collection member. Note that you can disable this by either not putting multi-valued fields in the FetchPlan, or by setting the query extension "datanucleus.multivaluedFetch" to "none" (default is "bulk-fetch" using the single SQL per field). For non-RDBMS datastores the collection/map is stored by way of a Collection of ids of the related objects in a single "column" of the object and so is retrievable in the same query. See also Fetch Groups.

---

**Extension**

DataNucleus also allows an extension to give further control. As mentioned above, when the transaction containing the Query is committed, all remaining results are read so that they can then be accessed later (meaning that the query is still usable). Where you have a large result set and you don't want this behaviour you can turn it off by specifying a Query extension

```java
q.addExtension("datanucleus.query.loadResultsAtCommit", "false");
```

so when the transaction is committed, no more results will be available from the query.

---

**Extension**

In some situations you don't want all FetchPlan fields retrieving, and DataNucleus provides an extension to turn this off, like this

```java
q.addExtension("datanucleus.query.useFetchPlan", "false");
```

---

**94.1.7 Control over locking of fetched objects**

JDO allows control over whether objects found by a query are locked during that transaction so that other transactions can't update them in the meantime. To do this you would do

```java
Query q = pm.newQuery(...);
q.setSerializeRead(true);
```

You can also specify this for all queries for all PMs using a PMF property `datanucleus.SerializeRead`. In addition you can perform this on a per-transaction basis by doing

```java
tx.setSerializeRead(true);
```

*If the datastore in use doesn’t support locking of objects then this will do nothing*
94.1.8 Flush changes before execution

When using optimistic transactions all updates to data are held until flush()/commit(). This means that executing a query may not take into account changes made during that transaction in some objects. DataNucleus allows a convenience of calling flush() just before execution of queries so that all updates are taken into account. The property name is `datanucleus.query.flushBeforeExecution` and defaults to "false".

To do this on a per query basis for JDO you would do

```java
query.addExtension("datanucleus.query.flushBeforeExecution","true");
```

You can also specify this for all queries using a persistence property `datanucleus.query.flushBeforeExecution` which would then apply to ALL queries for that PMF.

94.1.9 Controlling the execution : timeout on datastore reads

```java
q.setDatastoreReadTimeout(1000);
```

*Sets the timeout for this query (in milliseconds).* Will throw a JDOUnsupportedOperationException if the query implementation doesn't support timeouts.

94.1.10 Controlling the execution : timeout on datastore writes

```java
q.setDatastoreWriteTimeout(1000);
```

*Sets the timeout for this query (in milliseconds) when it is a delete/update.* Will throw a JDOUnsupportedOperationException if the query implementation doesn't support timeouts.
95 Query Cache

95.1 JDO : Query Caching

JDO doesn't currently define a mechanism for caching of queries. DataNucleus provides 3 levels of caching:

- **Generic Compilation**: when a query is compiled it is initially compiled *generically* into expression trees. This generic compilation is independent of the datastore in use, so can be used for other datastores. This can be cached.
- **Datastore Compilation**: after a query is compiled into expression trees (above) it is then converted into the native language of the datastore in use. For example with RDBMS, it is converted into SQL. This can be cached.
- **Results**: when a query is run and returns objects of the candidate type, you can cache the identities of the result objects.

### 95.1.1 Generic Query Compilation Cache

This cache is by default set to *weak*, meaning that the generic query compilation is cached using weak references. This is set using the persistence property `datanucleus.cache.queryCompilation.type`. You can also set it to *strong* meaning that strong references are used, or *soft* meaning that soft references are used, or finally to *none* meaning that there is no caching of generic query compilation information.

You can turn caching on/off (default = on) on a query-by-query basis by specifying the query extension `datanucleus.query.compilation.cached` as true/false.

```java
query.addExtension("datanucleus.query.compilation.cached", "true");
```

### 95.1.2 Datastore Query Compilation Cache

This cache is by default set to *weak*, meaning that the datastore query compilation is cached using weak references. This is set using the persistence property `datanucleus.cache.queryCompilationDatastore.type`. You can also set it to *strong* meaning that strong references are used, or *soft* meaning that soft references are used, or finally to *none* meaning that there is no caching of datastore-specific query compilation information.

You can turn caching on/off (default = on) on a query-by-query basis by specifying the query extension `datanucleus.query.compilation.cached` as true/false.
query.addExtension("datanucleus.query.compilation.cached", "true");

95.1.3 Query Results Cache

This cache is by default set to weak, meaning that the datastore query results are cached using weak references. This is set using the persistence property datanucleus.cache.queryResult.type. You can also set it to strong meaning that strong references are used, or soft meaning that soft references are used, or finally to none meaning that there is no caching of query results information. You can also specify datanucleus.cache.queryResult.cacheName to define the name of the cache used for the query results cache.

You can turn caching on/off (default = off) on a query-by-query basis by specifying the query extension datanucleus.query.results.cached as true/false. As a finer degree of control, where cached results are used, you can omit the validation of object existence in the datastore by setting the query extension datanucleus.query.resultCache.validateObjects.

query.addExtension("datanucleus.query.results.cached", "true");
query.addExtension("datanucleus.query.resultCache.validateObjects", "false");

Obviously with a cache of query results, you don't necessarily want to retain this cached over a long period. In this situation you can evict results from the cache like this.

import org.datanucleus.api.jdo.JDOQueryCache;
import org.datanucleus.api.jdo.JDOPersistenceManagerFactory;
...
JDOQueryCache cache = ((JDOPersistenceManagerFactory)pmf).getQueryCache();

  cache.evict(query);

which evicts the results of the specific query. The JDOQueryCache has more options available should you need them ...

javadoc
96.1 JDO : JDOQL Queries

JDO defines ways of querying objects persisted into the datastore. It provides its own object-based query language (JDOQL). JDOQL is designed as the Java developers way of having the power of SQL queries, yet retaining the Java object relationship that exist in their application model. A typical JDOQL query may be created in several ways. Here's an example expressed in the 3 supported ways

```
Single-String JDOQL :
Query q = pm.newQuery("SELECT FROM mydomain.Person WHERE lastName == 'Jones' && age < age_limit PARAMETERS int age_limit");
List<Person> results = (List<Person>)q.execute(20);

Declarative JDOQL :
Query q = pm.newQuery(Person.class);
qu.setFilter("lastName == 'Jones' && age < age_limit");
qu.declareParameters("int age_limit");
List<Person> results = (List<Person>)q.execute(20);

Typesafe JDOQL (DataNucleus) :
TypesafeQuery<Person> tq = pm.newTypesafeQuery(Person.class);
QPerson cand = QPerson.candidate();
List<Person> results =
tq.filter(cand.lastName.eq("Jones").and(cand.age.lt(tq.intParameter("age_limit"))))
.setParameter("age_limit", "20").executeList();
```

So here in our example we select all "Person" objects with surname of "Jones" and where the persons age is below 20. The language is intuitive for Java developers, and is intended as their interface to accessing the persisted data model. As can be seen above, the query is made up of distinct parts. The class being selected (the SELECT clause in SQL), the filter (which equates to the WHERE clause in SQL), together with any sorting (the ORDER BY clause in SQL), etc.

**In this section we will express all examples using the single-string format** since it is the simplest to highlight how to use JDOQL, so please refer to the Declarative JDOQL and Typesafe JDOQL guides for details if wanting to use those.

96.1.1 JDOQL Single-String syntax

JDOQL queries can be defined in a single-string form, as follows
SELECT [UNIQUE] [<result>] [INTO <result-class>]
  [FROM <candidate-class> [EXCLUDE SUBCLASSES]]
  [WHERE <filter>]
  [VARIABLES <variable declarations>]
  [PARAMETERS <parameter declarations>]
  [<import declarations>]
  [GROUP BY <grouping>]
  [ORDER BY <ordering>]
  [RANGE <start>, <end>]

The "keywords" in the query are shown in UPPER CASE but can be in UPPER or lower case (but not MiXeD case). So giving an example

SELECT UNIQUE FROM mydomain.Employee ORDER BY departmentNumber

96.1.2 Candidate Class

By default the candidate "class" with JDOQL has to be a persistable class. This can then be referred to in the query using the this keyword (just like in Java). Also by default your query will return instances of subclasses of the candidate class. You can restrict to just instances of the candidate by specifying to exclude subclasses (see EXCLUDE SUBCLASSES in the single-string syntax, or by setSubclasses(false) when using the declarative API).

DataNucleus also allows you to specify a candidate class as persistent interface. This is used where we want to query for instances of implementations of the interface. Let's take an example. We have an interface

```java
@PersistenceCapable
public interface ComputerPeripheral
{
  @PrimaryKey
  long getId();
  void setId(long val);

  @Persistent
  String getManufacturer();
  void setManufacturer(String name);

  @Persistent
  String getModel();
  void setModel(String name);
}
```

and we have the following implementations
@PersistenceCapable
class Mouse implements ComputerPeripheral {
    ...
}

@PersistenceCapable
class Keyboard implements ComputerPeripheral {
    ...
}

So we have made our interface persistable, and defined the identity property(ies) there. The implementations of the interface will use the identity defined in the interface. To query it we simply do

```java
Query q = pm.newQuery("SELECT FROM \{ComputerPeripheral.class\} where ...");
List<ComputerPeripheral> results = (List<ComputerPeripheral>)q.execute();
```

The key rules are

- You must define the interface as persistent
- The interface must define the identity/primary key member(s)
- The implementations must have the same definition of identity and primary key

### 96.1.3 Filter

The most important thing to remember when defining the filter for JDOQL is that think how you would write it in Java, and it's likely the same. The filter has to be a boolean expression, and can include the candidate, fields/properties, literals, methods, parameters, variables, operators, instanceof, subqueries and casts.

### 96.1.4 Fields/Properties

In JDOQL you refer to fields/properties in the query by referring to the field/bean name. For example, if you are querying a candidate class called `Product` and it has a field "price", then you access it like this

```java
price < 150.0
```

Note that, just like in Java, if you want to refer to a field/property of the candidate you can prefix the field by `this`

```java
this.price < 150.0
```

You can also chain field references if you have a candidate class `Product` with a field of (persistable) type `Inventory`, which has a field `name`, so you could do

```java
this.inventory.name == 'Backup'
```
In addition to the persistent fields, you can also access "public static final" fields of any class. You can do this as follows:

```java
taxPercent < mydomain.Product.TAX_BAND_A```

So this will find all products that include a tax percentage less than some "BAND A" level. Where you are using "public static final" fields you can either fully-qualify the class name or you can include it in the "imports" section of the query (see later).

### 96.1.5 Methods

When writing the "filter" for a JDOQL Query you can make use of some methods on the various Java types. The range of methods included as standard in JDOQL is not as flexible as with the true Java types, but the ones that are available are typically of much use. While DataNucleus supports all of the methods in the JDO standard, it also supports several yet to be standardised (extension) method. The tables below also mark whether a particular method is supported for evaluation in-memory.

Please note that you can easily add support for other methods for evaluation "in-memory" using this [DataNucleus plugin point](#)

Please note that you can easily add support for other methods with RDBMS datastore using this [DataNucleus plugin point](#)

#### 96.1.5.1 String Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Standard</th>
<th>In-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>startsWith(String)</code></td>
<td>Returns if the string starts with the passed string</td>
<td><img src="image" alt="Standard" /></td>
<td><img src="image" alt="In-Memory" /></td>
</tr>
<tr>
<td><code>startsWith(String, int)</code></td>
<td>Returns if the string starts with the passed string, from the passed position</td>
<td><img src="image" alt="Standard" /></td>
<td><img src="image" alt="In-Memory" /></td>
</tr>
<tr>
<td><code>endsWith(String)</code></td>
<td>Returns if the string ends with the passed string</td>
<td><img src="image" alt="Standard" /></td>
<td><img src="image" alt="In-Memory" /></td>
</tr>
<tr>
<td><code>indexOf(String)</code></td>
<td>Returns the first position of the passed string</td>
<td><img src="image" alt="Standard" /></td>
<td><img src="image" alt="In-Memory" /></td>
</tr>
<tr>
<td><code>indexOf(String,int)</code></td>
<td>Returns the position of the passed string, after the passed position</td>
<td><img src="image" alt="Standard" /></td>
<td><img src="image" alt="In-Memory" /></td>
</tr>
<tr>
<td><code>substring(int)</code></td>
<td>Returns the substring starting from the passed position</td>
<td><img src="image" alt="Standard" /></td>
<td><img src="image" alt="In-Memory" /></td>
</tr>
<tr>
<td><code>substring(int,int)</code></td>
<td>Returns the substring between the passed positions</td>
<td><img src="image" alt="Standard" /></td>
<td><img src="image" alt="In-Memory" /></td>
</tr>
</tbody>
</table>
Here's an example using a Product class, looking for objects which their abbreviation is the beginning of a trade name. The trade name is provided as parameter.

**Declarative JDOQL:**

```java
Query query = pm.newQuery(mydomain.Product.class);
query.setFilter(" :tradeName.startsWith(this.abbreviation)");
List results = (List)query.execute("Workbook Advanced");
```

**Single-String JDOQL:**

```java
Query query = pm.newQuery(
    "SELECT FROM mydomain.Product " +
    "WHERE :tradeName.startsWith(this.abbreviation)"");
List results = (List)query.execute("Workbook Advanced");
```
96.1.5.2 Collection Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Standard</th>
<th>In-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty()</td>
<td>Returns whether the collection is empty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>contains(value)</td>
<td>Returns whether the collection contains the passed element</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size()</td>
<td>Returns the number of elements in the collection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get(int)</td>
<td>Returns the element at that position of the List</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here's an example demonstrating use of contains(). We have an Inventory class that has a Collection of Product objects, and we want to find the Inventory objects with 2 particular Products in it. Here we make use of a variable (prd to represent the Product being contained

Declarative JDOQL:
```
Query query = pm.newQuery(mydomain.Inventory.class);
query.setFilter("products.contains(prd) && (prd.name=='product 1' || prd.name=='product 2')");
List results = (List)query.execute();
```

Single-String JDOQL:
```
Query query = pm.newQuery("SELECT FROM mydomain.Inventory WHERE products.contains(prd) && (prd.name=='product 1' || prd.name=='product 2')");
List results = (List)query.execute();
```

96.1.5.3 Map Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Standard</th>
<th>In-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty()</td>
<td>Returns whether the map is empty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>containsKey(key)</td>
<td>Returns whether the map contains the passed key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>containsValue(value)</td>
<td>Returns whether the map contains the passed value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get(key)</td>
<td>Returns the value from the map with the passed key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size()</td>
<td>Returns the number of entries in the map</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
containsEntry(key, value)  Returns whether the map contains the passed entry

Here’s an example using a Product class as a value in a Map. Our example represents an organisation that has several Inventories of products. Each Inventory of products is stored using a Map, keyed by the Product name. The query searches for all Inventories that contain a product with the name "product 1".

Declarative JDOQL:
Query query = pm.newQuery(mydomain.Inventory.class, "products.containsKey('product 1')");
List results = (List)query.execute();

Single-String JDOQL:
Query query = pm.newQuery("SELECT FROM mydomain.Inventory WHERE products.containsKey('product 1')");
List results = (List)query.execute();

Here’s the source code for reference

```java
class Inventory {
    Map<String, Product> products;
    ...
}
class Product {
    ...
}
```

96.1.5.4 Temporal Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Standard</th>
<th>In-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDate()</td>
<td>Returns the day (of the month) for the date (java.util.Date types)</td>
<td><img src="image" alt="Standard" /></td>
<td><img src="image" alt="In-Memory" /></td>
</tr>
<tr>
<td>getMonth()</td>
<td>Returns the month for the date (java.util.Date types)</td>
<td><img src="image" alt="Standard" /></td>
<td><img src="image" alt="In-Memory" /></td>
</tr>
<tr>
<td>getYear()</td>
<td>Returns the year for the date (java.util.Date types)</td>
<td><img src="image" alt="Standard" /></td>
<td><img src="image" alt="In-Memory" /></td>
</tr>
<tr>
<td>getHour()</td>
<td>Returns the hour for the time (java.util.Date types)</td>
<td><img src="image" alt="Standard" /></td>
<td><img src="image" alt="In-Memory" /></td>
</tr>
<tr>
<td>getMinute()</td>
<td>Returns the minute for the time (java.util.Date types)</td>
<td><img src="image" alt="Standard" /></td>
<td><img src="image" alt="In-Memory" /></td>
</tr>
</tbody>
</table>
getSecond() Returns the second for the time (java.util.Date types)

96.1.5.5 Enum Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Standard</th>
<th>In-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordinal()</td>
<td>Returns the ordinal of the enum (not implemented for enum expression when persisted as a string)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>toString()</td>
<td>Returns the string form of the enum (not implemented for enum expression when persisted as a numeric)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

96.1.5.6 Other Methods

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Description</th>
<th>Standard</th>
<th>In-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.awt.Point</td>
<td>getX()</td>
<td>Returns the X coordinate. Only on RDBMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.awt.Point</td>
<td>getY()</td>
<td>Returns the Y coordinate. Only on RDBMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.awt.Rectangle</td>
<td>getX()</td>
<td>Returns the X coordinate. Only on RDBMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.awt.Rectangle</td>
<td>getY()</td>
<td>Returns the Y coordinate. Only on RDBMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.awt.Rectangle</td>
<td>getWidth()</td>
<td>Returns the width. Only on RDBMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.awt.Rectangle</td>
<td>getHeight()</td>
<td>Returns the height. Only on RDBMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.joda.time.Interval</td>
<td>getStart()</td>
<td>Returns the start date/time object. Only on RDBMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.joda.time.Interval</td>
<td>getEnd()</td>
<td>Returns the end date/time object. Only on RDBMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[]</td>
<td>length</td>
<td>Returns the length of an array. Only on RDBMS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
96.1.5.7 Static Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Standard</th>
<th>In-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.abs(number)</td>
<td>Returns the absolute value of the passed number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Math.sqrt(number)</td>
<td>Returns the square root of the passed number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Math.cos(number)</td>
<td>Returns the cosine of the passed number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Math.sin(number)</td>
<td>Returns the absolute value of the passed number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Math.tan(number)</td>
<td>Returns the tangent of the passed number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Math.acos(number)</td>
<td>Returns the arc cosine of the passed number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Math.asin(number)</td>
<td>Returns the arc sine of the passed number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Math.atan(number)</td>
<td>Returns the arc tangent of the passed number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Math.ceil(number)</td>
<td>Returns the ceiling of the passed number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Math.exp(number)</td>
<td>Returns the exponent of the passed number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Math.floor(number)</td>
<td>Returns the floor of the passed number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Math.log(number)</td>
<td>Returns the log(base e) of the passed number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JDOHelper.getObjectId(object)</td>
<td>Returns the object identity of the passed persistent object</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JDOHelper.getVersion(object)</td>
<td>Returns the version of the passed persistent object</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_rollup(object)</td>
<td>Perform a rollup operation over the results. Only for some RDBMS e.g DB2, MSSQL, Oracle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_cube(object)</td>
<td>Perform a cube operation over the results. Only for some RDBMS e.g DB2, MSSQL, Oracle</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
96.1.6 Literals

JDOQL supports the following literals: IntegerLiteral, FloatingPointLiteral, BooleanLiteral, CharacterLiteral, StringLiteral, and NullLiteral. When String literals are specified using single-string format they should be surrounded by single-quotes ‘.’

96.1.7 Parameters

With a query you can pass values into the query as parameters. This is useful where you don’t want to embed particular values in the query itself, so making it reusable with different values. JDOQL allows two types of parameters.

96.1.7.1 Explicit Parameters

If you declare the parameters when defining the query (using the PARAMETERS keyword in the single-string form, or via the declareParameters method) then these are explicit parameters. This sets the type of the parameter, and when you pass the value in at execute it has to be of that type. For example

```java
Query query = pm.newQuery(
    "SELECT FROM mydomain.Product WHERE price < limit PARAMETERS double limit");
List results = (List)query.execute(150.00);
```

Note that if declaring multiple parameters then they should be comma-separated.

96.1.7.2 Implicit Parameters

If you don’t declare the parameters when defining the query but instead prefix identifiers in the query with : (colon) then these are implicit parameters. For example

```java
Query query = pm.newQuery(
    "SELECT FROM mydomain.Product WHERE price < :limit");
List results = (List)query.execute(150.00);
```

In some situations you may have a map of parameters keyed by their name, yet the query in question doesn’t need all parameters. Normal JDO execution would throw an exception here since they are inconsistent with the query. You can omit this check by setting

```java
q.addExtension("datanucleus.query.ignoreParameterCountCheck", "true");
```
96.1.8 Variables

In JDOQL you can connect two parts of a query using something known as a variable. For example, we want to retrieve all objects with a collection that contains a particular element, and where the element has a particular field value. We define a query like this:

```
Query query = pm.newQuery("SELECT FROM mydomain.Supplier " +
   "WHERE products.contains(prod) && prod.name == 'Beans' VARIABLES mydomain.Product prod");
```

So we have a variable in our query called `prod` that connects the two parts. You can declare your variables (using the `VARIABLES` keyword in the single-string form, or via the `declareVariables` method) if you want to define the type like here (explicit), or you can leave them for the query compilation to determine (implicit).

Note that if declaring multiple variables then they should be semicolon-separated.

96.1.9 Imports

JDOQL uses the imports declaration to create a type namespace for the query. During query compilation, the classes used in the query, if not fully qualified, are searched in this namespace. The type namespace is built with primitives types, `java.lang.*` package, package of the candidate class, import declarations (if any).

To resolve a class, the JDOQL compiler will use the class fully qualified name to load it, but if the class is not fully qualified, it will search by prefixing the class name with the imported package names declared in the type namespace. All classes loaded by the query must be accessible by either the candidate class classloader, the PersistenceManager classloader or the current Thread classloader. The search algorithm for a class in the JDOQL compiler is the following:

- if the class is fully qualified, load the class.
- if the class is not fully qualified, iterate each package in the type namespace and try to load the class from that package. This is done until the class is loaded, or the type namespace package names are exhausted. If the class cannot be loaded an exception is thrown.

Note that the search algorithm can be problematic in performance terms if the class is not fully qualified or declared in imports using package notation. To avoid such problems, either use fully qualified class names or import the class in the imports declaration.

96.1.10 Operators

The following list describes the operator precedence in JDOQL.

1. Cast
2. Unary ("~") ("!"")
3. Unary ("+"") ("-")
4. Multiplicative ("*") ("/") ("%")
5. Additive ("+") ("-")
6. Relational (">=") (">") ("<=") ("<") ("instanceof")
7. Equality ("==") ("!=")
8. Boolean logical AND ("&")
9. Boolean logical OR ("|")
10. Conditional AND ("&&")
11 Conditional OR ("||")
The concatenation operator (+) concatenates a String to either another String or Number. Concatenations of String or Numbers to null results in null.

96.1.11 instanceof
JDOQL allows the Java keyword instanceof so you can compare objects against a class.
Let's take an example. We have a class A that has a field "b" of type B and B has subclasses B1, B2, B3. Clearly the field "b" of A can be of type B, B1, B2, B3 etc, and we want to find all objects of type A that have the field "b" that is of type B2. We do it like this

Declarative JDOQL:
Query query = pm.newQuery(A.class);
query.setFilter("b instanceof mydomain.B2");
List results = (List)query.execute();

Single-String JDOQL:
Query query = pm.newQuery("SELECT FROM mydomain.A WHERE b instanceof mydomain.B2");
List results = (List)query.execute();

96.1.12 casting
JDOQL allows use of Java-style casting so you can type-convert fields etc.
Let's take an example. We have a class A that has a field "b" of type B and B has subclasses B1, B2, B3. The B2 subtype has a field "other", and we know that the filtered A will have a B2. You could specify a filter using the "B2.other" field like this

((mydomain.B2)b).other == :someVal"

96.1.13 Subqueries
With JDOQL the user has a very flexible query syntax which allows for querying of the vast majority of data components in a single query. In some situations it is desirable for the query to utilise the results of a separate query in its calculations. JDO allows subqueries, so that both calculations can be performed in one query. Here's an example, using single-string JDOQL

SELECT FROM org.datanucleus.Employee WHERE salary >
(SELECT avg(salary) FROM org.datanucleus.Employee e)

So we want to find all Employees that have a salary greater than the average salary. In single-string JDOQL the subquery must be in parentheses (brackets). Note that we have defined the subquery with an alias of "e", whereas in the outer query the alias is "this".
We can specify the same query using the Declarative API, like this
So we define a subquery as its own Query (that could be executed just like any query if so desired), and the in the main query have an implicit variable that we define as being represented by the subquery.

96.1.13.1 Referring to the outer query in the subquery

JDOQL subqueries allows use of the outer query fields within the subquery if so desired. Taking the above example and extending it, here is how we do it in single-string JDOQL.

```
SELECT FROM org.datanucleus.Employee WHERE salary >
    (SELECT avg(salary) FROM org.datanucleus.Employee e WHERE e.lastName == this.lastName)
```

So with single-string JDOQL we make use of the alias identifier "this" to link back to the outer query.

Using the Declarative API, to achieve the same thing we would do

```
Query averageSalaryQuery = pm.newQuery(Employee.class);
averageSalaryQuery.setResult("avg(this.salary)");

Query q = pm.newQuery(Employee.class, "salary > averageSalary");
q.declareVariables("double averageSalary");
q.addSubquery(averageSalaryQuery, "double averageSalary", null, null);
List results = (List)q.execute();
```

So with the Declarative API we make use of parameters, and the last argument to `addSubquery` is the value of the parameter `lastNameParam`.

96.1.13.2 Candidate of the subquery being part of the outer query

There are occasions where we want the candidate of the subquery to be part of the outer query, so JDOQL subqueries has the notion of a candidate expression. This is an expression relative to the candidate of the outer query. An example

```
SELECT FROM org.datanucleus.Employee WHERE this.weeklyhours >
    (SELECT AVG(e.weeklyhours) FROM this.department.employees e)
```

so the candidate of the subquery is `this.department.employees`. If using a candidate expression we must provide an alias.

You can do the same with the Declarative API. Like this
so now our subquery has a candidate related to the outer query candidate.

96.1.14 Result clause

By default (when not specifying the result) the objects returned will be of the candidate class type, where they match the query filter. The result clause can contain (any of) the following:

- DISTINCT - optional keyword at the start of the results to make them distinct
- this - the candidate instance
- A field name
- A variable
- A parameter (though why you would want a parameter returning is hard to see since you input the value in the first place)
- An aggregate (count(), avg(), sum(), min(), max())
- An expression involving a field (e.g. "field1 + 1")
- A navigational expression (navigating from one field to another ... e.g. "field1.field4")

so you could specify something like

\[ \text{count(field1), field2} \]

There are situations when you want to return a single number for a column, representing an aggregate of the values of all records. There are 5 standard JDO aggregate functions available. These are:

- \text{avg(val)} - returns the average of "val". "val" can be a field, numeric field expression or "distinct field". Returns double.
- \text{sum(val)} - returns the sum of "val". "val" can be a field, numeric field expression, or "distinct field". Returns the same type as the type being summed
- \text{count(val)} - returns the count of records of "val". "val" can be a field, or can be "this", or "distinct field". Returns long
- \text{min(val)} - returns the minimum of "val". "val" can be a field. Returns the same type as the type used in "min"
- \text{max(val)} - returns the maximum of "val". "val" can be a field. Returns the same type as the type used in "max"

So to utilise these you could specify a result like

\[ \text{max(price), min(price)} \]

This will return a single row of results with 2 values, the maximum price and the minimum price. Note that what you specify in the result defines what form of result you get back when executing the query.

- \{\text{ResultClass}\} - this is returned if you have only a single row in the results and you specified a result class.
• **Object** - this is returned if you have only a single row in the results and a single column. This is achieved when you specified either UNIQUE, or just an aggregate (e.g. "max(field2)"
• **Object[]** - this is returned if you have only a single row in the results, but more than 1 column (e.g. "max(field1), avg(field2)"
• **List<\{ResultClass\}>** - this is returned if you specified a result class.
• **List<Object>** - this is returned if you have only a single column in the result, and you don’t have only aggregates in the result (e.g. "field2")
• **List<Object[]>** - this is returned if you have more than 1 column in the result, and you don’t have only aggregates in the result (e.g. "field2, avg(field3)"

### 96.1.15 Result Class

By default a JDOQL query will return a result matching the result clause. You can override this if you wish by specifying a result class. If your query has only a single row in the results then you will get an object of your result class back, otherwise you get a List of result class objects. The Result Class has to meet certain requirements. These are

• Can be one of Integer, Long, Short, Float, Double, Character, Byte, Boolean, String, java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp, or Object[]
• Can be a user defined class, that has either a constructor taking arguments of the same type as those returned by the query (in the same order), or has a public put(Object, Object) method, or public setXXX() methods, or public fields.

In terms of how the Result Class looks, you have two options

• Constructor taking arguments of the same types and the same order as the result clause. An instance of the result class is created using this constructor. For example

```java
public class Price
{
  protected double amount = 0.0;
  protected String currency = null;

  public Price(double amount, String currency)
  {
    this.amount = amount;
    this.currency = currency;
  }

  ...
}
```

• Default constructor, and setters for the different result columns, using the alias name for each column as the property name of the setter. For example

```java
public class Price
{
  protected double amount = 0.0;
  protected String currency = null;

  public Price() {
    this.currency = null;
  }

  public void setAmount(double amount) {
    this.amount = amount;
  }

  public void setCurrency(String currency) {
    this.currency = currency;
  }

  ...
}
```
public class Price
{
    protected double amount = 0.0;
    protected String currency = null;

    public Price()
    {
    }

    public void setAmount(double amt) {this.amount = amt;}
    public void setCurrency(String curr) {this.currency = curr;}

    ...
}

96.1.16 Grouping of Results
By default your results will have no specified "grouping". You can specify a grouping with optional having expression. When grouping is specified, each result expression must either be an expression contained in the grouping, or an aggregate evaluated once per group.

96.1.17 Ordering of Results
By default your results will be returned in the order determined by the datastore, so don't rely on any particular order. You can, of course, specify the order yourself. You do this using field/property names and ASC/DESC keywords. For example

field1 ASC, field2 DESC

which will sort primarily by field1 in ascending order, then secondarily by field2 in descending order.

Although it is not (yet) standard JDOQL, DataNucleus also supports specifying a directive for where NULL values of the ordered field/property go in the order, so the full syntax supported is

fieldName [ASC|DESC] [NULLS FIRST|NULLS LAST]

Note that this is only supported for a few RDBMS (H2, HSQLDB, PostgreSQL, DB2, Oracle, Derby, Firebird, SQLServer v11+).

96.1.18 Range of Results
By default your query will return all results matching the specified filter. You can select just a particular range of results by specifying the RANGE part of the query (or by using setRange when using the declarative API). For example

RANGE 10,20
which will return just the results numbers 10-19 inclusive. Obviously bear in mind that if specifying the range then you should really specify an ordering otherwise the range positions will be not defined.

96.2 JDOQL In-Memory queries

The typical use of a JDOQL query is to translate it into the native query language of the datastore and return objects matched by the query. Sometimes you want to query over a set of objects that you have to hand, or for some datastores it is simply impossible to support the full JDOQL syntax in the datastore native query language. In these situation we need to evaluate the query in-memory. In the latter case of the datastore not supported the full JDOQL syntax we evaluate as much as we can in the datastore and then instantiate those objects and evaluate further in-memory. Here we document the current capabilities of in-memory evaluation in DataNucleus.

To enable evaluation in memory you specify the query extension `datanucleus.query.evaluateInMemory` to `true` as follows

```
query.addExtension("datanucleus.query.evaluateInMemory","true");
```

This is also useful where you have a Collection of (persisted) objects and want to run a query over the Collection. Simply turn on in-memory evaluation, and supply the candidate collection to the query, and no communication with the datastore will be needed.

96.2.1 Specify candidates to query over

With JDO you can define a set of candidate objects that should be queried, rather than just going to the datastore to retrieve those objects. When you specify this you will automatically be switched to evaluate the query in-memory. You set the candidates like this

```
Query query = pm.newQuery(...);
query.setCandidates(myCandidates);
List<Product> results = (List<Product>)query.execute();
```

96.3 Update/Delete queries

JDOQL offers some possibilities for updating/deleting data in the datastore via query. Note that only the first of these is standard JDOQL, whereas the others are DataNucleus extensions.

96.3.1 Deletion by Query

If you want to delete instances of a candidate using a query, you simply define the query candidate/filter in the normal way, and then instead of calling `query.execute()` you call `query.deletePersistentAll()`. Like this
Query query = pm.newQuery("SELECT FROM mydomain.A WHERE this.value < 50");
Long number = (Long)query.deletePersistentAll();

The value returned is the number of instances that were deleted. Note that this will perform any
cascade deletes that are defined for these instances. In addition, all instances in memory will reflect
this deletion.

96.3.2 Bulk Delete

DataNucleus provides an extension to allow bulk deletion. This differs from the "Deletion by Query"
above in that it simply goes straight to the datastore and performs a bulk delete, leaving it to the
datastore referential integrity to handle relationships. To enable "bulk delete" you need the persistence property `datanucleus.query.jdoql.allowAll` set to `true`. You then perform "bulk delete" like this

Query query = pm.newQuery("DELETE FROM mydomain.A WHERE this.value < 50");
Long number = (Long)query.execute();

Again, the number returned is the number of records deleted.

96.3.3 Bulk Update

DataNucleus provides an extension to allow bulk update. This allows you to do bulk updates direct to
the datastore without having to load objects into memory etc. To enable "bulk update" you need the persistence property `datanucleus.query.jdoql.allowAll` set to `true`. You then perform "bulk update" like this

Query query = pm.newQuery("UPDATE mydomain.A SET this.value=this.value-5.0 WHERE this.value > 100");
Long number = (Long)query.execute();

Again, the number returned is the number of records updated.
97 JDOQL Declarative

97.1 JDO: JDOQL Declarative API

As shown earlier, there are two primary ways of defining a JDOQL query. In this guide we describe the declarative approach, defining the individual components of the query via an API. You can also refer to the API javadoc, We firstly need to look at a typical declarative JDOQL Query.

```java
Query query = pm.newQuery(MyClass.class);
query.setFilter("field2 < threshold");
query.declareImports("import java.util.Date");
query.declareParameters("Date threshold");
query.setOrdering("field1 ascending");
List results = (List) query.execute(my_threshold);
```

In this Query, we create it to return objects of type `mydomain.MyClass` (or subclasses), and set the filter to restrict to instances of that type which have the field `field2` less than some threshold value, which we don't know at that point. We've specified the query like this because we want to pass the threshold value in dynamically. We then import the type of our `threshold` parameter, and the parameter itself, and set the `ordering` of the results from the Query to be in ascending order of some field `field1`. The Query is then executed, passing in the threshold value. The example is to highlight the typical methods specified for a Query. Clearly you may only specify the Query line if you wanted something very simple. The result of the Query is cast to a List since in this case it returns a List of results.

97.1.1 setClass()

Set the class of the candidate instances of the query. The class specifies the class of the candidates of the query. Elements of the candidate collection that are of the specified class are filtered before being put into the results.

97.1.2 setUnique()

Specify that only the first result of the query should be returned, rather than a collection. The execute method will return null if the query result size is 0.

Sometimes you know that the query can only return 0 or 1 objects. In this case you can simplify your job by adding

```java
query.setUnique(true);
```

In this case the return from the execution of the Query will be a single Object, so you've no need to use iterators, just cast it to your candidate class type. Note that if you specify `unique` and there are more results than just 1 then it will throw a JDOUserException.
97.1.3 setResult()

Specifies what type of data this query should return. If this is unset or set to null, this query returns instances of the query's candidate class. If set, this query will return expressions, including field values (projections) and aggregate function results.

The normal behaviour of JDOQL queries is to return a List of Objects of the type of the candidate class. Sometimes you want to have the query perform some processing and return things like count(), min(), max() etc. You specify this with

```java
query.setResult("count(param1), max(param2), param3");
```

In this case the results will be List<Object[]> since there are more than 1 column in each row. If you have only 1 column in the results then the results would be List<Object>. If you have only aggregates (sum, avg, min, max, count) in the result clause then there will be only 1 row in the results and so the results will be of the form Object[] (or Object if only 1 aggregate). Please refer to JDOQL Result Clauses for more details.

97.1.4 setResultClass()

Specify the type of object in which to return each element of the result of invoking execute(). If the result is not set or set to null, the result class defaults to the candidate class of the query. If the result consists of one expression, the result class defaults to the type of that expression. If the result consists of more than one expression, the result class defaults to Object[].

When you perform a query, using JDOQL or SQL the query will, in general, return a List of objects. These objects are by default of the same type as the candidate class. This is good for the majority of situations but there are some situations where you would like to control the output object. This can be achieved by specifying the Result Class.

```java
query.setResultClass(myResultClass);
```

The Result Class has to meet certain requirements. These are

- Can be one of Integer, Long, Short, Float, Double, Character, Byte, Boolean, String, java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp, or Object[]
- Can be a user defined class, that has either a constructor taking arguments of the same type as those returned by the query (in the same order), or has a public put(Object, Object) method, or public setXXX() methods, or public fields.

Where you have a query returning a single field, you could specify the Result Class to be one of the first group for example. Where your query returns multiple fields then you can set the Result Class to be your own class. So we could have a query like this

```java
Query query = pm.newQuery(pm.getExtent(Payment.class, false));
query.setFilter("amount > 10.0");
query.setResultClass(Price.class);
query.setResult("amount, currency");
List results = (List)query.execute();
```

and we define our Result Class Price as follows
In this case our query is returning 2 fields (a Double and a String), and these map onto the constructor arguments, so DataNucleus will create objects of the `Price` class using that constructor. We could have provided a class with public fields instead, or provided `setXXX` methods or a `put` method. They all work in the same way.

97.1.5 `setRange()`

Set the range of results to return. The execution of the query is modified to return only a subset of results. If the filter would normally return 100 instances, and `fromIncl` is set to 50, and `toExcl` is set to 70, then the first 50 results that would have been returned are skipped, the next 20 results are returned and the remaining 30 results are ignored. An implementation should execute the query such that the range algorithm is done at the data store.

Sometimes you have a Query that returns a large number of objects. You may want to just display a range of these to your user. In this case you can do

```java
query.setRange(10, 20);
```

This has the effect of only returning items 10 through to 19 (inclusive) of the query's results. The clear use of this is where you have a web system and you're displaying paginated data, and so the user hits page down, so you get the next "n" results.

`setRange` is implemented efficiently for MySQL, Postgresql, HSQL (using the LIMIT SQL keyword) and Oracle (using the ROWNUM keyword), with the query only finding the objects required by the user directly in the datastore. For other RDBMS the query will retrieve all objects up to the "to" record, and will not pass any unnecessary objects that are before the "from" record.

97.1.6 `setFilter()`

Set the filter for the query. The filter specification is a String containing a Boolean expression that is to be evaluated for each of the instances in the candidate collection. If the filter is not specified, then it defaults to "true", which has the effect of filtering the input `Collection` only for class type.
97.1.7 declareImports()

Set the import statements to be used to identify the fully qualified name of variables or parameters. Parameters and unbound variables might come from a different class from the candidate class, and the names need to be declared in an import statement to eliminate ambiguity. Import statements are specified as a String with semicolon-separated statements.

In JDOQL you can declare parameters and variables. Just like in Java it is often convenient to just declare a variable as say Date, and then have an import in your Java file importing the java.util.Date class. The same applies in JDOQL. Where you have defined parameters or variables in shorthand form, you can specify their imports like this

```java
query.declareVariables("Date startDate");
query.declareParameters("Locale myLocale");
query.declareImports("import java.util.Locale; import java.util.Date;");
```

Just like in Java, if you declare your parameters or variables in fully-specified form (for example "java.util.Date myDate") then you do not need any import.

The JDOQL uses the imports declaration to create a type namespace for the query. During query compilation, the classes used in the query, if not fully qualified, are searched in this namespace. The type namespace is built with the following:

- primitives types
- java.lang.* package
- package of the candidate class
- import declarations (if any)

To resolve a class, the JDOQL compiler will use the class fully qualified name to load it, but if the class is not fully qualified, it will search by prefixing the class name with the imported package names declared in the type namespace. All classes loaded by the query must be accessible by either the candidate class classloader, the PersistenceManager classloader or the current Thread classloader. The search algorithm for a class in the JDOQL compiler is the following:

- if the class is fully qualified, load the class.
- if the class is not fully qualified, iterate each package in the type namespace and try to load the class from that package. This is done until the class is loaded, or the type namespace package names are exhausted. If the class cannot be loaded an exception is thrown.

Note that the search algorithm can be problematic in performance terms if the class is not fully qualified or declared in imports using package notation. To avoid such problems, either use fully qualified class names or import the class in the imports declaration. The 2 queries below are examples of good usage:

```java
query.declareImports("import java.util.Locale;");
query.declareParameters("Locale myLocale");
```

or

```java
query.declareParameters("java.util.Locale myLocale");
```

However, the below example will suffer in performance, due to the search algorithm.
97.1.8 declareParameters()

Declare the list of parameters query execution. The parameter declaration is a String containing one or more query parameter declarations separated with commas. Each parameter named in the parameter declaration must be bound to a value when the query is executed.

When using explicit parameters you need to declare them and their types. With the declarative API you do it like this

```java
query.declareImports("import java.math.*; import java.util.*;");
query.declareParameters("Locale myLocale");
```

So we make use of imports to define some package names (just like in Java). You can use * notation too. Note that java.lang is not needed to be imported. Alternatively you could have just done

```java
query.declareParameters("java.lang.String myparam1, java.util.Date myparam2");
```

97.1.9 declareVariables()

Declare the unbound variables to be used in the query. Variables might be used in the filter, and these variables must be declared with their type. The unbound variable declaration is a String containing one or more unbound variable declarations separated with semicolons.

With explicit variables, you declare your variables and their types. In declarative JDOQL it is like this

```java
query.declareVariables("mydomain.Product prod");
```

Multiple variables can be declared using semi-colon (;) to separate variable declarations.

```java
query.declareVariables("String var1; String var2");
```

97.1.10 setOrdering()

Set the ordering specification for the result Collection. The ordering specification is a String containing one or more ordering declarations separated by commas. Each ordering declaration is the name of the field on which to order the results followed by one of the following words: “ascending” or "descending". The field must be declared in the candidate class or must be a navigation expression starting with a field in the candidate class.

With JDOQL you can specify the ordering using the normal JDOQL syntax for a parameter, and then add ascending or descending (UPPER or lower case are both valid) are to give the direction. In
addition the abbreviated forms of \texttt{asc} and \texttt{desc} (again, UPPER and lower case forms are accepted) to save typing. For example, you may set the ordering as follows

\begin{verbatim}
query.setOrdering("productId DESC");
\end{verbatim}

97.1.11 \texttt{setGrouping()}

\textit{Set the grouping expressions, optionally including a "having" clause. When grouping is specified, each result expression must either be an expression contained in the grouping, or an aggregate evaluated once per group.}
98 JDOQL Typesafe

98.1 JDO : Typesafe JDOQL Queries

In JPA2 there is a query API referred to as "criteria". This is really an API allowing the construction of queries expression by expression, and optionally making it type safe so that if you refactor a field name then it is changed in the queries. JDO has no such feature currently, but there exist third party extensions providing this. One is called QueryDSL. DataNucleus now provides its own typesafe JDOQL query API, inspired by the ideas in QueryDSL (and some others), and is proposed for inclusion in JDO3.1.

With this API you can express your queries in a typesafe way and allow easier refactoring. This API produces queries that are much more elegant and simpler than the equivalent "Criteria" API in JPA, or the Hibernate Criteria API. See this comparison of JPA Criteria and JDO Typesafe.

98.1.1 Preparation

To set up your environment to use this typesafe query API you need to enable annotation processing (JDK1.6), place some DataNucleus jars in your build path, and specify an @PersistenceCapable annotation on your classes to be used in queries (you can still provide the remaining information in XML metadata if you wish to).

With Maven2 you need to have the following in your POM
<dependencies>
  <dependency>
    <groupId>org.datanucleus</groupId>
    <artifactId>datanucleus-core</artifactId>
    <version>[2.9, )</version>
  </dependency>
  <dependency>
    <groupId>org.datanucleus</groupId>
    <artifactId>datanucleus-api-jdo</artifactId>
    <version>[2.9, )</version>
  </dependency>
  <dependency>
    <groupId>org.datanucleus</groupId>
    <artifactId>datanucleus-jdo-query</artifactId>
    <version>[2.9, )</version>
  </dependency>
  <dependency>
    <groupId>javax.jdo</groupId>
    <artifactId>jdo-api</artifactId>
    <version>3.1-SNAPSHOT</version>
  </dependency>
  ...
</dependencies>

<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>1.6</source>
    <target>1.6</target>
  </configuration>
</plugin>

With Eclipse you need to

- Go to Java Compiler and make sure the compiler compliance level is 1.6 or above
- Go to Java Compiler -> Annotation Processing and enable the project specific settings and enable annotation processing
- Go to Java Compiler -> Annotation Processing -> Factory Path, enable the project specific settings and then add the following jars to the list: "datanucleus-jdo-query.jar", "datanucleus-api-jdo.jar", "datanucleus-core.jar", "jdo-api.jar"

98.1.2 Query Classes

The above preparation will mean that whenever you compile, the DataNucleus annotation processor will generate a query class for each model class that is annotated as persistable. So what is a query class you ask. It is simply a mechanism for providing an intuitive API to generating queries. If we have the following model class

```java
public class MyModel {
    @PersistenceCapable
    @Entity
}
```
the query class for this will be

```java
public class QProduct
    extends org.datanucleus.api.jdo.query.PersistableExpressionImpl<Product>
    implements PersistableExpression<Product>
{
    public static QProduct candidate(String name) {...}
    public static QProduct candidate() {...}
    public static QProduct variable(String name) {...}
    public static QProduct parameter(String name) {...}

    public NumericExpression<Long> id;
    public StringExpression name;
    public NumericExpression<Double> value;
    ...
}
```

Note that it has the name Q{className}. Also the generated class, by default, has a public field for each persistable field/property and is of a type XXXExpression. These expressions allow us to give Java like syntax when defining your queries (see below). So you access your persistable members in a query as candidate.name for example.

As mentioned above this is the default style of query class. However you can also create it in property style, where you access your persistable members as candidate.name() for example. The benefit of this approach is that if you have 1-1, N-1 relationship fields then it only initialises the members when called, whereas in the field case above it has to initialise all in the constructor, so at static initialisation. You enable use of property mode by adding the compiler argument -AqueryMode=PROPERTY. All examples below use field mode but just add () after the field to see the equivalent in property mode.

### 98.1.3 Query API - Filtering and Ordering

Let's provide a sample usage of this query API. We want to construct a query for all products with a value below a certain level, and where the name starts with "Wal", and then order the results by the product name. So a typical query in a JDO-enabled application
This equates to the single-string query

```
SELECT FROM mydomain.Product
    WHERE this.value < 40.0 && this.name.startsWith("Wal")
ORDER BY this.name ASCENDING
```

As you see, we create a parametrised query, and then make use of the `query class` to access the candidate, and from that make use of its fields, and the various Java methods present for the types of those fields. Also the API is fluent.

### 98.1.4 Query API - Results

Let's take the query in the above example and return the name and value of the Products only

```
TypesafeQuery<Product> tq = jdopm.newTypesafeQuery(Product.class);
QProduct cand = QProduct.candidate();
List<Object[]> results =
    tq.filter(cand.value.lt(40.00).and(cand.name.startsWith(tq.stringParameter("prefix"))))
        .orderBy(cand.name.asc())
        .setParameter("prefix", "Wal")
        .executeResultList(false, cand.name, cand.value);
```

This equates to the single-string query

```
SELECT this.name, this.value FROM mydomain.Product
    WHERE this.value < 40.0 && this.name.startsWith(:prefix)
ORDER BY this.name ASCENDING
```

A further example using aggregates

```
TypesafeQuery<Product> tq = jdopm.newTypesafeQuery(Product.class);
QProduct cand = QProduct.candidate();
Object[] results = tq.executeResultUnique(false, cand.max(), cand.min());
```

This equates to the single-string query

```
SELECT max(this.value), min(this.value) FROM mydomain.Product
```
98.1.5 Query API - Parameters

Let's take the query in the above example and specify the "Wal" in a parameter.

```java
TypesafeQuery<Product> tq = jdopm.newTypesafeQuery(Product.class);
QProduct cand = QProduct.candidate();
List<Product> results =
    tq.filter(cand.value.lt(40.00).and(cand.name.startsWith(tq.stringParameter("prefix"))))
    .orderBy(cand.name.asc())
    .setParameter("prefix", "Wal")
    .executeList();
```

This equates to the single-string query

```
SELECT FROM mydomain.Product
WHERE this.value < 40.0 && this.name.startsWith("Wal")
ORDER BY this.name ASCENDING
```

98.1.6 Query API - Variables

Let's try to find all Inventory objects containing a Product with a particular name. This means we need to use a variable.

```java
TypesafeQuery<Inventory> tq = jdopm.newTypesafeQuery(Inventory.class);
QProduct var = QProduct.variable("var");
QInventory cand = QInventory.candidate();
List<Inventory> results =
    tq.filter(cand.products.contains(var).and(var.name.startsWith("Wal")))
    .executeList();
```

This equates to the single-string query

```
SELECT FROM mydomain.Inventory
WHERE this.products.contains(var) && var.name.startsWith("Wal")
```

98.1.7 Query API - Subqueries

Let's try to find all Products that have a value below the average of all Products. This means we need to use a subquery

```java
TypesafeQuery<Product> tq = jdopm.newTypesafeQuery(Product.class);
QProduct cand = QProduct.candidate();
TypesafeSubquery<Product> tqsub = tq.subquery(Product.class, "p");
QProduct candsub = QProduct.candidate("p");
List<Product> results =
    tq.filter(cand.value.lt(tqsub.selectUnique(candsub.value.avg())))
    .executeList();
```

This equates to the single-string query

```
SELECT FROM mydomain.Product
WHERE this.value < (SELECT AVG(x.value) FROM mydomain.Product x)
```
98.1.8 Query API - Candidates

If you don't want to query instances in the datastore but instead query a collection of candidate instances, you can do this by setting the candidates, like this

```java
TypesafeQuery<Product> tq = jdopm.newTypesafeQuery(Product.class);
QProduct cand = QProduct.candidate();
List<Product> results =
    tq.filter(cand.value.lt(40.00)).setCandidates(myCandidates).executeList();
```

This will process the query **in-memory**.
99 SQL

99.1 JDO : SQL Queries
The ability to query the datastore is an essential part of any system that persists data. Sometimes an object-based query language (such as JDOQL) is considered not suitable, maybe due to the lack of familiarity of the application developer with such a query language. In this case it is desirable to query using SQL. JDO standardises this as a valid query mechanism, and DataNucleus supports this. Please be aware that the SQL query that you invoke has to be valid for your RDBMS, and that the SQL syntax differs across almost all RDBMS.

To utilise SQL syntax in queries, you create a Query as follows

```java
Query q = pm.newQuery("javax.jdo.query.SQL", the_query);
```

You have several forms of SQL queries, depending on what form of output you require.

- **No candidate class and no result class** - the result will be a List of Objects (when there is a single column in the query), or a List of Object[]s (when there are multiple columns in the query)

- **Candidate class specified, no result class** - the result will be a List of candidate class objects, or will be a single candidate class object (when you have specified "unique"). The columns of the query's result set are matched up to the fields of the candidate class by name. You need to select a minimum of the PK columns in the SQL statement.

- **No candidate class, result class specified** - the result will be a List of result class objects, or will be a single result class object (when you have specified "unique"). Your result class has to abide by the rules of JDO2 result classes (see Result Class specification) - this typically means either providing public fields matching the columns of the result, or providing setters/getters for the columns of the result.

- **Candidate class and result class specified** - the result will be a List of result class objects, or will be a single result class object (when you have specified "unique"). The result class has to abide by the rules of JDO2 result classes (see Result Class specification).

99.1.1 Setting candidate class
If you want to return instances of persistable types, then you can set the candidate class.

```java
Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT MY_ID, MY_NAME FROM MYTABLE");
query.setClass(MyClass.class);
List<MyClass> results = (List<MyClass>) query.execute();
```

99.1.2 Unique results
If you know that there will only be a single row returned from the SQL query then you can set the query as unique. Note that the query will return null if the SQL has no results.

Sometimes you know that the query can only return 0 or 1 objects. In this case you can simplify your job by adding
99.1.3 Defining a result type
If you want to dump each row of the SQL query results into an object of a particular type then you can set the result class.

```java
Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT MY_ID, MY_NAME FROM MYTABLE");
query.setClass(MyClass.class);
query.setUnique(true);
MyClass obj = (MyClass) query.execute();
```

The Result Class has to meet certain requirements. These are

- Can be one of Integer, Long, Short, Float, Double, Character, Byte, Boolean, String, java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp, or Object[].
- Can be a user defined class, that has either a constructor taking arguments of the same type as those returned by the query (in the same order), or has a public put(Object, Object) method, or public setXXX() methods, or public fields.

For example, if we are returning two columns like above, an int and a String then we define our result class like this

```java
public class MyResultClass
{
    protected int id = 0;
    protected String name = null;

    public MyResultClass(int id, String name)
    {
        this.id = id;
        this.name = name;
    }

    ...
}
```

So here we have a result class using the constructor arguments. We could equally have provided a class with public fields instead, or provided setXXX methods or a put method. They all work in the same way.

99.1.4 Inserting/Updating/Deleting
In JDO all SQL queries must begin "SELECT ...", and consequently it is not possible to execute queries that change data. In DataNucleus we have an extension that allows this to be overridden. To
enable this you should pass the property `datanucleus.query.sql.allowAll` as true when creating the PersistenceManagerFactory. Thereafter you just invoke your statements like this

```java
Query q = pm.newQuery("javax.jdo.query.SQL",
   "UPDATE MY_TABLE SET MY_COLUMN = ? WHERE MY_ID = ?");
```

you then pass any parameters in as normal for an SQL query. Note that DataNucleus currently supports queries starting "INSERT", "UPDATE", "MERGE", "DELETE" and "CREATE" as update/delete queries and so will be invoked with JDBC `executeStatementUpdate`. If you have a query starting with something other than that it will be invoked with `executeStatementQuery`. If your statement really needs to be executed differently then you should look at contributing support for those statements to DataNucleus.

### 99.1.5 Parameters

In JDO SQL queries can have parameters but must be **positional**. This means that you do as follows

```java
Query q = pm.newQuery("javax.jdo.query.SQL",
   "SELECT col1, col2 FROM MYTABLE WHERE col3 = ? AND col4 = ? and col5 = ?");
List results = (List) q.execute(val1, val2, val3);
```

So we used traditional JDBC form of parametrisation, using "?".

DataNucleus also supports two further variations. The first is called **numbered** parameters where we assign numbers to them, so the previous example could have been written like this

```java
Query q = pm.newQuery("javax.jdo.query.SQL",
   "SELECT col1, col2 FROM MYTABLE WHERE col3 = ?1 AND col4 = ?2 and col5 = ?1");
List results = (List) q.execute(val1, val2);
```

so we can reuse parameters in this variation. The second variation is called **named** parameters where we assign names to them, and so the example can be further rewritten like this

```java
Query q = pm.newQuery("javax.jdo.query.SQL",
   "SELECT col1, col2 FROM MYTABLE WHERE col3 = :firstVal AND col4 = :secondVal and col5 = :firstVal");
Map params = new HashMap();
params.put("firstVal", val1);
params.put("secondVal", val1);
List results = (List) q.executeWithMap(params);
```

### 99.1.6 Example 1 - Using SQL aggregate functions, without candidate class

Here's an example for getting the size of a table without a candidate class.

```java
Query q = pm.newQuery("javax.jdo.query.SQL",
   "SELECT COUNT(*) FROM MYTABLE");
List results = (List) q.execute();
```
Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT count(*) FROM MYTABLE");
List results = (List) query.execute();
Integer tableSize = (Integer) result.iterator().next();

Here's an example for getting the maximum and minimum of a parameter without a candidate class.

Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT max(PARAM1), min(PARAM1) FROM MYTABLE");
List results = (List) query.execute();
Object[] measures = (Object[])result.iterator().next();
Double maximum = (Double)measures[0];
Double minimum = (Double)measures[1];

99.1.7 Example 2 - Using SQL aggregate functions, with result class

Here's an example for getting the size of a table with a result class. So we have a result class of

```java
public class TableStatistics {
    private int total;

    public setTotal(int total);
}
```

So we define our query to populate this class

```java
Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT count(*) AS total FROM MYTABLE");
query.setResultClass(TableStatistics.class);
List results = (List) query.execute();
TableStatistics tableStats = (TableStatistics) result.iterator().next();
```

Each row of the results is of the type of our result class. Since our query is for an aggregate, there is actually only 1 row.

99.1.8 Example 3 - Retrieval using candidate class

When we want to retrieve objects of a particular PersistenceCapable class we specify the candidate class. Here we need to select, as a minimum, the identity columns for the class.

```java
Query query = pm.newQuery("javax.jdo.query.SQL",
    "SELECT MY_ID, MY_NAME FROM MYTABLE");
query.setClass(MyClass.class);
List results = (List) query.execute();
Iterator resultsIter = results.iterator();
while (resultsIter.hasNext())
{
    MyClass obj = (MyClass)resultsIter.next();
}
```
99.1.9 Example 4 - Using parameters, without candidate class

Here's an example for getting the number of people with a particular email address. You simply add a "?" for all parameters that are passed in, and these are substituted at execution time.

```java
Query query = pm.newQuery("javax.jdo.query.SQL", "SELECT count(*) FROM PERSON WHERE EMAIL_ADDRESS = ");
List results = (List) query.execute("nobody@datanucleus.org");
Integer tableSize = (Integer) result.iterator().next();
```

99.1.10 Example 5 - Named Query

While "named" queries were introduced primarily for JDOQL queries, we can define "named" queries for SQL also. So let's take a Product class, and we want to define a query for all products that are "sold out". We firstly add this to our MetaData
And then in our application code we utilise the query

```java
Query q = pm.newNamedQuery(Product.class, "SoldOut");
List results = (List)q.execute();
```
100 Stored Procedures

100.1 JDO : Stored Procedures
JDO doesn't include explicit support for stored procedures. However DataNucleus provides two options for allowing use of stored procedures with RDBMS datasstores.

100.1.1 Using JDO SQL Query API
In JDO all SQL queries must begin "SELECT ...", and consequently it is not possible to execute stored procedures. In DataNucleus we have an extension that allows this to be overridden. To enable this you should pass the property `datanucleus.query.sql.allowAll` as true when creating the PersistenceManagerFactory. Thereafter you just invoke your stored procedures like this

```java
Query q = pm.newQuery("javax.jdo.query.SQL", "EXECUTE sp_who");
```

Where "sp_who" is the stored procedure being invoked. Clearly the same rules will apply regarding the results of the stored procedure and mapping them to any result class. The syntax of calling a stored procedure differs across RDBMS. Some require "CALL ..." and some "EXECUTE ...". Go consult your manual.

100.1.2 Using DataNucleus Stored Procedure API

```java
import org.datanucleus.api.jdo.JDOQuery;
import org.datanucleus.store.rdbms.query.StoredProcedureQuery;
...
StoredProcedureQuery spq = (StoredProcedureQuery)((JDOQuery)q).getInternalQuery());
```

Now we can control things like parameters, and what is returned from the stored procedure query. Let's start by registering any parameters (IN, OUT, or INOUT) for our stored proc. In our example we use named parameters, but you can also use positional parameters.

```java
spq.registerParameter("PARAM1", String.class, StoredProcQueryParamMode.IN);
spq.registerParameter("PARAM2", Integer.class, StoredProcQueryParamMode.OUT);
```
Simple execution is like this (where you omit the paramValueMap if you have no input parameters).

```java
boolean hasResultSet = spq.executeWithMap(paramValueMap);
```

That method returns whether a result set is returned from the stored procedure (some return results, but some return an update count, and/or output parameters). If we are expecting a result set we then do

```java
List results = (List)spq.getNextResults();
```

and if we are expecting output parameter values then we get them using the API too. Note again that you can also access via position rather than name.

```java
Object val = spq.getOutputParameterValue("PARAM2");
```

That summarises our stored procedure API. It also allows things like multiple result sets for a stored procedure, all using the `StoredProcedureQuery` API.
101 JPQL

101.1 JDO : JPQL Queries

JDO provides a flexible API for use of query languages. DataNucleus makes use of this to allow use of the query language defined in the JPA1 specification (JPQL) with JDO persistence. JPQL is a pseudo-OO language based around SQL, and so not using Java syntax, unlike JDOQL. To provide a simple example, this is what you would do

```java
Query q = pm.newQuery("JPQL", "SELECT p FROM Person p WHERE p.lastName = 'Jones'");
List results = (List)q.execute();
```

This finds all "Person" objects with surname of "Jones". You specify all details in the query.

101.1.1 SELECT Syntax

In JPQL queries you define the query in a single string, defining the result, the candidate class(es), the filter, any grouping, and the ordering. This string has to follow the following pattern

```
SELECT [<result>]
    [FROM <candidate-class(es)>]
    [WHERE <filter>]
    [GROUP BY <grouping>]
    [HAVING <having>]
    [ORDER BY <ordering>]
```

The "keywords" in the query are shown in UPPER CASE are case-insensitive.

101.1.2 Entity Name

In the example shown you note that we did not specify the full class name. We used `Person p` and thereafter could refer to `p` as the alias. The `Person` is called the **entity name** and in JPA MetaData this can be defined against each class in its definition. With JDO we don't have this MetaData attribute so we simply define the **entity name** as the name of the class omitting the package name. So `org.datanucleus.test.samples.Person` will have an entity name of `Person`.

In strict JPA the entity name cannot be a `MappedSuperclass` entity name. That is, if you have an abstract superclass that is persistable, you cannot query for instances of that superclass and its subclasses. We consider this a significant shortcoming of the querying capability, and allow the entity name to also be of a `MappedSuperclass`. You are unlikely to find this supported in other JPA implementations, but then maybe that's why you chose DataNucleus?
101.1.3 Unique Results

When you know that there will be only a single result, you can set the query as unique. This simplifies the process of getting the result.

```java
Query query = pm.newQuery("JPQL",
    "SELECT p FROM Person p WHERE p.lastName = 'Obama' AND o.firstName = 'Barak'");
query.setUnique(true);
Person pers = (Person) query.execute();
```

101.1.4 Result Class

If you are defining the result of the JPQL query and want to obtain each row of the results as an object of a particular type, then you can set the result class.

```java
Query query = pm.newQuery("JPQL", "SELECT p.firstName, p.lastName FROM Person p");
query.setResultClass(Name.class);
List<Name> names = (List<Name>) query.execute();
```

The `Result Class` has to meet certain requirements. These are:

- Can be one of Integer, Long, Short, Float, Double, Character, Byte, Boolean, String, java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp, or Object[]
- Can be a user defined class, that has either a constructor taking arguments of the same type as those returned by the query (in the same order), or has a public put(Object, Object) method, or public setXXX() methods, or public fields.

So in our example, we are returning 2 String fields, and we define our `Result Class` `Name` as follows:

```java
public class Name
{
    protected String firstName = null;
    protected String lastName = null;

    public Name(String first, String last)
    {
        this.firstName = first;
        this.lastName = last;
    }

    ...
}
```

So here we have a result class using the constructor arguments. We could equally have provided a class with public fields instead, or provided setXXX methods or a put method. They all work in the same way.
101.1.5 Fetched Fields
By default a query will fetch fields according to their defined EAGER/LAZY setting, so fields like primitives, wrappers, Dates, and 1-1/N-1 relations will be fetched, whereas 1-N/M-N fields will not be fetched. JPQL allows you to include FETCH JOIN as a hint to include 1-N/M-N fields where possible. For RDBMS datasstores any multi-valued field will not be fetched even if you specify FETCH JOIN, due to the complications in doing so. All non-RDBMS datastores do however respect this FETCH JOIN setting, since a collection/map is stored in a single "column" in the object and so is readily retrievable.

Note that you can also make use of Fetch Groups to have fuller control over what is retrieved from each query.

101.1.6 Filter
The most important thing to remember when defining the filter for JPQL is that think how you would write it in SQL, and its likely the same except for field names instead of column names. The filter has to be a boolean expression, and can include the candidate entity, fields/properties, literals, functions, parameters, operators and subqueries.

101.1.7 Fields/Properties
In JPQL you refer to fields/properties in the query by referring to the field/bean name. For example, if you are querying a candidate entity called Product and it has a field "price", then you access it like this:

\[ \text{price < 150.0} \]

Note that, just like in Java, if you want to refer to a field/property of an entity you can prefix the field by its alias:

\[ \text{p.price < 150.0} \]

You can also chain field references if you have an entity Product (alias = p) with a field of (persistable) type Inventory, which has a field name, so you could do:

\[ \text{p.inventory.name = 'Backup'} \]

101.1.8 Operators
The operators are listed below in order of decreasing precedence.

- Navigation operator (.)
- Arithmetic operators:
  - +, - unary
  - *, / multiplication and division
  - +, - addition and subtraction
- Logical operators:
101.1.9 Literals

JPQL supports the following literals: IntegerLiteral, FloatingPointLiteral, BooleanLiteral, CharacterLiteral, StringLiteral, and NullLiteral. When String literals are specified using single-string format they should be surrounded by single-quotes ‘.’

101.1.10 Input Parameters

In JPQL queries it is convenient to pass in parameters so we don’t have to define the same query for different values. Let’s take two examples:

**Named Parameters**

```
Named Parameters:
Query q = pm.newQuery("JPQL",
    "SELECT p FROM Person p WHERE p.lastName = :surname AND o.firstName = :forename");
Map params = new HashMap();
params.put("surname", theSurname);
params.put("forename", theForename);
List<Person> results = (List<Person>)q.executeWithMap(params);
```

**Numbered Parameters**

```
Numbered Parameters:
Query q = pm.newQuery("JPQL",
    "SELECT p FROM Person p WHERE p.lastName = ?1 AND p.firstName = ?2");
List<Person> results = (List<Person>)q.execute(theSurname, theForename);
```

So in the first case we have parameters that are prefixed by : (colon) to identify them as a parameter and we use that name in the parameter map passed to execute(). In the second case we have parameters that are prefixed by ? (question mark) and are numbered starting at 1. We then pass the parameters in to execute in that order.

101.1.11 JPQL Functions

JPQL provides an SQL-like query language. Just as with SQL, JPQL also supports a range of functions to enhance the querying possibilities. The tables below also mark whether a particular method is supported for evaluation in-memory.

Please note that you can easily add support for other functions for evaluation "in-memory" using this DataNucleus plugin point
Please note that you can easily add support for other functions with RDBMS datastore using this DataNucleus plugin point

101.1.11.1 Aggregate Functions
There are a series of aggregate functions for aggregating the values of a field for all rows of the results.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Standard</th>
<th>In-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT(field)</td>
<td>Returns the aggregate count of the field (Long)</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>MIN(field)</td>
<td>Returns the minimum value of the field (type of the field)</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>MAX(field)</td>
<td>Returns the maximum value of the field (type of the field)</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>AVG(field)</td>
<td>Returns the average value of the field (Double)</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>SUM(field)</td>
<td>Returns the sum of the field value(s) (Long, Double, BigInteger, BigDecimal)</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

101.1.11.2 String Functions
There are a series of functions to be applied to String fields.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Standard</th>
<th>In-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONCAT(str_field, str_field2 [, str_fieldX])</td>
<td>Returns the concatenation of the string fields</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>SUBSTRING(str_field, num1 [, num2])</td>
<td>Returns the substring of the string field starting at position num1, and optionally with the length of num2</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>TRIM([trim_spec] [trim_char] [FROM] str_field)</td>
<td>Returns trimmed form of the string field</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>LOWER(str_field)</td>
<td>Returns the lower case form of the string field</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>UPPER(str_field)</td>
<td>Returns the upper case form of the string field</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
### 101.1.11.3 Temporal Functions
There are a series of functions for use with temporal values

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Standard</th>
<th>In-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT_DATE</td>
<td>Returns the current date (day month year) of the datastore server</td>
<td>![ Standard ]</td>
<td>![ In-Memory ]</td>
</tr>
<tr>
<td>CURRENT_TIME</td>
<td>Returns the current time (hour minute second) of the datastore server</td>
<td>![ Standard ]</td>
<td>![ In-Memory ]</td>
</tr>
<tr>
<td>CURRENT_TIMESTAMP</td>
<td>Returns the current timestamp of the datastore server</td>
<td>![ Standard ]</td>
<td>![ In-Memory ]</td>
</tr>
<tr>
<td>YEAR(dateField)</td>
<td>Returns the year of the specified date</td>
<td>![ Standard ]</td>
<td>![ In-Memory ]</td>
</tr>
<tr>
<td>MONTH(dateField)</td>
<td>Returns the month of the specified date</td>
<td>![ Standard ]</td>
<td>![ In-Memory ]</td>
</tr>
<tr>
<td>DAY(dateField)</td>
<td>Returns the day of the month of the specified date</td>
<td>![ Standard ]</td>
<td>![ In-Memory ]</td>
</tr>
<tr>
<td>HOUR(dateField)</td>
<td>Returns the hour of the specified date</td>
<td>![ Standard ]</td>
<td>![ In-Memory ]</td>
</tr>
<tr>
<td>MINUTE(dateField)</td>
<td>Returns the minute of the specified date</td>
<td>![ Standard ]</td>
<td>![ In-Memory ]</td>
</tr>
<tr>
<td>SECOND(dateField)</td>
<td>Returns the second of the specified date</td>
<td>![ Standard ]</td>
<td>![ In-Memory ]</td>
</tr>
</tbody>
</table>

### 101.1.11.4 Collection Functions
There are a series of functions for use with collection values

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Standard</th>
<th>In-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX(field)</td>
<td>Returns index number of the field element when that is the element of an indexed List field.</td>
<td>![ Standard ]</td>
<td>![ In-Memory ]</td>
</tr>
</tbody>
</table>
101.1.11.5 Arithmetic Functions
There are a series of functions for arithmetic use

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Standard</th>
<th>In-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS(numeric_field)</td>
<td>Returns the absolute value of the numeric field</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRT(numeric_field)</td>
<td>Returns the square root of the numeric field</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOD(num_field1, num_field2)</td>
<td>Returns the modulus of the two numeric fields (num_field1 % num_field2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

101.1.11.6 Other Functions
You have a further function available

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Standard</th>
<th>In-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION(name, [arg1 [,arg2 ...]])</td>
<td>Executes the specified SQL function &quot;name&quot; with the defined arguments</td>
<td></td>
<td>✗</td>
</tr>
</tbody>
</table>

101.1.12 Ordering of Results
By default your results will be returned in the order determined by the datastore, so don't rely on any particular order. You can, of course, specify the order yourself. You do this using field/property names and ASC/DESC keywords. For example

```plaintext
field1 ASC, field2 DESC
```

which will sort primarily by `field1` in ascending order, then secondarily by `field2` in descending order.

Although it is not (yet) standard JPQL, DataNucleus also supports specifying a directive for where NULL values of the ordered field/property go in the order, so the full syntax supported is

```plaintext
fieldName [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Note that this is only supported for a few RDBMS (H2, HSQLDB, PostgreSQL, DB2, Oracle, Derby).
101.1.13 Subqueries

With JPQL the user has a very flexible query syntax which allows for querying of the vast majority of data components in a single query. In some situations it is desirable for the query to utilise the results of a separate query in its calculations. JPQL also allows the use of subqueries. Here's an example

```java
SELECT Object(e) FROM org.datanucleus.Employee e
WHERE e.salary > (SELECT avg(f.salary) FROM org.datanucleus.Employee f)
```

So we want to find all Employees that have a salary greater than the average salary. The subquery must be in parentheses (brackets). Note that we have defined the subquery with an alias of "f", whereas in the outer query the alias is "e".

101.1.13.1 ALL/ANY Expressions

One use of subqueries with JPQL is where you want to compare with some or all of a particular expression. To give an example

```java
SELECT emp FROM Employee emp
WHERE emp.salary > ALL (SELECT m.salary FROM Manager m WHERE m.department = emp.department)
```

So this returns all employees that earn more than all managers in the same department! You can also compare with some/any, like this

```java
SELECT emp FROM Employee emp
WHERE emp.salary > ANY (SELECT m.salary FROM Manager m WHERE m.department = emp.department)
```

So this returns all employees that earn more than any one Manager in the same department.

101.1.13.2 EXISTS Expressions

Another use of subqueries in JPQL is where you want to check on the existence of a particular thing. For example

```java
SELECT DISTINCT emp FROM Employee emp
WHERE EXISTS (SELECT emp2 FROM Employee emp2 WHERE emp2 = emp.spouse)
```

So this returns the employees that have a partner also employed.

101.1.14 Range of Results

With JPQL you can select the range of results to be returned. For example if you have a web page and you are paginating the results of some search, you may want to get the results from a query in blocks of 20 say, with results 0 to 19 on the first page, then 20 to 39, etc. You can facilitate this as follows

```java
Query q = pm.newQuery("JPQL", "SELECT p FROM Person p WHERE p.age > 20");
q.setRange(0, 20);
```

So with this query we get results 0 to 19 inclusive.
101.1.15 Query Result

Whilst the majority of the time you will want to return instances of a candidate class, JPQL also allows you to return customised results. Consider the following example:

```java
Query q = pm.newQuery("JPQL", "SELECT p.firstName, p.lastName FROM Person p WHERE p.age > 20");
List<Object[]> results = (List<Object[]>)q.execute();
```

This returns the first and last name for each Person meeting that filter. Obviously we may have some container class that we would like the results returned in, so if we change the query to this:

```java
Query<PersonName> q = pm.newQuery("JPQL", "SELECT p.firstName, p.lastName FROM Person p WHERE p.age > 20");
q.setResultClass(PersonName.class);
List<PersonName> results = (List<PersonName>)q.execute();
```

so each result is a PersonName, holding the first and last name. This result class needs to match one of the following structures:

- Constructor taking arguments of the same types and the same order as the result clause. An instance of the result class is created using this constructor. For example

  ```java
  public class PersonName
  {
      protected String firstName = null;
      protected String lastName = null;

      public PersonName(String first, String last)
      {
          this.firstName = first;
          this.lastName = last;
      }
      ...
  }
  ```

- Default constructor, and setters for the different result columns, using the alias name for each column as the property name of the setter. For example

  ```java
  public class PersonName
  {
      protected String firstName = null;
      protected String lastName = null;

      public PersonName()
      {
      }

      public void setFirstName(String first) {this.firstName = first;}
      public void setLastName(String last) {this.lastName = last;}
      ...
  }
  ```
Note that if the setter property name doesn't match the query result component name, you should use \textit{AS \{alias\}} in the query so they are the same.

\section*{101.2 JPQL In-Memory queries}

The typical use of a JPQL query is to translate it into the native query language of the datastore and return objects matched by the query. For many datastores it is simply impossible to support the full JPQL syntax in the datastore \textit{native query language} and so it is necessary to evaluate the query in-memory. This means that we evaluate as much as we can in the datastore and then instantiate those objects and evaluate further in-memory. Here we document the current capabilities of \textit{in-memory evaluation} in DataNucleus.

\begin{itemize}
  \item Subqueries using \textit{ALL}, \textit{ANY}, \textit{SOME}, \textit{EXISTS} are not currently supported
  \item \textit{MEMBER OF} syntax is not currently supported.
\end{itemize}

To enable evaluation in memory you specify the query hint \texttt{datanucleus.query.evaluateInMemory} to \texttt{true} as follows:

\begin{verbatim}
query.setHint("datanucleus.query.evaluateInMemory","true");
\end{verbatim}

\section*{101.3 JPQL DELETE Queries}

The JPA specification defines a mode of JPQL for deleting objects from the datastore.

\subsection*{101.3.1 DELETE Syntax}

The syntax for deleting records is very similar to selecting them

\begin{verbatim}
DELETE FROM [<candidate-class>]
[WHERE <filter>]
\end{verbatim}

The "keywords" in the query are shown in \textbf{UPPER CASE} are case-insensitive.

\begin{verbatim}
Query query = pm.newQuery("JPQL", "DELETE FROM Person p WHERE firstName = 'Fred'");
Long numRowsDeleted = (Long)query.execute();
\end{verbatim}

\section*{101.4 JPQL UPDATE Queries}

The JPA specification defines a mode of JPQL for updating objects in the datastore.

\subsection*{101.4.1 UPDATE Syntax}

The syntax for updating records is very similar to selecting them
UPDATE [<candidate-class>] SET item1=value1, item2=value2
[WHERE <filter>]

The "keywords" in the query are shown in UPPERCASE are case-insensitive.

Query query = pm.newQuery("JPQL", "UPDATE Person p SET p.salary = 10000 WHERE age = 18");
Long numRowsUpdated = (Long)query.execute();

101.5 JPQL BNF Notation

The BNF defining the JPQL query language is shown below.
102 Guides

102.1 Guides for JDO

The following guides demonstrate the use of JDO using DataNucleus. If you have a guide that you think would be useful in educating users in some concepts of JDO, please contribute it via our website.

- Datastore Replication
- JavaEE Environments
- OSGi Environments
- Security
- Troubleshooting
- Performance Tuning
- Monitoring
- Logging
- Maven with DataNucleus
- Eclipse with DataNucleus
- Tutorial with RDBMS
- Tutorial with ODF
- Tutorial with Excel
- Tutorial with MongoDB
- Tutorial with HBase
- Tutorial with Neo4J
- 1-N Bidir FK Relation
- 1-N Bidir Join Relation
- M-N Relation
- M-N Attributed Relation
- Spatial Types Tutorial
- DAO Layer Design
103 Datastore Replication

103.1 JDO : Datastore Replication

Many applications make use of multiple datastores. It is a common requirement to be able to replicate parts of one datastore in another datastore. Obviously, depending on the datastore, you could make use of the datastores own capabilities for replication. DataNucleus provides its own extension to JDO to allow replication from one datastore to another. This extension doesn't restrict you to using 2 datastores of the same type. You could replicate from RDBMS to XML for example, or from MySQL to HSQLDB.

You need to make sure you have the persistence property `datanucleus.attachSameDatastore` set to `false` if using replication

Note that the case of replication between two RDBMS of the same type is usually way more efficiently replicated using the capabilities of the datastore itself

The following sample code will replicate all objects of type `Product` and `Employee` from PMF1 to PMF2. These PMFs are created in the normal way so, as mentioned above, PMF1 could be for a MySQL datastore, and PMF2 for XML. By default this will replicate the complete object graphs reachable from these specified types.

```java
import org.datanucleus.api.jdo.JDOReplicationManager;
...
JDOReplicationManager replicator = new JDOReplicationManager(pmf1, pmf2);
replicator.replicate(new Class[]{Product.class, Employee.class});
```

103.2 Example without using the JDOReplicationManager helper

If we just wanted to use pure JDO, we would handle replication like this. Let's take an example
public class ElementHolder
{
    long id;
    private Set elements = new HashSet();

    ...
}

public class Element
{
    String name;

    ...
}

public class SubElement extends Element
{
    double value;

    ...
}

so we have a 1-N unidirectional (Set) relation, and we define the metadata like this

```xml
<package name="org.datanucleus.samples">
    <class name="ElementHolder" identity-type="application" detachable="true">
        <inheritance strategy="new-table"/>
        <field name="id" primary-key="true"/>
        <field name="elements" persistence-modifier="persistent">
            <collection element-type="org.datanucleus.samples.Element"/>
            <join/>
        </field>
    </class>
    <class name="Element" identity-type="application" detachable="true">
        <inheritance strategy="new-table"/>
        <field name="name" primary-key="true"/>
    </class>
    <class name="SubElement">
        <inheritance strategy="new-table"/>
        <field name="value"/>
    </class>
</package>
```

and so in our application we create some objects in datastore1, like this
PersistenceManagerFactory pmf1 = JDOHelper.getPersistenceManagerFactory("dn.1.properties");
PersistenceManager pm1 = pmf1.getPersistenceManager();
Transaction tx1 = pm1.currentTransaction();
Object holderId = null;
try {
    tx1.begin();
    ElementHolder holder = new ElementHolder(101);
    holder.addElement(new Element("First Element"));
    holder.addElement(new Element("Second Element"));
    holder.addElement(new SubElement("First Inherited Element"));
    holder.addElement(new SubElement("Second Inherited Element"));
    pm1.makePersistent(holder);

    tx1.commit();
    holderId = JDOHelper.getObjectId(holder);
} finally {
    if (tx1.isActive()) {
        tx1.rollback();
    }
    pm1.close();
}

and now we want to replicate these objects into datastore2, so we detach them from datastore1 and attach them to datastore2, like this.
// Detach the objects from "datastore1"
ElementHolder detachedHolder = null;
pm1 = pmf1.getPersistenceManager();
tx1 = pm1.currentTransaction();
try {
    pm1.getFetchPlan().setGroups(new String[] {FetchPlan.DEFAULT, FetchPlan.ALL});
    pm1.getFetchPlan().setMaxFetchDepth(-1);

    tx1.begin();
    ElementHolder holder = (ElementHolder) pm1.getObjectById(holderID);
    detachedHolder = (ElementHolder) pm1.detachCopy(holder);
    tx1.commit();
} finally {
    if (tx1.isActive())
        tx1.rollback();
    pm1.close();
}

// Attach the objects to datastore2
PersistenceManagerFactory pmf2 = JDOHelper.getPersistenceManagerFactory("dn.2.properties");
PersistenceManager pm2 = pmf2.getPersistenceManager();
Transaction tx2 = pm2.currentTransaction();
try {
    tx2.begin();
    pm2.makePersistent(detachedHolder);
    tx2.commit();
} finally {
    if (tx2.isActive())
        tx2.rollback();
    pm2.close();
}

That's all there is. These objects are now replicated into datastore2. Clearly you can extend this basic idea and replicate large amounts of data.
104 JEE Environments

104.1 JDO : Usage of DataNucleus within a JavaEE environment

The JavaEE framework has become popular in some places in the last few years. It provides a container within which java processes operate and it provides mechanisms for, amongst other things, transactions (JTA), and for connecting to other (3rd party) utilities (using Java Connector Architecture, JCA). DataNucleus Access Platform can be utilised within a JavaEE environment via this JCA system, and we provide a Resource Adaptor (RAR file) containing this JCA adaptor allowing Access Platform to be used with the likes of WebLogic and JBoss. Instructions are provided for the following JavaEE servers

- WebLogic
- JBoss 3.0/3.2
- JBoss 4.0
- JBoss 7.0
- Jonas 4.8

The provided DataNucleus JCA rar provides default resource adapter descriptors, one general, and the other for the WebLogic JavaEE server. These resource adapter descriptors can be configured to meet your needs, for example allowing XA transactions instead of the default Local transactions.

104.1.1 Requirements

To use DataNucleus with JCA the first thing that you will require is the datanucleus-jca-{version}.rar file (available from the download section).

104.1.2 DataNucleus Resource Adaptor and transactions

A great advantage of DataNucleus implementing the ManagedConnection interface is that the JavaEE container manages transactions for you (no need to call the begin/commit/rollback-methods). Currently, local transactions and distributed (XA) transactions are supported. Within a JavaEE environment, JDO transactions are nested in JavaEE transactions. All you have to do is to declare that a method needs transaction management. This is done in the EJB meta data. Here you will see, how a SessionBean implementation could look like.

The EJB meta data is defined in a file called ejb-jar.xml and can be found in the META-INF directory of the jar you deploy. Suppose you deploy a bean called DataNucleusBean, your ejb-jar.xml should contain the following configuration elements:

```
<session>
  <ejb-name>DataNucleusBean</ejb-name>
  ...
  <transaction-type>Container</transaction-type>
  ...
</session>
```

Imagine your bean defines a method called testDataNucleusTrans():

©2015, DataNucleus • ALL RIGHTS RESERVED.
You hereby define that transaction management is required for this method. The container will automatically begin a transaction for this method. It will be committed if no error occurs or rolled back otherwise. A potential SessionBean implementation containing methods to retrieve a PersistenceManager then could look like this:

```xml
<container-transaction>
  <method>
    <ejb-name>DataNucleusBean</ejb-name>
    ...
    <method-name>testDataNucleusTrans</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
```
public abstract class DataNucleusBean implements SessionBean
{
    // EJB methods
    public void ejbCreate()
    throws CreateException
    {
    }

    public void ejbRemove()
    throws EJBException, RemoteException
    {
    }

    // convenience methods to get
    // a PersistenceManager

    /**
     * static final for the JNDI name of the PersistenceManagerFactory
     */
    private static final String PMF_JNDI_NAME = "java:/datanucleus1";

    /**
     * Method to get the current InitialContext
     */
    private InitialContext getInitialContext() throws NamingException
    {
        InitialContext initialContext = new InitialContext();
        // or other code to create the InitialContext eg. new InitialContext(myProperties);
        return initialContext;
    }

    /**
     * Method to lookup the PersistenceManagerFactory
     */
    private PersistenceManagerFactory getPersistenceManagerFactory(InitialContext context)
    throws NamingException
    {
        return (PersistenceManagerFactory) context.lookup(PMF_JNDI_NAME);
    }

    /**
     * Method to get a PersistenceManager
     */
    public PersistenceManager getPersistenceManager()
    throws NamingException
    {
        return getPersistenceManagerFactory(getInitialContext()).getPersistenceManager();
    }

    // Now finally the bean method within a transaction
    public void testDataNucleusTrans()
    throws Exception
    {
        PersistenceManager pm = getPersistenceManager()
        try
        {
            // Do something with your PersistenceManager
        } finally
        {
            // close the PersistenceManager
        }
Make sure, you close the PersistenceManager in your bean methods. If you don't, the JavaEE server will usually close it for you (one of the advantages), but of course not without a warning or error message.

To avoid the need of editing multiple files, you could use XDoclet to generate your classes and control the metadata by xdoclet tags. The method declaration then would look like this:

```java
/**
 * @ejb.interface-method
 * @ejb.transaction type="Required"
 */
public void testDataNucleusTrans()
throws Exception
{
    //...
    //...
}
```

These instructions were adapted from a contribution by a DataNucleus user Alexander Bieber.

104.1.3 Persistence Properties

When creating a PMF using the JCA adaptor, you should specify your persistence properties using a persistence.xml or jdoconfig.xml. This is because DataNucleus JCA adapter from version 1.2.2 does not support Java bean setters/getters for all properties - since it is an inefficient and inflexible mechanism for property specification. The more recent persistence.xml and jdoconfig.xml methods lead to more extensible code.

104.1.4 General configuration

A resource adapter has one central configuration file /META-INF/ra.xml which is located within the rar file and which defines the default values for all instances of the resource adapter (i.e. all instances of PersistenceManagerFactory). Additionally, it uses one or more deployment descriptor files (in JBoss, for example, they are named *.ds.xml) to set up the instances. In these files you can override the default values from the ra.xml.

Since it is bad practice (and inconvenient) to edit a library's archive (in this case the datanucleus-jca-$\{version\}.rar) for changing the configuration (it makes updates more complicated, for example), it is recommended, not to edit the ra.xml within DataNucleus' rar file, but instead put all your configuration into your deployment descriptors. This way, you have a clean separation of which files you maintain (your deployment descriptors) and which files are maintained by others (the libraries you use and which you simply replace in case of an update).

Nevertheless, you might prefer to declare default values in the ra.xml in certain circumstances, so here's an example:
To define persistence properties you should make use of persistence.xml or jdoconfig.xml and refer to the documentation for persistence properties for full details of the properties.

104.1.5 WebLogic
To use DataNucleus on Weblogic the first thing that you will require is the datanucleus-jca-{version}.rar file. You then may need to edit the /META-INF/weblogic-ra.xml file to suit the exact version of your WebLogic server (the included file is for WebLogic 8.1).

You then deploy the RAR file on your WebLogic server.

104.1.6 JBoss 3.0/3.2
To use DataNucleus on JBoss (Ver 3.2) the first thing that you will require is the datanucleus-jca-{version}.rar file. You should put this in the deploy ("$/JBOSS/server/default/deploy/") directory of your JBoss installation.

You then create a file, also in the deploy directory with name datanucleus-ds.xml. To give a guide on what this file will typically include, see the following
This example creates 3 connection factories to MySQL databases, but you can create as many or as few as you require for your system to whichever databases you prefer (as long as they are supported by DataNucleus). With the above definition we can then use the JNDI names `java:/datanucleus`, `java:/datanucleus1`, and `java:/datanucleus2` to refer to our datastores.

Note, that you can use separate deployment descriptor files. That means, you could for example create the three files `datanucleus1-ds.xml`, `datanucleus2-ds.xml` and `datanucleus3-ds.xml` with each declaring one `PersistenceManagerFactory` instance. This is useful (or even required) if you need a distributed configuration. In this case, you can use JBoss' hot deployment feature and deploy a new `PersistenceManagerFactory`, while the server is running (and working with the existing PMFs). If you create a new `*-ds.xml` file (instead of modifying an existing one), the server does not undeploy...
You are now set to work on DataNucleus-enabling your actual application. As we have said, you can use the above JNDI names to refer to the datastores, so you could do something like the following to access the PersistenceManagerFactory to one of your databases.

```java
import javax.jdo.PersistenceManagerFactory;

InitialContext context = new InitialContext();
PersistenceManagerFactory pmf = (PersistenceManagerFactory) context.lookup("java:/datanucleus1");
```

These instructions were adapted from a contribution by a DataNucleus user Marco Schulze.

### 104.1.7 JBoss 4.0

With JBoss 4.0 there are some changes in configuration relative to JBoss 3.2 in order to allow use of some new features of JCA 1.5. Here you will see how to configure JBoss 4.0 to use with DataNucleus JCA adapter for DB2.

To use DataNucleus on JBoss 4.0 the first thing that you will require is the `datanucleus-jca-{version}.rar` file. You should put this in the deploy directory (`${JBOSS}/server/default/deploy/`) of your JBoss installation. Additionally, you have to remember to put any JDBC driver files to `lib` directory (`${JBOSS}/server/default/lib/`) if JBoss does not have them installed by default. In case of DB2 you need to copy `db2jcc.jar` and `db2jcc_license_c.jar`.

You then create a file, also in the deploy directory with name `datanucleus-ds.xml`. To give a guide on what this file will typically include, see the following.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<connection-factories>
  <tx-connection-factory>
    <jndi-name>datanucleus</jndi-name>
    <rar-name>datanucleus-jca-{version}.rar</rar-name> <!-- the name here must be the same as JCA adapter filename -->
    <connection-definition>javax.resource.cci.ConnectionFactory</connection-definition>
    <config-property name="ConnectionDriverName" type="java.lang.String">com.ibm.db2.jcc.DB2Driver</config-property>
    <config-property name="UserName" type="java.lang.String">app</config-property>
    <config-property name="Password" type="java.lang.String">app</config-property>
  </tx-connection-factory>
</connection-factories>
```

You are now set to work on DataNucleus-enabling your actual application. You can use the above JNDI name to refer to the datastores, and so you could do something like the following to access the PersistenceManagerFactory to one of your databases.
import javax.jdo.PersistenceManagerFactory;

InitialContext context=new InitialContext();
PersistenceManagerFactory pmFactory=(PersistenceManagerFactory)context.lookup("java:/datanucleus");

These instructions were adapted from a contribution by a DataNucleus user Maciej Wegorkiewicz.

104.1.8 JBoss 7.0

A tutorial for running DataNucleus under JBoss 7 is available on the internet, provided by a DataNucleus user Kiran Kumar.

104.1.9 Jonas

To use DataNucleus on Jonas the first thing that you will require is the datanucleus-jca-{version}.rar file. You then may need to edit the /META-INF/jonas-ra.xml file to suit the exact version of your Jonas server (the included file is tested for Jonas 4.8).

You then deploy the RAR file on your Jonas server.

104.1.10 Transaction Support

DataNucleus JCA adapter supports both Local and XA transaction types. Local means that a transaction will not have more than one resource managed by a Transaction Manager and XA means that multiple resources are managed by the Transaction Manager. Use XA transaction, if DataNucleus is configured to use data sources deployed in application servers, or if other resources such as JMS connections are used in the same transaction, otherwise use Local transaction.

You need to configure the ra.xml file with the appropriate transaction support, which is either XATransaction or LocalTransaction. See the example:

```xml
<connector>
    <display-name>DataNucleus Connector</display-name>
    <description></description>
    <vendor-name>DataNucleus Team</vendor-name>
    <spec-version>1.0</spec-version>
    <eis-type>JDO Adaptor</eis-type>
    <version>1.0</version>
    <resourceadapter>
        <managedconnectionfactory-class>org.datanucleus.jdo.connector.ManagedConnectionFactoryImpl</managedconnectionfactory-class>
        <connectionfactory-interface>javax.resource.cci.ConnectionFactory</connectionfactory-interface>
        <connectionfactory-impl-class>org.datanucleus.jdo.connector.PersistenceManagerFactoryImpl</connectionfactory-impl-class>
        <connection-interface>javax.resource.cci.Connection</connection-interface>
        <connection-impl-class>org.datanucleus.jdo.connector.PersistenceManagerImpl</connection-impl-class>
        <transaction-support>XATransaction</transaction-support> <!-- change this line -->
    </resourceadapter>
...
104.1.11 Data Source

To use a data source, you have to configure the connection factory name in `ra.xml` file. See the example:

```xml
<connector>
  <display-name>DataNucleus Connector</display-name>
  <vendor-name>DataNucleus Team</vendor-name>
  <eis-type>JDIC Adaptor</eis-type>
  <version>1.0</version>
  <managedconnectionfactory-class>org.datanucleus.jdo.connector.ManagedConnectionFactoryImpl</managedconnectionfactory-class>
  <connectionfactory-interface>javax.resource.cci.ConnectionFactory</connectionfactory-interface>
  <connection-interface>javax.resource.cci.Connection</connection-interface>
  <connection-impl-class>org.datanucleus.jdo.connector.PersistenceManagerImpl</connection-impl-class>
  <transaction-support>XATransaction</transaction-support>
  <managedconnectionfactory-class>org.datanucleus.jdo.connector.ManagedConnectionFactoryImpl</managedconnectionfactory-class>
  <connectionfactory-interface>javax.resource.cci.ConnectionFactory</connectionfactory-interface>
  <connection-interface>javax.resource.cci.Connection</connection-interface>
  <connection-impl-class>org.datanucleus.jdo.connector.PersistenceManagerImpl</connection-impl-class>
  <transaction-support>XATransaction</transaction-support>

  <config-property>
    <config-property-name>ConnectionFactoryName</config-property-name>
    <config-property-type>java.lang.String</config-property-type>
    <config-property-value>jndiName_for_datasource_1</config-property-value>
  </config-property>

  <config-property>
    <config-property-name>ConnectionResourceType</config-property-name>
    <config-property-type>java.lang.String</config-property-type>
    <config-property-value>JTA</config-property-value>
  </config-property>

  <config-property>
    <config-property-name>ConnectionFactory2Name</config-property-name>
    <config-property-type>java.lang.String</config-property-type>
    <config-property-value>jndiName_for_datasource_2</config-property-value>
  </config-property>

  ...
</connector>
```

See also:

- (RDBMS) Data Sources usage with DataNucleus
105 OSGi Environments

105.1 JDO: Usage of DataNucleus within an OSGi environment

DataNucleus jars are OSGi bundles, and as such, can be deployed in an OSGi environment. Being an OSGi environment care must be taken with respect to class-loading. In particular the persistence property `datanucleus.primaryClassLoader` will need setting. Please refer to the following guide(s) for assistance until a definitive guide can be provided:

- Guide to use of DataNucleus with OSGi and Spring dmServer
- Guide to DataNucleus inside Eclipse RCP
- Guide to DataNucleus with Spring and Eclipse RCP
- Guide to using Log4J with DataNucleus under OSGi

Some key points around integration with OSGi are as follows:-

- Any dependent jar that is required by DataNucleus needs to be OSGi enabled. By this we mean the jar needs to have the `MANIFEST.MF` file including `ExportPackage` for the packages required by DataNucleus. Failure to have this will result in `ClassNotFoundException` when trying to load its classes.
- Use `jdo-api.jar` v3.0.1 or later since those are OSGi-enabled
- The Geronimo "jpa" jar that is included in the DataNucleus distribution is OSGi enabled.
- When using DataNucleus in an Eclipse Equinox OSGi environment set the persistence property `datanucleus.plugin.pluginRegistryClassName` as `org.datanucleus.plugin.EclipsePluginRegistry`
- When using DataNucleus in other OSGi environments set the persistence property `datanucleus.plugin.pluginRegistryClassName` to `org.datanucleus.plugin.OSGiPluginRegistry`
- If you redeploy a JDO-enabled OSGi application, likely you will need to refresh the `javax.jdo` and maybe other bundles.

105.2 HOWTO Use Datanucleus with OSGi and Spring DM

This guide was written by Jasper Siepkes.

This guide is based on my personal experience and is not the authoritative guide to using DataNucleus with OSGi and Spring DM. I’ve updated this guide to use DataNucleus 3.x and Eclipse Gemini (formerly Spring DM). I haven’t extensively tested it yet. This guide explains how to use DataNucleus, Spring, OSGi and the OSGi blueprint specification together. This guide assumes the reader is familiar with concepts like OSGi, Spring, JDO, DataNucleus etc. This guide only explains how to wire these technologies together and not how they work. Now there have been a lot of (name) changes in over a short course of time. Some webpages might not have been updated yet so to undo some of the confusion created here is the deal with Eclipse Gemini. Eclipse Gemini started out as Spring OSGi, which was later renamed to Spring Dynamic Modules or Spring DM for short. Spring DM is _NOT_ to be confused with Spring DM Server. Spring DM Server is a complete server product with management UI and tons of other features. Spring DM is the core of Spring DM Server and provides only the service / dependency injection part. At some point in time the Spring team decided to donate their OSGi efforts to the Eclipse foundation. Spring DM became Eclipse Gemini and Spring DM Server became Eclipse Virgo. The whole Spring OSGi / Spring DM / Eclipse Gemini later became standardised as the OSGi Blueprint specification. To summarise: Spring OSGi = Spring DM = Eclipse Gemini, Spring DM Server = Eclipse Virgo.

Technologies used in this guide are:

- IDE (Eclipse 3.7)
105.2.1 Step 1: Adding OSGi

The first ingredient we are adding to our platform is the OSGi implementation. In this guide we will use Eclipse Equinox as our OSGi implementation. However one could also use Apache Felix, Knoplerfish, Concierge or any other compatible OSGi implementation for this purpose. Download the "org.eclipse.osgi_3.7.1.R37x_v20110808-1106.jar" ("Framework Only" download) from the Eclipse Equinox website and put in the target platform.

105.2.2 Step 2 - Adding DI

We are now going to add the Spring, Spring ORM, Spring JDBC, Spring Transaction and Spring DM bundles to our target platform. Download the Spring Community distribution from their website "spring-framework-3.0.6.RELEASE.zip". Extract the following files to our target platform directory:

- org.springframework.aop-3.0.6.RELEASE.jar
- org.springframework.asm-3.0.6.RELEASE.jar
- org.springframework.aspects-3.0.6.RELEASE.jar
- org.springframework.beans-3.0.6.RELEASE.jar
- org.springframework.context.support-3.0.6.RELEASE.jar
- org.springframework.context-3.0.6.RELEASE.jar
- org.springframework.core-3.0.6.RELEASE.jar
- org.springframework.expression-3.0.6.RELEASE.jar
- org.springframework.jdbc-3.0.6.RELEASE.jar
- org.springframework.orm-3.0.6.RELEASE.jar
- org.springframework.spring-library-3.0.6.RELEASE.libd
- org.springframework.transaction-3.0.6.RELEASE.jar

105.2.3 Step 3 - Adding OSGi Blueprint

Download the Eclipse Gemini release from their website ("gemini-blueprint-1.0.0.RELEASE.zip") and extract the following files to our target platform:

- gemini-blueprint-core-1.0.0.RELEASE.jar
- gemini-blueprint-extender-1.0.0.RELEASE.jar
- gemini-blueprint-io-1.0.0.RELEASE.jar

105.2.4 Step 4 - Adding ORM

We are now going to add JDO and DataNucleus to our target platform.

- datanucleus-core-3.0.2.jar
- datanucleus-api-jdo-3.0.2.jar
- datanucleus-rdbms-3.0.2.jar
105.2.5 Step 5 - Adding miscellaneous bundles

The following bundles are dependencies of our core bundles and can be downloaded from the Spring Enterprise Bundle Repository:

- com.springsource.org.aopalliance-1.0.0.jar (Dependency of Spring AOP, the core AOP bundle.)
- com.springsource.org.apache.commons.logging-1.1.1.jar (Dependency of various Spring bundles, logging abstraction library.)
- com.springsource.org.postgresql.jdbc4-8.3.604.jar (PostgreSQL JDBC driver, somewhat dated.)

We now have a basic target platform. This is how the directory housing the target platform looks on my PC:

```
$ ls -las
  4 drwxrwxr-x 2 siepkes siepkes 4096 Oct 22 15:28 .
  4 drwxrwxr-x 3 siepkes siepkes 4096 Oct 22 15:29 ..
  8 -rw-r----- 1 siepkes siepkes 4615 Oct 22 15:27 com.springsource.org.aopalliance-1.0.0.jar
 68 -rw-r----- 1 siepkes siepkes 61464 Oct 22 15:28 com.springsource.org.apache.commons.logging-1.1.1.jar
472 -rw-r----- 1 siepkes siepkes 476053 Oct 22 15:28 com.springsource.org.postgresql.jdbc4-8.3.604.jar
312 -rw-r----- 1 siepkes siepkes 314358 Oct 22 11:36 datanucleus-api-jdo-3.0.2.jar
1624 -rw-r----- 1 siepkes siepkes 1658797 Oct 22 11:36 datanucleus-core-3.0.2.jar
1400 -rw-r----- 1 siepkes siepkes 1427439 Oct 22 11:36 datanucleus-rdbms-3.0.2.jar
 572 -rw-r----- 1 siepkes siepkes 578205 Aug 22 22:37 gemini-blueprint-core-1.0.0.RELEASE.jar
 180 -rw-r----- 1 siepkes siepkes 178525 Aug 22 22:37 gemini-blueprint-extender-1.0.0.RELEASE.jar
  32 -rw-r----- 1 siepkes siepkes 31903 Aug 22 22:37 gemini-blueprint-jo-1.0.0.RELEASE.jar
 208 -rw-r----- 1 siepkes siepkes 208742 Oct 2 11:36 jdo-api-3.1-rc1.jar
1336 -rw-r----- 1 siepkes siepkes 1363464 Oct 22 14:26 org.eclipse.osgi_3.7.1.R37x_v20110808-1106.jar
 320 -rw-r----- 1 siepkes siepkes 321428 Aug 18 16:50 org.springframework.aop-3.0.6.RELEASE.jar
  56 -rw-r----- 1 siepkes siepkes 53082 Aug 18 16:50 org.springframework.asm-3.0.6.RELEASE.jar
  36 -rw-r----- 1 siepkes siepkes 35557 Aug 18 16:50 org.springframework.aspects-3.0.6.RELEASE.jar
  548 -rw-r----- 1 siepkes siepkes 556590 Aug 18 16:50 org.springframework.beans-3.0.6.RELEASE.jar
  660 -rw-r----- 1 siepkes siepkes 670258 Aug 18 16:50 org.springframework.context-3.0.6.RELEASE.jar
 104 -rw-r----- 1 siepkes siepkes 101450 Aug 18 16:50 org.springframework.context.support-3.0.6.RELEASE.jar
  380 -rw-r----- 1 siepkes siepkes 382184 Aug 18 16:50 org.springframework.core-3.0.6.RELEASE.jar
  172 -rw-r----- 1 siepkes siepkes 169752 Aug 18 16:50 org.springframework.expression-3.0.6.RELEASE.jar
  384 -rw-r----- 1 siepkes siepkes 386033 Aug 18 16:50 org.springframework.jbdc-3.0.6.RELEASE.jar
  332 -rw-r----- 1 siepkes siepkes 334743 Aug 18 16:50 org.springframework.orm-3.0.6.RELEASE.jar
     4 -rw-r----- 1 siepkes siepkes  437 Oct 22 15:28 org.springframework.spring-library-3.0.6.RELEASE.jar
 1400 -rw-r----- 1 siepkes siepkes 1427439 Aug 18 16:50 org.springframework.transaction-3.0.6.RELEASE.jar
```

105.2.6 Step 6 - Set up Eclipse

Here I will show how one can create a base for an application with our newly created target platform.

Create a Target Platform in Eclipse by going to 'Window' -> 'Preferences' -> 'Plugin Development' -> 'Target Platform' and press the 'Add' button. Select 'Nothing: Start with an empty target platform', give the platform a name and point it to the directory we put all the jars/bundles in. When you are done press the 'Finish' button. Indicate to Eclipse we want to use this new platform by ticking the checkbox in front of our newly created platform in the 'Target Platform' window of the 'Preferences' screen.

Create a new project in Eclipse by going to 'File' -> 'New...' -> 'Project' and Select 'Plug-in Project' under the 'Plugin development' leaf. Give the project a name (I'm going to call it 'nl.siepkes.test.project.a' in this example). In the radiobox options 'This plugin is targetted to run with:'
select 'An OSGi framework' -> 'standard'. Click 'Next'. Untick the 'Generate an activator, a Java class that...' and press 'Finish'.

**Obviously Eclipse is not the mandatory IDE for the steps described above. Other technologies can be used instead. For this guide I used Eclipse because it is easy to explain, but for most of my projects I use Maven.** If you have the Spring IDE plugin installed (which is advisable if you use Spring) you can add a Spring Nature to your project by right clicking your project and then clicking 'Spring Tools' -> 'Add Spring Nature'. This will enable error detection in your Spring bean configuration file.

Create a directory called 'spring' in your 'META-INF' directory. In this directory create a Spring bean configuration file by right clicking the directory and click 'New...' -> 'Other...'. A menu called 'New' will popup, select 'Spring Bean Configuration File'. Call the file beans.xml.

It is important to realize that the Datanucleus plugin system uses the Eclipse extensions system and NOT the plain OSGi facilities. There are two ways to make the DataNucleus plugin system work in a plain OSGi environment:

- Tell DataNucleus to use a simplified plugin manager which does not use the Eclipse plugin system (called "OSGiPluginRegistry").
- Add the Eclipse plugin system to the OSGi platform.

We are going to use the simplified plugin manager. The upside is that its easy to setup. The downside is that is less flexible then the Eclipse plugin system. The Eclipse plugin system allows you to manage different version of DataNucleus plugins. With the simplified plugin manager you can have only _one_ version of a DataNucleus plugin in your OSGi platform at any given time.

Declare a Persistence Manager Factory Bean inside the beans.xml:

```xml
<bean id="pmf" class="nl.siepkes.util.DatanucleusOSGiLocalPersistenceManagerFactoryBean">
  <property name="jdoProperties">
    <props>
      <prop key="javax.jdo.PersistenceManagerFactoryClass">org.datanucleus.api.jdo.JDOPersistenceManagerFactory</prop>
      <!-- PostgreSQL DB connection settings. Add '?loglevel=2' to Connection URL for JDBC Connection debugging. -->
      <prop key="javax.jdo.option.ConnectionURL">jdbc:postgresql://localhost/testdb</prop>
      <prop key="javax.jdo.option.ConnectionDriverName">org.postgresql.Driver</prop>
      <prop key="javax.jdo.option.ConnectionUserName">foo</prop>
      <prop key="javax.jdo.option.ConnectionPassword">bar</prop>
      <prop key="datanucleus.storeManagerType">rdbms</prop>
      <prop key="datanucleus.autoCreateSchema">true</prop>
      <prop key="datanucleus.validateTables">true</prop>
      <prop key="datanucleus.validateColumns">true</prop>
      <prop key="datanucleus.validateConstraints">true</prop>
      <prop key="datanucleus.rdbms.CheckExistTablesOrViews">true</prop>
      <prop key="datanucleus.plugin.pluginRegistryClassName">org.datanucleus.plugin.OSGiPluginRegistry</prop>
    </props>
  </property>
</bean>

<osgi:service ref="pmf" interface="javax.jdo.PersistenceManagerFactory" />
```

You can specify all the JDO/DataNucleus options you need following the above `prop`, `key` pattern. Notice the `osgi:service` line. This exports our persistence manager as an OSGi sevice and makes it possible for other bundles to access it. Also notice that the
Persistence Manager Factory is not the normal `LocalPersistenceManagerFactoryBean` class, but instead the `_OSGiLocalPersistenceManagerFactoryBean_` class. The `OSGiLocalPersistenceManagerFactoryBean` is *NOT* part of the default DataNucleus distribution. So why do we need to use the `OSGiLocalPersistenceManagerFactoryBean` instead of the default `LocalPersistenceManagerFactoryBean`? The default `LocalPersistenceManagerFactoryBean` is not aware of the OSGi environment and expects all classes to be loaded by one single classloader (this is the case in a normal Java environment without OSGi). This makes the `LocalPersistenceManagerFactoryBean` unable to locate its plugins. The `OSGiLocalPersistenceManagerFactoryBean` is a subclass of the `LocalPersistenceManagerFactoryBean` and is aware of the OSGi environment:
```java
public class OSGiLocalPersistenceManagerFactoryBean extends LocalPersistenceManagerFactoryBean implements BundleContextAware {

    private BundleContext bundleContext;
    private DataSource dataSource;

    public DatanucleusOSGiLocalPersistenceManagerFactoryBean() {
    }

    @Override
    protected PersistenceManagerFactory newPersistenceManagerFactory(String name) {
        return JDOHelper.getPersistenceManagerFactory(name, getClassLoader());
    }

    @Override
    protected PersistenceManagerFactory newPersistenceManagerFactory(Map props) {
        ClassLoader classLoader = getClassLoader();
        props.put("datanucleus.primaryClassLoader", classLoader);
        return JDOHelper.getPersistenceManagerFactory(props, classLoader);
    }

    private ClassLoader getClassLoader() {
        ClassLoader classloader = null;
        Bundle[] bundles = bundleContext.getBundles();
        for (int x = 0; x < bundles.length; x++) {
            if ("org.datanucleus.store.rdbms".equals(bundles[x].getSymbolicName())) {
                try {
                    classloader = bundles[x].loadClass("org.datanucleus.ClassLoaderResolverImpl").getClassLoader();
                } catch (ClassNotFoundException e) {
                    e.printStackTrace();
                }
            } else {
                break;
            }
        }
        return classloader;
    }

    @Override
    public void setBundleContext(BundleContext bundleContext) {
        this.bundleContext = bundleContext;
    }
}
```
If we create a new, similar (Plug-in) project, for example 'nl.siepkes.test.project.b' we can import/use our Persistence Manager Factory service by specifying the following in its beans.xml:

```
<osgi:reference id="pmf" interface="javax.jdo.PersistenceManagerFactory" />
```

The Persistence Manager Factory (pmf) bean can then be injected into other beans as you normally would do when using Spring and JDO/DataNucleus together.

**105.2.7 Step 7 - Accessing your services from another bundle**

The reason why you are probably using OSGi is because you want to separate/modularize all kinds of code. A common use case is that you have your service layer in bundle A and another bundle, bundle B, who invokes methods in your service layer. Bundle B knows absolutely nothing about DataNucleus (ie. no imports and dependencies on DataNucleus or Datastore JDBC drivers) and will just call methods with signatures like 'public FooRecord getFooRecord(long fooId)'. When you create such a setup and access a method in bundle A from bundle B you might be surprised to find out a ClassNotFound Exception is being thrown. The ClassNotFound exception will probably be about some DataNucleus or Datastore JDBC driver class not being found. How can bundle B complain about not finding implementation classes which only belong in bundle A (which has the correct imports)? The reason for this is that when you invoke the method in bundle A from bundle B the classloader from bundle B is used to execute the method in bundle A. And since the classloader of bundle B does not have DataNucleus imports things go awry.

To solve this we need to change the ClassLoader in the ThreadContext which invokes the method in Bundle A. We could of course do this manually in every method in Bundle A but since we are already using Spring and AOP its much easier to do it that way. Create the following class (which is our aspect that is going to do the heavy lifting) in bundle A:
package nl.siepkes.util;

/**
* Aspect for setting the correct class loader when invoking a method in the
* service layer.
*/

public class BundleClassLoaderAspect implements Ordered {

    private static final int ASPECT_PRECEDENCE = 0;

    public Object setClassLoader(ProceedingJoinPoint pjp) throws Throwable {
        // Save a reference to the classloader of the caller
        ClassLoader oldLoader = Thread.currentThread().getContextClassLoader();
        // Get a reference to the classloader of the owning bundle
        ClassLoader serviceLoader = pjp.getTarget().getClass().getClassLoader();
        // Set the class loader of the current thread to the class loader of the
        // owner of the bundle
        Thread.currentThread().setContextClassLoader(serviceLoader);

        Object returnValue = null;
        try {
            // Make the actual call to the method.
            returnValue = pjp.proceed();
        } finally {
            // Reset the classloader of this Thread to the original
            // classloader of the method invoker.
            Thread.currentThread().setContextClassLoader(oldLoader);
        }

        return returnValue;
    }

    @Override
    public int getOrder() {
        return ASPECT_PRECEDENCE;
    }
}

Add the following to your Spring configuration in bundle A:

```xml
<tx:advice id="txAdvice" transaction-manager="txManager">
   <tx:attributes>
      <tx:method name="get*" read-only="true" />
      <tx:method name="*" />
   </tx:attributes>
</tx:advice>

<aop:pointcut id="fooServices" expression="execution(* nl.siepkes.service.*.*(..))" />
   <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServices" />

   <!-- Ensures the class loader of this bundle is used to invoke public methods in the service layer of classes in the package 'nl.siepkes.service' -->
   <aop:aspect id="bundleLoaderAspect" ref="bundleLoaderAspectBean">
      <aop:around pointcut-ref="fooServices" method="setClassLoader"/>
   </aop:aspect>
</aop:config>
```

Now all methods in classes in the package 'nl.siepkes.service' will always use the class loader of bundle A.

**105.3 Using DataNucleus with Eclipse RCP**

*This guide was written by Stuart Robertson.*

Using DataNucleus inside an Eclipse plugin (that is, Eclipse's Equinox OSGi runtime) should be simple, because DataNucleus is implemented as a collection of OSGi bundles. My early efforts to use DataNucleus from within my Eclipse plugins all ran into problems. First classloader problems of various kinds began to show themselves. See this post on the DataNucleus Forum for details. My initial faulty configuration was as follows:

```
model
src/main/java/...*.java    (persistent POJO classes, enhanced using Maven DataNucleus plugin)
src/main/resources/datanucleus.properties* (PMF properties)

rcp.jars
plugin.xml
META-INF/
    MANIFEST.MF  (OSGi bundle manifest)
lib/
    datanucleus-core-XXX.jar
     ...
    spring-2.5.jar

rcp.ui
plugin.xml
META-INF/
    MANIFEST.MF  (OSGi bundle manifest)
```
Using the standard pattern, I had created a "jars" plugin whose only purpose in life was to provide a way to bring all of the 3rd party jars that my "model" depends on into the Eclipse plugin world. Each of the jars in the "jars" project's lib directory were also added to the MANIFEST.MF "Bundle-ClassPath" section as follows:

```
Bundle-ClassPath:* lib\asm-3.0.jar,
lib\aspectjtools-1.5.3.jar,
lib\commons-dbcp-1.2.2.jar,
lib\commons-logging-1.1.1.jar,
lib\commons-pool-1.3.jar,
lib\geronimo-spec-jta-1.0.1B-rc2.jar,
lib\h2-1.0.63.jar,
lib\jdo2-api-2.1-SNAPSHOT.jar,
lib\datanucleus-core-XXX.jar,
lib\datanucleus-rdbms-XXX.jar,
lib\...*
lib\log4j-1.2.14.jar,
lib\model-1.0.0-SNAPSHOT.jar,
lib\persistence-api-1.0.jar,
lib\spring-2.5.jar
```

Notice that the rcp.jars plugin's lib directory contains model-1.0.0-SNAPSHOT.jar - this is the jar containing my enhanced persistent classes and PMF properties file (which I called datanucleus.properties). Also, all of the packages from all of the jars listed in the Bundle-Classpath were exported using the Export-Package bundle-header.

Note, that the plugin.xml file in the "jars" project is an empty plugin.xml file containing only <plugin></plugin>, used only to trick Eclipse into using the Plugin Editor to open the MANIFEST.MF file so the bundle info can be edited in style.

The rcp.ui plugin depends on the rcp.jars so that it can "see" all of the necessary classes. Inside the Bundle Activator class in my UI plugin I initialized DataNucleus as normal, creating a PersistenceManagerFactory from the embedded datanucleus.properties file.

It all looks really promising, but doesn't work due to all kinds of classloading issues.

### 105.3.1 DataNucleus jars as plugins

The first part of the solution was to use the DataNucleus as a set of Eclipse plugins. Initially I wasn't sure where to get MANIFEST.MF and plugin.xml files to do this, but I later discovered that each of the datanucleus jar files are already packaged as Eclipse plugins. Open any of the datanucleus jar files up and you'll see an OSGi manifest and Eclipse plugin.xml. All that was needed was to copy datanucleus-XXX.jar into $ECLIPSE_HOME/plugins directory and restart Eclipse.

Once this was done, I removed the datanucleus jar files from my lib/ directory and instead modified my jars plugin, removing the datanucleus jars and all datanucleus packages from Bundle-Classpath and Export-Package. Next, I modified my rcp.ui plugin to depend not only on rcp.jars, but also on all of the datanucleus plugins. The relevant section of my rcp.ui plugin's manifest were changed to:
Require-Bundle: org.eclipse.core.runtime, 
org.datanucleus, 
org.datanucleus.enhancer, 
org.datanucleus.store.rdbms,

This moved things along, resulting in the following message:

javax.jdo.JDOException: Class org.datanucleus.store.rdbms.RDBMSManager was not found in the CLASSPATH. Please check your specification and your CLASSPATH.

Turns out that the class that could not be found was not org.datanucleus.store.rdbms.RDBMSManager, but rather my H2 database driver class. I figured the solution might lie in using Eclipse's buddy-loading mechanism to allow the *org.datanucleus.store.rdbms* plugin to see my JDBC driver, which is was packaged into my 'jars' plugin. Thus, I added the following to rcp.ui's MANIFEST.MF:

Eclipse-RegisterBuddy: org.datanucleus.store.rdbms

That too, didn't work. Checking the org.datanucleus.store.rdbms MANIFEST.MF showed no 'Eclipse-BuddyPolicy: registered' entry, so Eclipse-RegisterBuddy: org.datanucleus.store.rdbms wouldn't have helped anyway. If you are new to Eclipse's classloading ways, I can highly recommend you read "A Tale of Two VMs", as you'll likely run into the need for buddy-loading sooner or later.

105.3.2 PrimaryClassLoader saves the day

Returning to Erik Bengtson's example (about half-way down the post) gave me inspiration:

```
//set classloader for driver (using classloader from the "rcp.jars" bundle)
ClassLoader clrDriver = Platform.getBundle("rcp.jars").loadClass("org.h2.Driver").getClassLoader();
map.put("org.datanucleus.primaryClassLoader", clrDriver);

//set classloader for DataNucleus (using classloader from the "org.datanucleus" bundle)
ClassLoader clrDN = Platform.getBundle("org.datanucleus").loadClass("org.datanucleus.api.jdo.JDOPersistenceManagerFactory").getClassLoader();

PersistencemanagerFactory pmf = JDOHelper.getPersistenceManagerFactory(map, clrDN);
```

With the above change made, things worked. So, in summary

- Don't embed DataNucleus jars inside your plugin
- Do install DataNucleus jars into Eclipse/plugins and add dependencies to them from your plugin's MANIFEST
- Do tell DataNucleus which classloader to use for both its primaryClassLoader and for its own implementation

105.4 DataNucleus + Eclipse RCP + Spring

This guide was written by Stuart Robertson.
In my application, I have used Spring's elegant JdoDaoSupport class to implement my DAOs, have used Spring's BeanFactory to instantiate PersistenceManagerFactory and DAO instances and have set up declarative transaction management. See the Spring documentation section 12.3 if you are unfamiliar with Spring's JDO support. I assumed, naively, that since my code all worked when built and unit-tested in a plain Java world (with Maven 2 building my jars and running my unit-tests), that it would work inside Eclipse. I found out above that using DataNucleus inside Eclipse RCP application needs a little special attention to classloading. Once this has been taken care of, you'll know that you need to provide your PersistenceManagerFactory with the correct classloader to use as "primaryClassLoader". However, since everything is going to be instantiated by the Spring bean container, it somehow has to know what "the correct classloader" is. The recipe is fairly simple.

105.4.1 Add a Factory-bean and factory-method

At first I wasn't sure what needed doing, but a little browsing of the Spring documentation revealed what I needed (see section 3.2.3.2.3. Instantiation using an instance factory method). Spring provides a mechanism whereby a Spring beans definition file (beans.xml, in my case) can defer the creation of an object to either a static method on some factory class, or a non-static (instance) method one some factory bean. The following quote from the Spring documentation describes how things are meant to work:

In a fashion similar to instantiation via a static factory method, instantiation using an instance factory method is where a non-static method of an existing bean from the container is invoked to create a new bean. To use this mechanism, the 'class' attribute must be left empty, and the 'factory-bean' attribute must specify the name of a bean in the current (or parent/ancestor) container that contains the instance method that is to be invoked to create the object. The name of the factory method itself must be set using the 'factory-method' attribute.

The example bean definitions below show how a bean can be created using this pattern:

```xml
<!-- the factory bean, which contains a method called createService() -->
<bean id="serviceLocator" class="com.foo.DefaultServiceLocator">
    <!-- inject any dependencies required by this locator bean -->
</bean>

<!-- the bean to be created via the factory bean -->
<bean id="exampleBean" factory-bean="serviceLocator" factory-method="createService"/>
```

105.4.2 Add a little ClassLoaderFactory

In my case, I replaced the "serviceLocator" factory bean with a "classloaderFactory" bean with factory-methods that return Classloader instances, as shown below:
The two public methods, jdbcClassloader() and dnClassloader(), ask the bundle Activator to load a particular class, and then return the Classloader that was used to load the class. Note that Activator is the standard bundle activator created by Eclipse. OSGi classloading is based on a setup where each bundle has its own classloader. For example, if bundle A depends on bundles B and C, attempting to load a class (ClassC, say) provided by bundle C will result in bundle A's classloader delegating the class-load to bundle C. Calling getClassLoader() on the loaded ClassC will return bundle C's classloader, not bundle A's classloader. And this is exactly the behaviour we need. Thus, asking Activator's classloader to load "org.h2.Driver" will ultimately delegate the loading to the classloader associated with the bundle that contains the JDBC driver classes. Likewise with "org.datanucleus.api.jdo.JDOPersistenceManagerFactory".

105.4.3 Mix well

Now we have all of the pieces needed to configure our Spring beans. The bean definitions below are a part of a larger beans.xml file, but show the relevant setup. The list below describes each of the beans working from top to bottom, where the text in bold is the bean id:

- **placeholderConfigurer**: This is a standard Spring property configuration mechanism that loads a properties file from the classpath location "classpath:/config/jdbc.${datanucleus.profile}.properties", where ${datanucleus.profile} represents the value of the
"datanucleus.profile" environment variable which I set externally so that I can switch between in-memory, on-disk embedded or on-disk server DB configurations.

- **dataSource**: A JDBC DataSource (using Apache DBCP's connection pooling DataSource). Values for the properties ${jdbc.driverClassName}, ${jdbc.url}, etc are obtained from the properties file that was loaded by **placeholderConfigurer**.

- **pmf**: The DataNucleus PersistenceManagerFactory (implementation) that underpins the entire persistence layer. It's a fairly standard setup, with a reference to *dataSource* being stored in connectionFactory. The important part for this discussion is the **primaryClassLoaderResolver** part, which stores a reference to a Classloader instance (a Classloader "bean", that is).

- **classloaderFactory** and **jdbcClassloader**: Here we pull in the factory-bean pattern discussed above. When asked for the **jdbcClassloader** bean (which is a Classloader instance), Spring will defer to **classloaderFactory**, creating an instance of ClassLoaderFactory and then calling its jdbcClassloader() method to obtain the Classloader that is to become the **jdbcClassloader** bean. This works, because the the Spring jar is able to "see" my ClassLoaderFactory class. If the Spring jar is contained in one bundle, A, say, and your factory class is in some other bundle, B, say, then you may encounter ClassNotFoundException if bundle A doesn't depend on bundle B. This is normally the case if you follow the "jars plugin" pattern, creating a single plugin to house all third-party jars. In this case, you will need to add "Eclipse-BuddyPolicy: registered" to the "jars" plugin's manifest, and then add "Eclipse-RegisterBuddy: <jars.bundle.symbolicname>" to the manifest of the bundle that houses your factory class (where <jars.bundle.symbolicname> must be replaced with the actual symbolic name of the bundle). See A Tale of Two VMs if this is Greek to you.
<bean id="placeholderConfigurer" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" p:location="classpath:/config/jdbc.${datanucleus.profile}.properties" /

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close"
    p:driverClassName="${jdbc.driverClassName}"
    p:url="${jdbc.url}"
    p:username="${jdbc.username}"
    p:password="${jdbc.password}" />

<bean id="pmf" class="org.datanucleus.api.jdo.JDOPersistenceManagerFactory"
    destroy-method="close"
    p:connectionFactory-ref="dataSource"
    p:attachmentSameDatastore="true"
    p:autoCreateColumns="true"
    p:autoCreateSchema="true"
    p:autoStartMechanism="None"
    p:detachAllOnCommit="true"
    p:detachOnClose="false"
    p:nontransactionalRead="true"
    p:stringDefaultLength="255"
    p:primaryClassLoaderResolver-ref="jdbcClassloader" />

<bean id="classloaderFactory" class="rcp.model.ClassLoaderFactory" />

<!-- the bean to be created via the factory bean -->
<bean id="jdbcClassloader"
    factory-bean="classloaderFactory"
    factory-method="jdbcClassloader" />

105.4.4 Enjoy

Now that the hard-work is done, we can ask Spring to do its magic:
private void loadSpringBeans()
{
    if (beanFactory == null)
    {
        beanFactory = new ClassPathXmlApplicationContext("/config/beans.xml", Activator.class);
    }
    this.daoFactory = (IDAOFactory) beanFactory.getBean("daoFactory");
}

private void testDAO()
{
    IAccountDAO accountsDAO = this.daoFactory.accounts();
    accountsDAO.persist(entities.newAccount("Account A", AccountType.Asset));
    accountsDAO.persist(entities.newAccount("Account B", AccountType.Bank));
    List<IAccount> accounts = accountsDAO.findAll();
}

Finally, I should clarify things by mentioning that in my code, my bundle Activator provides the
loadSpringBeans() method and calls it when the bundle is started. Other classes, such as the main
application, then use Activator.getDefault().getDAOFactory() to obtain a reference to IDAOFactory,
which is another Spring bean that provides a central point of reference to all of the DAOs in the
system. All of the DAOs themselves are Spring beans too.

105.4.5 Postscript

Someone asked to see the complete applicationContext.xml (referred to as /config/beans.xml in the
loadSpringBeans() method above), so here it is:
<?xml version="1.0" encoding="UTF-8"?>
<beans
xmlns="http://www.springframework.org/schema/beans"
xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:p="http://www.springframework.org/schema/p"
xmlns:tx="http://www.springframework.org/schema/tx"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans-2.5.xsd

<!-- Enable the use of @Autowired annotations. -->
<context:annotation-config />

<!-- ====== MAIN ENTRY-POINTS ====== -->
<bean
id="daoFactory"
class="ca.eulogica.bb.model.dao.impl.DAOFactory"
p:accountDAO-ref="accountDAO"
p:budgetDAO-ref="budgetDAO"
p:budgetItemDAO-ref="budgetItemDAO"
p:commodityDAO-ref="commodityDAO"
p:institutionDAO-ref="institutionDAO"
p:splitDAO-ref="splitDAO"
p:transactionDAO-ref="transactionDAO" />

<bean
id="entityFactory"
class="ca.eulogica.bb.model.entities.impl.EntityFactory" />

<bean
id="servicesFactory"
class="ca.eulogica.bb.model.services.impl.ServicesFactory"
p:accountService-ref="accountService"
p:transactionService-ref="transactionService" />

<!-- ====== BUSINESS SERVICES ====== -->
<bean
id="accountService"
class="ca.eulogica.bb.model.services.impl.AccountService"
p:DAOFactory-ref="daoFactory"
p:entityFactory-ref="entityFactory" />

<bean
id="transactionService"
class="ca.eulogica.bb.model.services.impl.TransactionService"
p:DAOFactory-ref="daoFactory"
p:entityFactory-ref="entityFactory" />

<!-- ====== DAO ====== -->
<bean
id="accountDAO"
class="ca.eulogica.bb.model.dao.impl.AccountDAO"
p:persistenceManagerFactory-ref="pmt" />

<bean
id="budgetDAO"
106 Troubleshooting

106.1 JDO : Troubleshooting

This section describes the most common problems found when using DataNucleus in different architectures. It describes symptoms and methods for collecting data for troubleshooting thus reducing time to narrow the problem down and come to a solution.

106.2 Out Of Memory error

106.2.1 Introduction

Java allocate objects in the runtime memory data area called heap. The heap is created on virtual machine start-up. The memory allocated to objects are reclaimed by Garbage Collectors when the object is no longer referenced (See Object References). The heap may be of a fixed size, but can also be expanded when more memory is needed or contracted when no longer needed. If a larger heap is needed and it cannot be allocated an OutOfMemory is thrown. See JVM Specification.

Native memory is used by the JVM to perform its operations like creation of threads, sockets, jdbc drivers using native code, libraries using native code, etc.

The maximum size of heap memory is determined by the -Xmx on the java command line. If Xmx is not set, then the JVM decides for the maximum heap. The heap and native memory are limited to the maximum memory allocated by the JVM. For example, if the JVM Xmx is set to 1GB and currently use of native memory is 256MB then the heap can only use 768MB.

106.2.2 Causes

Common causes of out of memory:

- Not enough heap - The JVM needs more memory to deal with the application requirements. Queries returning more objects than usual can be the cause.
- Not enough PermGen - The JVM needs more memory to load class definitions.
- Memory Leaks - The application does not close the resources, like the PersistenceManager or Queries, and the JVM cannot reclaim the memory.
- Caching - Caching in the application or inside DataNucleus holding strong references to objects.
- Garbage Collection - If no full garbage collection is performed before the OutOfMemory it can indicate a bug in the JVM Garbage Collector.
- Memory Fragmentation - A large object needs to be placed in the memory, but the JVM cannot allocate a continous space to it because the memory is fragmented.
- JDBC driver - a bug in the JDBC driver not flushing resources or keeping large result sets in memory.

106.2.3 Troubleshooting

106.2.3.1 JVM

Collect garbage collection information by adding -verbosegc to the java command line. The verbosegc flag will print garbage collections to System output.
106.2.3.2 Sun JVM
The Sun JVM 1.4 or upper accepts the flag `-XX:+PrintGCDetails`, which prints detailed information on Garbage Collections. The Sun JVM accepts the flag `-verbose:class`, which prints information about each class loaded. This is useful to troubleshoot issues when OutOfMemory occurs due to lack of space in the PermGen, or when NoClassDefFoundError or Linkage errors occurs. The Sun JVM 1.5 or upper accepts the flag `-XX:+HeapDumpOnOutOfMemoryError`, which creates a hprof binary file head dump in case of an OutOfMemoryError. You can analyse the heap dump using tools such as jhat or YourKit profiler.

106.2.3.3 DataNucleus
DataNucleus keeps in cache persistent objects using weak references by default. Enable debug mode `DataNucleus.Cache` category to investigate the size of the cache in DataNucleus.

106.2.4 Resolution
DataNucleus can be configured to reduce the number of objects in cache. DataNucleus has cache for persistent objects, metadata, datastore metadata, fields of type Collection or Map, or query results.

106.2.4.1 Query Results Cache
The query results hold strong references to the retrieved objects. If a query returns too many objects it can lead to OutOfMemory error. To be able to query over large result sets, change the result set type to `scroll-insensitive` in the pmf setting `datanucleus.rdbms.query.resultSetType`.

106.2.4.2 Query leak
The query results are kept in memory until the PersistenceManager or Query are closed. To avoid memory leaks caused by queries in memory, it’s capital to explicitly close the query as soon as possible. The following snippet shows how to do it.

```java
Query query = pm.newQuery("SELECT FROM org.datanucleus.samples.store.Product WHERE price < :limit");
List results = (List)query.execute(new Double(200.0));
//...
//...
//closes the query
query.closeAll();
```

106.2.4.3 PersistenceManager leak
It’s also a best practice to ensure the PersistenceManager is closed in a try finally block. The PersistenceManager has level 1 cache of persistence objects. See the following example:
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try {
    tx.begin();
    //...
    tx.commit();
} finally {
    if (tx.isActive())
        tx.rollback();
    pm.close();
}

106.2.4.4 Cache for fields of Collection or Map
If collection or map fields have large number of elements, the caching of elements can be disabled with the property `datanucleus.cache.collections` setting it to false.

106.2.4.5 Persistent Objects cache
The cache control of persistent objects is described in the Cache Guide

106.2.4.6 Metadata and Datastore Metadata cache
The metadata and datastore metadata caching cannot be controled by the application, because the memory required for it is insignificant.

106.2.4.7 OutOfMemory when persisting new objects
When persistent many objects, the flush operation should be periodically invoked. This will give a hint to DataNucleus to flush the changes to the database and release the memory. In the below sample the `pm.flush()` operation is invoked on every 10,000 objects persisted.
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try {
    tx.begin();
    for (int i=0; i<100000; i++) {
        Wardrobe wardrobe = new Wardrobe();
        wardrobe.setModel("3 doors");
        pm.makePersistent(wardrobe);
        if (i % 10000 == 0) {
            pm.flush();
        }
    }
    tx.commit();
} finally {
    if (tx.isActive()) {
        tx.rollback();
    }
    pm.close();
}

106.3 Frozen application

106.3.1 Introduction
The application pauses for short or long periods or hangs during very long time.

106.3.2 Causes
Common causes:

- Database Locking - Database waiting other transactions to release locks due to deadlock or locking contentions.
- Garbage Collection Pauses - The garbage collection pauses the application to free memory resources.
- Application Locking - Thread 2 waiting for resources locked by Thread 1.

106.3.3 Troubleshooting

106.3.3.1 Database locking
Use a database specific tool or database scripts to find the current database locks. In Microsoft SQL, the stored procedured `sp_lock` can be used to examine the database locks.
106.3.3.2 Query Timeout
To avoid database locking to hang the application when a query is performed, set the query timeout. See Query Timeout.

106.3.3.3 Garbage Collection pauses
Check if the application freezes when the garbage collection starts. Add -verbosegc to the java command line and restart the application.

106.3.3.4 Application Locking
Thread dumps are snapshots of the threads and monitors in the JVM. Thread dumps help to diagnose applications by showing what the application is doing at a certain moment of time. To generate Thread Dumps in MS Windows, press <ctrl>+<break> in the window running the java application. To generate Thread Dumps in Linux/Unix, execute kill -3 process_id
To effectively diagnose a problem, take 5 Thread Dumps with 3 to 5 seconds internal between each one. See An Introduction to Java Stack Traces.

106.4 Postgres

106.4.1 ERROR: schema does not exist

106.4.1.1 Problem
Exception org.postgresql.util.PSQLException: ERROR: schema "PUBLIC" does not exist raised during transaction.

106.4.1.2 Troubleshooting

- Verify that the schema "PUBLIC" exists. If the name is lowercased ("public"), set datanucleus.identifier.case=PreserveCase, since Postgres is case sensitive.
- Via pgAdmin Postgres tool, open a connection to the schema and verify it is accessible with issuing a SELECT 1 statement.

106.5 Command Line Tools

106.5.1 CreateProcess error=87

106.5.1.1 Problem
CreateProcess error=87 when running DataNucleus tools under Microsoft Windows OS.
Windows has a command line length limitation, between 8K and 64K characters depending on the Windows version, that may be triggered when running tools such as the Enhancer or the SchemaTool with too many arguments.

106.5.1.2 Solution
When running such tools from Maven or Ant, disable the fork mechanism by setting the option fork="false".
107 Performance Tuning

107.1 Performance Tuning

DataNucleus, by default, provides certain functionality. In particular circumstances some of this functionality may not be appropriate and it may be desirable to turn on or off particular features to gain more performance for the application in question. This section contains a few common tips.

107.1.1 Enhancement

You should perform enhancement before runtime. That is, do not use java agent since it will enhance classes at runtime, when you want responsiveness from your application.

107.1.2 Schema : Creation

DataNucleus provides 4 persistence properties `datanucleus.autoCreateSchema`, `datanucleus.autoCreateTables`, `datanucleus.autoCreateColumns`, and `datanucleus.autoCreateConstraints` that allow creation of the datastore tables. This can cause performance issues at startup. We recommend setting these to false at runtime, and instead using SchemaTool to generate any required database schema before running DataNucleus (for RDBMS, HBase).

107.1.3 Schema : O/R Mapping

Where you have an inheritance tree it is best to add a discriminator to the base class so that it’s simple for DataNucleus to determine the class name for a particular row. For RDBMS: this results in cleaner/simpler SQL which is faster to execute, otherwise it would be necessary to do a UNION of all possible tables. For other datastores the instantiation of objects on retrieval ought to be faster with a discriminator since there is no work needed to determine the type of the object.

107.1.4 Schema : Validation

DataNucleus provides 3 persistence properties `datanucleus.validateTables`, `datanucleus.validateConstraints`, `datanucleus.validateColumns` that enforce strict validation of the datastore tables against the Meta-Data defined tables. This can cause performance issues at startup. In general this should be run only at schema generation, and should be turned off for production usage. Set all of these properties to false. In addition there is a property `datanucleus.rdbms.CheckExistTablesOrViews` which checks whether the tables/views that the classes map onto are present in the datastore. This should be set to false if you require fast start-up. Finally, the property `datanucleus.rdbms.InitializeColumnInfo` determines whether the default values for columns are loaded from the database. This property should be set to NONE to avoid loading database metadata.

To sum up, the optimal settings with schema creation and validation disabled are:
### 107.1.5 PersistenceManagerFactory usage

Creation of **PersistenceManagerFactory** objects can be expensive and should be kept to a minimum. Depending on the structure of your application, use a single factory per datastore wherever possible. Clearly if your application spans multiple servers then this may be impractical, but should be borne in mind.

You can improve startup speed by setting the property `datanucleus.autoStartMechanism` to `None`. This means that it won't try to load up the classes (or better said the metadata of the classes) handled the previous time that this schema was used. If this isn't an issue for your application then you can make this change. Please refer to the **Auto-Start Mechanism** for full details.

Some RDBMS (such as Oracle) have trouble returning information across multiple catalogs/schemas and so, when DataNucleus starts up and tries to obtain information about the existing tables, it can take some time. This is easily remedied by specifying the catalog/schema name to be used - either for the PMF as a whole (using the persistence properties `javax.jdo.mapping.Catalog`, `javax.jdo.mapping.Schema`) or for the package/class using attributes in the MetaData. This subsequently reduces the amount of information that the RDBMS needs to search through and so can give significant speed ups when you have many catalogs/schemas being managed by the RDBMS.

### 107.1.6 Database Connection Pooling

DataNucleus, by default, will allocate connections when they are required. It then will close the connection. In addition, when it needs to perform something via JDBC (RDBMS datastores) it will allocate a PreparedStatement, and then discard the statement after use. This can be inefficient relative to a database connection and statement pooling facility such as Apache DBCP. With Apache DBCP a Connection is allocated when required and then when it is closed the Connection isn't actually closed but just saved in a pool for the next request that comes in for a Connection. This saves the time taken to establish a Connection and hence can give performance speed ups the order of maybe 30% or more.

You can read about how to enable connection pooling with DataNucleus in the **Connection Pooling Guide**.

As an addendum to the above, you could also turn on caching of PreparedStatements. This can also give a performance boost, depending on your persistence code, the JDBC driver and the SQL being issued. Look at the persistence property `datanucleus.connectionPool.maxStatements`.

```java
# schema creation
datanucleus.autoCreateSchema=false
datanucleus.autoCreateTables=false
datanucleus.autoCreateColumns=false
datanucleus.autoCreateConstraints=false

# schema validation
datanucleus.validateTables=false
datanucleus.validateConstraints=false
datanucleus.validateColumns=false
datanucleus.rdbms.CheckExistTablesOrViews=false
datanucleus.rdbms.initializeColumnInfo=None
```
107.1.7 PersistenceManager usage

Clearly the structure of your application will have a major influence on how you utilise a PersistenceManager. A pattern that gives a clean definition of process is to use a different persistence manager for each request to the data access layer. This reduces the risk of conflicts where one thread performs an operation and this impacts on the successful completion of an operation being performed by another thread. Creation of PM's is not an expensive process and use of multiple threads writing to the same persistence manager should be avoided.

Make sure that you always close the PersistenceManager after use. It releases all resources connected to it, and failure to do so will result in memory leaks. Also note that when closing the PersistenceManager if you have the persistence property datanucleus.detachOnClose set to true this will detach all objects in the Level1 cache. Disable this if you don't need these objects to be detached, since it can be expensive when there are many objects.

107.1.8 Persistence Process

To optimise the persistence process for performance you need to analyse what operations are performed and when, to see if there are some features that you could disable to get the persistence you require and omit what is not required. If you think of a typical transaction, the following describes the process

- Start the transaction (if running non-transactional then this is seamless)
- Perform persistence operations.
  - If you are using "optimistic" transactions then all datastore operations will be delayed until commit. Otherwise all datastore operations will default to being performed immediately. If you are handling a very large number of objects in the transaction you would benefit by either disabling "optimistic" transactions, or alternatively setting the persistence property datanucleus.flush.mode to AUTO, or alternatively again do a manual flush every "n" objects, like this

```java
for (int i=0;i<1000000;i++)
{
  if ((i%10000)/10000 == 0 && i != 0)
  {
    pm.flush();
  }
  ...
}
```

- If you are retrieving any object by its identity (pm.getObjectById) and know that it will be present in the Level2 cache, for example, you can set the persistence property datanucleus.findObject.validateWhenCached to false and this will skip a separate call to the datastore to validate that the object exists in the datastore.
- Commit the transaction (if running non-transactional then this happens immediately after your persistence operation, seamlessly).
  - All dirty objects are flushed.
  - DataNucleus verifies if newly persisted objects are memory reachable on commit, if they are not, they are removed from the database. This process mirrors the garbage collection, where objects not referenced are garbage collected or removed from memory. Reachability is expensive because it traverses the whole object tree and may require reloading data from database. If reachability is not needed by your
application, you should disable it. To disable reachability set the persistence property `datanucleus.persistenceByReachabilityAtCommit` to `false`.

- DataNucleus will, by default, perform a check on any bidirectional relations to make sure that they are set at both sides at commit. If they aren’t set at both sides then they will be made consistent. This check process can involve the (re-)loading of some instances. You can skip this step if you always set both sides of a relation by setting the persistence property `datanucleus.manageRelationships` to `false`.
- Objects enlisted in the transaction are put in the Level 2 cache. You can disable the level 2 cache with the persistence property `datanucleus.cache.level2.type` set to `none`.
- Objects enlisted in the transaction are detached if you have the persistence property `datanucleus.detachAllOnCommit` set to `true`. Disable this if you don’t need these objects to be detached.

### 107.1.9 Identity Generators

DataNucleus provides a series of value generators for generation of identity values. These can have an impact on the performance depending on the choice of generator, and also on the configuration of the generator.

- The `sequence` strategy allows configuration of the datastore sequence. The default can be non-optimum. As a guide, you can try setting `key-cache-size` to 10.
- The `max` strategy should not really be used for production since it makes a separate DB call for each insertion of an object. Something like the `increment` strategy should be used instead. Better still would be to choose `native` and let DataNucleus decide for you.

The `native` identity generator value is the recommended choice since this will allow DataNucleus to decide which identity generator is best for the datastore in use.

### 107.1.10 Collection/Map caching

DataNucleus has 2 ways of handling calls to SCO Collections/Maps. The original method was to pass all calls through to the datastore. The second method (which is now the default) is to cache the collection/map elements/keys/values. This second method will read the elements/keys/values once only and thereafter use the internally cached values. This second method gives significant performance gains relative to the original method. You can configure the handling of collections/maps as follows:

- **Globally for the PMF** - this is controlled by setting the persistence property `datanucleus.cache.collections`. Set it to `true` for caching the collections (default), and `false` to pass through to the datastore.
- **For the specific Collection/Map** - this overrides the global setting and is controlled by adding a MetaData `<collection>` or `<map>` extension `cache`. Set it to `true` to cache the collection data, and `false` to pass through to the datastore.

The second method also allows a finer degree of control. This allows the use of lazy loading of data, hence elements will only be loaded if they are needed. You can configure this as follows:

- **Globally for the PMF** - this is controlled by setting the property `datanucleus.cache.collections.lazy`. Set it to `true` to use lazy loading, and set it to `false` to load the elements when the collection/map is initialised.
- **For the specific Collection/Map** - this overrides the global PMF setting and is controlled by adding a MetaData `<collection>` or `<map>` extension `cache-lazy-loading`. Set it to `true` to use lazy loading, and `false` to load once at initialisation.
107.1.11 NonTransactional Reads (Reading persistent objects outside a transaction)

Performing non-transactional reads has advantages and disadvantages in performance and data freshness in cache. The objects read are held cached by the PersistenceManager. The second time an application requests the same objects from the PersistenceManager they are retrieved from cache. The time spent reading the object from cache is minimum, but the objects may become stale and not represent the database status. If fresh values need to be loaded from the database, then the user application should first call `refresh` on the object.

Another disadvantage of performing non-transactional reads is that each operation realized opens a new database connection, but it can be minimized with the use of connection pools, and also on some of the datastore the (nontransactional) connection is retained.

107.1.12 Accessing fields of persistent objects when not managed by a PersistenceManager

Reading fields of unmanaged objects (outside the scope of a `PersistenceManager`) is a trivial task, but performed in a certain manner can determine the application performance. The objective here is not give you an absolute response on the subject, but point out the benefits and drawbacks for the many possible solutions.

- Use `makeTransient` to get transient versions of the objects. Note that to recurse you need to call the `makeTransient` method which has a boolean argument "useFetchPlan".

```java
Object pc = null;
try {
    PersistenceManager pm = pmf.getPersistenceManager();
    pm.currentTransaction().begin();

    //retrieve in some way the object, query, getObjectById, etc
    pc = pm.getObjectById(id);
    pm.makeTransient(pc);

    pm.currentTransaction().commit();
} finally {
    pm.close();
}
//read the persistent object here
System.out.println(pc.getName());
```

- Use `RetainValues=true`.

©2015, DataNucleus • ALL RIGHTS RESERVED.
Object pc = null;
try {
    PersistenceManager pm = pmf.getPersistenceManager();
    pm.currentTransaction().setRetainValues(true);
    pm.currentTransaction().begin();
    //retrieve in some way the object, query, getObjectById, etc
    pc = pm.getObjectById(id);
    pm.currentTransaction().commit();
} finally {
    pm.close();
}
//read the persistent object here
System.out.println(pc.getName());

- Use detachCopy method to return detached instances.

Object copy = null;
try {
    PersistenceManager pm = pmf.getPersistenceManager();
    pm.currentTransaction().begin();
    //retrieve in some way the object, query, getObjectById, etc
    Object pc = pm.getObjectById(id);
    copy = pm.detachCopy(pc);
    pm.currentTransaction().commit();
} finally {
    pm.close();
}
//read or change the detached object here
System.out.println(copy.getName());

- Use detachAllOnCommit.
```java
Object pc = null;
try {
    PersistenceManager pm = pmf.getPersistenceManager();
    pm.setDetachAllOnCommit(true);
    pm.currentTransaction().begin();
    //retrieve in some way the object, query, getObjectById, etc
    pc = pm.getObjectById(id);
    pm.currentTransaction().commit(); // Object "pc" is now detached
} finally {
    pm.close();
} //read or change the detached object here
System.out.println(pc.getName());
```

The most expensive in terms of performance is the `detachCopy` because it makes copies of persistent objects. The advantage of detachment (via `detachCopy` or `detachAllOnCommit`) is that changes made outside the transaction can be further used to update the database in a new transaction. The other methods also allow changes outside of the transaction, but the changed instances can't be used to update the database.

With `RetainValues=true` and `makeTransient` no object copies are made and the object values are set down in instances when the PersistenceManager disassociates them. Both methods are equivalent in performance, however the `makeTransient` method will set the values of the object during the instant the `makeTransient` method is invoked, and the `RetainValues=true` will set values of the object during commit.

The bottom line is to not use detachment if instances will only be used to read values.

### 107.1.13 Queries usage

Make sure you close all query results after you have finished with them. Failure to do so will result in significant memory leaks in your application.

### 107.1.14 Fetch Control

When fetching objects you have control over what gets fetched. This can have an impact if you are then detaching those objects. With JDO the default "maximum fetch depth" is 1.

### 107.1.15 Logging

I/O consumes a huge slice of the total processing time. Therefore it is recommended to reduce or disable logging in production. To disable the logging set the DataNucleus category to OFF in the Log4j configuration. See Logging for more information.

```
log4j.category.DataNucleus=OFF
```
107.2 General Comments on Overall Performance

In most applications, the performance of the persistence layer is very unlikely to be a bottleneck. More likely the design of the datastore itself, and in particular its indices are more likely to have the most impact, or alternatively network latency. That said, it is the DataNucleus projects' committed aim to provide the best performance possible, though we also want to provide functionality, so there is a compromise with respect to resource.

What is a benchmark? This is simply a series of persistence operations performing particular things e.g persist *n* objects, or retrieve *n* objects. If those operations are representative of your application then the benchmark is valid to you.

To find (or create) a benchmark appropriate to your project you need to determine the typical persistence operations that your application will perform. Are you interested in persisting 100 objects at once, or 1 million, for example? Then when you have a benchmark appropriate for that operation, compare the persistence solutions.

The performance tuning guide above gives a good oversight of tuning capabilities, and also refer to the following blog entry for our take on performance of DataNucleus AccessPlatform. And then the later blog entry about how to tune for bulk operations

107.2.1.1 GeeCon JPA provider comparison (Jun 2012)

There is an interesting presentation on JPA provider performance that was presented at GeeCon 2012 by Patrycja Wegrzynowicz. This presentation takes the time to look at what operations the persistence provider is performing, and does more than just "persist large number of flat objects into a single table", and so gives you something more interesting to analyse. DataNucleus comes out pretty well in many situations. You can also see the PDF here.

107.2.1.2 PolePosition (Dec 2008)

The PolePosition benchmark is a project on SourceForge to provide a benchmark of the write, read and delete of different data structures using the various persistence tools on the market. JPOX was run against this benchmark just before being renamed as DataNucleus and the results are found in the DataNucleus Wiki. The input data used for that benchmark run is found in JPOX SVN. Some comments on the PolePos benchmark :-

- It is essential that tests for such as Hibernate and DataNucleus performance comparable things. Some of the original tests had the "delete" simply doing a "DELETE FROM TBL" for Hibernate yet doing an Extent followed by delete each object individually for a JDO implementation. This is an unfair comparison and in the source tree in JPOX SVN this is corrected. This fix was pointed out to the PolePos SourceForge project but is not, as yet, fixed
- It is essential that schema is generated before the test, otherwise the test is no longer a benchmark of just a persistence operation. The source tree in JPOX SVN assumes the schema exists. This fix was pointed out to the PolePos SourceForge project but is not, as yet, fixed
- Each persistence implementation should have its own tuning options, and be able to add things like discriminators since that is what would happen in a real application. The source tree in JPOX SVN does this for JPOX running. Similarly a JDO implementation would tune the fetch groups being used - this is not present in the SourceForge project but is in JPOX SVN.
- DataNucleus performance is considered to be significantly improved over JPOX particularly due to batched inserts, and due to a rewritten query implementation that does enhanced fetching.
108 Monitoring

108.1 JDO : Monitoring

DataNucleus allows a user to enable various MBeans internally. These can then be used for monitoring the number of datastore calls etc.

108.1.1 Via API

The simplest way to monitor DataNucleus is to use its API for monitoring. Internally there are several MBeans (as used by JMX) and you can navigate to these to get the required information. To enable this set the persistence property `datanucleus.enableStatistics` to `true`. There are then two sets of statistics; one for the PMF and one for each PM. You access these as follows

```java
JDOPersistenceManagerFactory dnpmf = (JDOPersistenceManagerFactory)pmf;
FactoryStatistics stats = dnpmf.getNucleusContext().getStatistics();
... (access the statistics information)

JDOPersistenceManager dnpm = (JDOPersistenceManager)pm;
ManagerStatistics stats = dnpm.getExecutionContext().getStatistics();
... (access the statistics information)
```

108.1.2 Using JMX

The MBeans used by DataNucleus can be accessed via JMX at runtime. More about JMX here.

An MBean server is bundled with Sun JRE since version 1.5, and you can easily activate DataNucleus MBeans registration by creating your PMF with the persistence property `datanucleus.jmxType` as `default`.

Additionally, setting a few system properties are necessary for configuring the Sun JMX implementation. The minimum properties required are the following:

- `com.sun.management.jmxremote`
- `com.sun.management.jmxremote.authenticate`
- `com.sun.management.jmxremote.ssl`
- `com.sun.management.jmxremote.port=<port number>`

Usage example:

```bash
java -cp TheClassPathInHere
   -Dcom.sun.management.jmxremote
   -Dcom.sun.management.jmxremote.authenticate=false
   -Dcom.sun.management.jmxremote.ssl=false
   -Dcom.sun.management.jmxremote.port=8001
   TheMainClassInHere
```

Once you start your application and DataNucleus is initialized you can browse DataNucleus MBeans using a tool called jconsole (jconsole is distributed with the Sun JDK) via the URL:
Note that the mode of usage is presented in this document as matter of example, and by no means we recommend to disable authentication and secured communication channels. Further details on the Sun JMX implementation and how to configure it properly can be found in here.

DataNucleus MBeans are registered in a MBean Server when DataNucleus is started up (e.g. upon JDO PMF instantiation). To see the full list of DataNucleus MBeans, refer to the javadocs.

To enable management using MX4J you must specify the persistence property `datanucleus.jmxType` as `mx4j` when creating the PMF, and have the `mx4j` and `mx4j-tools` jars in the CLASSPATH.
109 Maven with DataNucleus

109.1 DataNucleus JDO and Maven(2+)

Apache Maven is a project management and build tool that is quite common in organisations. Using DataNucleus and JDO with Maven is simple since the DataNucleus jars, JDO API jar and DataNucleus Maven plugin are present in the Maven central repository, so you don't need to define any repository to find the artifacts.

The only remaining thing to do is identify which artifacts are required for your project, updating your `pom.xml` accordingly.

```xml
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>javax.jdo</groupId>
      <artifactId>jdo-api</artifactId>
      <version>3.0.1</version>
    </dependency>
  </dependencies>
  ...
</project>
```

The only distinction to make here is that the above is for compile time since your persistence code (if implementation independent) will only depend on the basic persistence API. At runtime you will need the DataNucleus artifacts present also, so this becomes
<project>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>javax.jdo</groupId>
      <artifactId>jdo-api</artifactId>
      <version>3.0.1</version>
    </dependency>
    <dependency>
      <groupId>org.datanucleus</groupId>
      <artifactId>datanucleus-core</artifactId>
      <version>[3.2.0, 3.2.99)</version>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>org.datanucleus</groupId>
      <artifactId>datanucleus-api-jdo</artifactId>
      <version>[3.2.0, 3.2.99)</version>
    </dependency>
    <dependency>
      <groupId>org.datanucleus</groupId>
      <artifactId>datanucleus-rdbms</artifactId>
      <version>[3.2.0, 3.2.99)</version>
      <scope>runtime</scope>
    </dependency>
    ...
  </dependencies>
  ...
</project>

Obviously replace the datanucleus-rdbms jar with the jar for whichever datastore you are using. If running your app using Maven "exec" plugin then the runtime specification may not be needed.

Please note that you can alternatively use the convenience artifact for JDO+RDBMS (or JDO+ whichever datastore you're using).

<project>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>org.datanucleus</groupId>
      <artifactId>datanucleus-accessplatform-jdo-rdbms</artifactId>
      <version>3.3.0-release</version>
      <type>pom</type>
    </dependency>
    ...
  </dependencies>
  ...
</project>
109.1.1 Maven2 Plugin : Enhancement and SchemaTool

Now that you have the DataNucleus jars available to you, via the repositories, you want to perform DataNucleus operations. The primary operations are enhancement and SchemaTool. If you want to use the DataNucleus Maven plugin for enhancement or SchemaTool add the following to your `pom.xml`:

```xml
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.datanucleus</groupId>
        <artifactId>datanucleus-maven-plugin</artifactId>
        <version>3.3.0-release</version>
        <configuration>
          <api>JDO</api>
          <props>${basedir}/datanucleus.properties</props>
          <log4jConfiguration>${basedir}/log4j.properties</log4jConfiguration>
          <verbose>true</verbose>
        </configuration>
        <executions>
          <execution>
            <phase>process-classes</phase>
            <goals>
              <goal>enhance</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Note that this plugin step will automatically try to bring in the latest applicable version of `datanucleus-core` for use by the enhancer. It does this since you don't need to have `datanucleus-core` in your POM for compilation/enhancement. If you want to use an earlier version then you need to add exclusions to the `maven-datanucleus-plugin`.

The `executions` part of that will make enhancement be performed immediately after compile, so automatic. See also the Enhancer docs.

To run the enhancer manually you do:

```bash
mvn datanucleus:enhance
```

**DataNucleus SchemaTool** is achieved similarly, via:

```bash
mvn datanucleus:schema-create
```
110 Eclipse with DataNucleus

110.1 DataNucleus JDO and Eclipse

Eclipse provides a powerful development environment for Java systems. DataNucleus provides its own plugin for use within Eclipse, giving access to many features of DataNucleus from the convenience of your development environment.

- Installation
- General Preferences
- Preferences : Enhancer
- Preferences : SchemaTool
- Enable DataNucleus Support
- Generate JDO MetaData
- Generate persistence.xml
- Run the Enhancer
- Run SchemaTool

110.1.1 Plugin Installation

The DataNucleus plugin requires Eclipse 3.1 or above. To obtain and install the DataNucleus Eclipse plugin select Help -> Software Updates -> Find and Install On the panel that pops up select Search for new features to install Select New Remote Site, and in that new window set the URL as http://www.datanucleus.org/downloads/eclipse-update/ and the name as DataNucleus. Now select the site it has added "DataNucleus", and click "Finish". This will then find the releases of the DataNucleus plugin. Select the latest version of the DataNucleus Eclipse plugin. Eclipse then downloads and installs the plugin. Easy!

110.1.2 Plugin configuration

The DataNucleus Eclipse plugin allows saving of preferences so that you get nice defaults for all subsequent usage. You can set the preferences at two levels:

- **Globally for the Plugin**: Go to Window -> Preferences -> DataNucleus Eclipse Plugin and see the options below that
- **For a Project**: Go to [your project] -> Properties -> DataNucleus Eclipse Plugin and select "Enable project-specific properties"

110.1.3 Plugin configuration - General

Firstly open the main plugin preferences page, set the API to be used, and configure the libraries needed by DataNucleus. These are in addition to whatever you already have in your projects CLASSPATH, but to run the DataNucleus Enhancer/SchemaTool you will require the following

- jdo-api.jar
- datanucleus-core
- datanucleus-api-jdo
- datanucleus-rdbms : for running SchemaTool
- Datastore driver jar (e.g JDBC) : for running SchemaTool
Below this you can set the location of a configuration file for Log4j to use. This is useful when you want to debug the Enhancer/SchemaTool operations.

110.1.4 Plugin configuration - Enhancer

Open the "Enhancer" page. You have the following settings:

- **Input file extensions**: the enhancer accepts input defining the classes to be enhanced. This is typically performed by passing in the JDO XML MetaData files. When you use annotations you need to pass in class files. So you select the suffices you need.
- **Verbose**: selecting this means you get much more output from the enhancer.
- **PersistenceUnit**: Name of the persistence unit if enhancing a persistence-unit.
110.1.5 Plugin configuration - SchemaTool

Open the "SchemaTool" page. You have the following settings:

- **Input file extensions**: SchemaTool accepts input defining the classes to have their schema generated. This is typically performed by passing in the JDO XML MetaData files. When you use annotations you need to pass in class files. So you select the suffixes you need.

- **Verbose**: selecting this means you get much more output from SchemaTool.

- **PersistenceUnit**: Name of the persistence unit if running SchemaTool on a persistence-unit.

- **Datastore details**: You can either specify the location of a properties file defining the location of your datastore, or you supply the driver name, URL, username and password.
110.1.6 Enabling DataNucleus support

First thing to note is that the DataNucleus plugin is for Eclipse "Java project"s only. After having configured the plugin you can now add DataNucleus support on your projects. Simply right-click on your project in Package Explorer and select DataNucleus->"Add DataNucleus Support" from the context menu.
110.1.7 Defining JDO XML Metadata

It is standard practice to define the MetaData for your persistable classes in the same package as these classes. You now define your MetaData, by right-click on a package in your project and select "Create JDO 2.0 Metadata File" from DataNucleus context menu. The dialog prompts for the file name to be used and creates a basic Metadata file for all classes in this package, which can now be adapted to your needs. You can also perform same steps as above on a *.java file, which will create the metadata for the selected file only. Please note that the wizard will overwrite existing files without further notice.

![JDO 2.0 Metadata File](image)

110.1.8 Defining 'persistence.xml'

You can also use the DataNucleus plugin to generate a "persistence.xml" file adding all classes into a single persistence-unit. You do this by right-clicking on your project, and selecting the option. The "persistence.xml" is generated under META-INF for the source folder. Please note that the wizard will overwrite existing files without further notice.

110.1.9 Enhancing the classes

The DataNucleus Eclipse plugin allows you to easily byte-code enhance your classes using the DataNucleus enhancer. Right-click on your project and select "Enable Auto-Enhancement" from the DataNucleus context menu. Now that you have the enhancer set up you can enable enhancement of your classes. The DataNucleus Eclipse plugin currently works by enabling/disabling automatic enhancement as a follow on process for the Eclipse build step. This means that when you enable it, every time Eclipse builds your classes it will then enhance the classes defined by the available "jdo" MetaData files. Thereafter every time that you build your classes the JDO enabled ones will be enhanced. Easy! Messages from the enhancement process will be written to the Eclipse Console. Make sure that you have your Java files in a source folder, and that the binary class files are written elsewhere If everything is set-up right, you should see the output below.
110.1.10 Generating your database schema

Once your classes have been enhanced you are in a position to create the database schema (assuming you will be using a new schema - omit this step if you already have your schema). Click on the project under "Package Explorer" and under "DataNucleus" there is an option "Run SchemaTool". This brings up a panel to define your database location (URL, login, password etc). You enter these details and the schema will be generated.
Messages from the SchemaTool process will be written to the Eclipse Console.
111 Tutorial with RDBMS

111.1 DataNucleus - Tutorial for JDO using RDBMS

111.1.1 Background
An application can be JDO-enabled via many routes depending on the development process of the project in question. For example the project could use Eclipse as the IDE for developing classes. In that case the project would typically use the DataNucleus Eclipse plugin. Alternatively the project could use Ant, Maven or some other build tool. In this case this tutorial should be used as a guiding way for using DataNucleus in the application. The JDO process is quite straightforward.

1. Prerequisite: Download DataNucleus AccessPlatform
2. Step 1: Define their persistence definition using Meta-Data.
3. Step 2: Define the "persistence-unit"
4. Step 3: Compile your classes, and instrument them (using the DataNucleus enhancer).
5. Step 4: Write your code to persist your objects within the DAO layer.
6. Step 5: Run your application.

The tutorial guides you through this. You can obtain the code referenced in this tutorial from SourceForge (one of the files entitled "datanucleus-samples-jdo-tutorial-*").

111.1.2 Prerequisite: Download DataNucleus AccessPlatform
You can download DataNucleus in many ways, but the simplest is to download the distribution zip appropriate to your datastore. You can do this from SourceForge DataNucleus download page. When you open the zip you will find DataNucleus jars in the lib directory, and dependency jars in the deps directory.

111.1.3 Step 1: Take your model classes and mark which are persistable
For our tutorial, say we have the following classes representing a store of products for sale.
package org.datanucleus.samples.jdo.tutorial;

public class Inventory {
    String name = null;
    Set<Product> products = new HashSet();

    public Inventory(String name) {
        this.name = name;
    }

    public Set<Product> getProducts() { return products; }
}

package org.datanucleus.samples.jdo.tutorial;

public class Product {
    long id;
    String name = null;
    String description = null;
    double price = 0.0;

    public Product(String name, String desc, double price) {
        this.name = name;
        this.description = desc;
        this.price = price;
    }
}

package org.datanucleus.samples.jdo.tutorial;

public class Book extends Product {
    String author=null;
    String isbn=null;
    String publisher=null;

    public Book(String name, String desc, double price, String author, String isbn, String publisher) {
        super(name,desc,price);
        this.author = author;
        this.isbn = isbn;
        this.publisher = publisher;
    }
}
So we have a relationship (Inventory having a set of Products), and inheritance (Product-Book). Now we need to be able to persist objects of all of these types, so we need to define persistence for them. There are many things that you can define when deciding how to persist objects of a type but the essential parts are

- Mark the class as PersistenceCapable so it is visible to the persistence mechanism
- Identify which field(s) represent the identity of the object (or use datastore-identity if no field meets this requirement).

So this is what we do now. Note that we could define persistence using XML metadata, annotations or via the JDO API. In this tutorial we will use annotations.

```java
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Inventory
{
    @PrimaryKey
    String name = null;
    ...
}
```

```java
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Product
{
    @PrimaryKey
    @Persistent(valueStrategy=IdGeneratorStrategy.INCREMENT)
    long id;
    ...
}
```

```java
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Book extends Product
{
    ...
}
```

Note that we mark each class that can be persisted with @PersistenceCapable and their primary key field(s) with @PrimaryKey. In addition we defined a valueStrategy for Product field id so that it will have its values generated automatically. In this tutorial we are using application identity which means that all objects of these classes will have their identity defined by the primary key field(s). You can read more in datastore identity and application identity when designing your systems persistence.
111.1.4 Step 2: Define the 'persistence-unit'

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted. You do this via a file `META-INF/persistence.xml` at the root of the CLASSPATH. Like this

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">
    <!-- JDO tutorial "unit" -->
    <persistence-unit name="Tutorial">
        <class>org.datanucleus.samples.jdo.tutorial.Inventory</class>
        <class>org.datanucleus.samples.jdo.tutorial.Product</class>
        <class>org.datanucleus.samples.jdo.tutorial.Book</class>
        <exclude-unlisted-classes/>
        <properties>
            <property name="javax.jdo.option.ConnectionURL" value="jdbc:hsqldb:mem:datanucleus"/>
            <property name="javax.jdo.option.ConnectionDriverName" value="org.hsqldb.jdbcDriver"/>
            <property name="javax.jdo.option.ConnectionUserName" value="sa"/>
            <property name="javax.jdo.option.ConnectionPassword" value=""/>
            <property name="datanucleus.autoCreateSchema" value="true"/>
            <property name="datanucleus.validateTables" value="false"/>
            <property name="datanucleus.validateConstraints" value="false"/>
        </properties>
    </persistence-unit>
</persistence>
```

Note that you could equally use a properties file to define the persistence with JDO, but in this tutorial we use `persistence.xml` for convenience.

111.1.5 Step 3: Enhance your classes

JDO relies on the classes that you want to persist implementing `PersistenceCapable`. You could write your classes manually to do this but this would be laborious. Alternatively you can use a post-processing step to compilation that "enhances" your compiled classes, adding on the necessary extra methods to make them `PersistenceCapable`. There are several ways to do this, most notably at post-compile, or at runtime. We use the post-compile step in this tutorial. DataNucleus JDO provides its own byte-code enhancer for instrumenting/enhancing your classes (in `datanucleus-core`) and this is included in the DataNucleus AccessPlatform zip file prerequisite.

To understand on how to invoke the enhancer you need to visualise where the various source and jdo files are stored.
The first thing to do is compile your domain/model classes. You can do this in any way you wish, but the downloadable JAR provides an Ant task, and a Maven2 project to do this for you.

Using Ant:
```
ant compile
```

Using Maven2:
```
mvn compile
```

To enhance classes using the DataNucleus Enhancer, you need to invoke a command something like this from the root of your project.

Using Ant:
```
ant enhance
```

Using Maven: (this is usually done automatically after the "compile" goal)
```
mvn datanucleus:enhance
```

Manually on Linux/Unix:
```
java -cp target/classes:lib/datanucleus-core.jar:lib/datanucleus-api-jdo.jar:
   lib/jdo-api.jar
   org.datanucleus.enhancer.DataNucleusEnhancer -pu Tutorial
```

Manually on Windows:
```
java -cp target\classes;lib\datanucleus-core.jar;lib\datanucleus-api-jdo.jar;
   lib\jdo-api.jar
   org.datanucleus.enhancer.DataNucleusEnhancer -pu Tutorial
```

This command enhances the .class files that have @PersistenceCapable annotations. If you accidentally omitted this step, at the point of running your application and trying to persist an object, you would get a `ClassNotFoundException` thrown. The use of the enhancer is documented in more detail in the Enhancer Guide. The output of this step are a set of class files that represent `PersistenceCapable` classes.
111.1.6 Step 4 : Write the code to persist objects of your classes

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted, and when. Interaction with the persistence framework of JDO is performed via a PersistenceManager. This provides methods for persisting of objects, removal of objects, querying for persisted objects, etc. This section gives examples of typical scenarios encountered in an application.

The initial step is to obtain access to a PersistenceManager, which you do as follows

```
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory("Tutorial");
PersistenceManager pm = pmf.getPersistenceManager();
```

Now that the application has a PersistenceManager it can persist objects. This is performed as follows

```
Transaction tx=pm.currentTransaction();
try
{
    tx.begin();
    Inventory inv = new Inventory("My Inventory");
    Product product = new Product("Sony Discman", "A standard discman from Sony", 49.99);
    inv.getProducts().add(product);
    pm.makePersistent(inv);
    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
    pm.close();
}
```

Note the following

- We have persisted the `Inventory` but since this referenced the `Product` then that is also persisted.
- The `finally` step is important to tidy up any connection to the datastore, and close the PersistenceManager

If you want to retrieve an object from persistent storage, something like this will give what you need. This uses a "Query", and retrieves all Product objects that have a price below 150.00, ordering them in ascending price order.
Transaction tx = pm.currentTransaction();
try {
    tx.begin();

    Query q = pm.newQuery("SELECT * FROM " + Product.class.getName() + " WHERE price < 150.00 ORDER BY price ASC");
    List<Product> products = (List<Product>)q.execute();
    Iterator<Product> iter = products.iterator();
    while (iter.hasNext()) {
        Product p = iter.next();

        ... (use the retrieved objects)
    }

    tx.commit();
}
finally {
    if (tx.isActive()) {
        tx.rollback();
    }

    pm.close();
}

If you want to delete an object from persistence, you would perform an operation something like

Transaction tx = pm.currentTransaction();
try {
    tx.begin();

    ... (retrieval of objects etc)

    pm.deletePersistent(product);

    tx.commit();
}
finally {
    if (tx.isActive()) {
        tx.rollback();
    }

    pm.close();
}
Clearly you can perform a large range of operations on objects. We can’t hope to show all of these here. Any good JDO book will provide many examples.

111.1.7 Step 5 : Run your application

To run your JDO-enabled application will require a few things to be available in the Java CLASSPATH, these being

- Any persistence.xml file for the PersistenceManagerFactory creation
- Any JDO XML MetaData files for your persistable classes (not used in this example)
- Any JDBC driver classes needed for accessing your datastore
- The JDO API JAR (defining the JDO interface)
- The DataNucleus Core, DataNucleus JDO API and DataNucleus RDBMS JARs

After that it is simply a question of starting your application and all should be taken care of. You can access the DataNucleus Log file by specifying the logging configuration properties, and any messages from DataNucleus will be output in the normal way. The DataNucleus log is a very powerful way of finding problems since it can list all SQL actually sent to the datastore as well as many other parts of the persistence process.
Using Ant (you need the included "persistence.xml" to specify your database)
ant run

Using Maven:
mvn exec:java

Manually on Linux/Unix:
java -cp lib/jdo-api.jar:lib/datanucleus-core.jar:lib/datanucleus-rdbms.jar:
   lib/datanucleus-api-jdo.jar:lib/{jdbc-driver}.jar:target/classes/:
   org.datanucleus.samples.jdo.tutorial.Main

Manually on Windows:
java -cp lib\jdo-api.jar;lib\datanucleus-core.jar;lib\datanucleus-rdbms.jar;
   lib\datanucleus-api-jdo.jar;lib\{jdbc-driver}.jar;target\classes\;
   org.datanucleus.samples.jdo.tutorial.Main

Output:

DataNucleus Tutorial
====================
Persisting products
Product and Book have been persisted

Retrieving Extent for Products
> Product : Sony Discman [A standard discman from Sony]
> Book : JRR Tolkien - Lord of the Rings by Tolkien

Executing Query for Products with price below 150.00
> Book : JRR Tolkien - Lord of the Rings by Tolkien

Deleting all products from persistence
Deleted 2 products

End of Tutorial

111.2 Part 2 : Next steps

In the above simple tutorial we showed how to employ JDO and persist objects to an RDBMS. Obviously this just scratches the surface of what you can do, and to use JDO requires minimal work from the user. In this second part we show some further things that you are likely to want to do.

1. Step 6 : Controlling the schema.
2. Step 7 : Generate the database tables where your classes are to be persisted using SchemaTool.
111.2.1 Step 6: Controlling the schema

In the above simple tutorial we didn't look at controlling the schema generated for these classes. Now let's pay more attention to this part by defining XML Metadata for the schema.

```xml
<?xml version="1.0"?>
<!DOCTYPE orm PUBLIC
 "-//Sun Microsystems, Inc./DTD Java Data Objects Metadata 2.0//EN"
 "http://java.sun.com/dtd/orm_2_0.dtd">
<orm>
  <package name="org.datanucleus.samples.jdo.tutorial">
    <class name="Inventory" identity-type="datastore" table="INVENTORIES">
      <inheritance strategy="new-table"/>
      <field name="name">
        <column name="INVENTORY_NAME" length="100" jdbc-type="VARCHAR"/>
      </field>
      <field name="products">
        <join/>
      </field>
    </class>
    <class name="Product" identity-type="datastore" table="PRODUCTS">
      <inheritance strategy="new-table"/>
      <field name="name">
        <column name="PRODUCT_NAME" length="100" jdbc-type="VARCHAR"/>
      </field>
      <field name="description">
        <column length="255" jdbc-type="VARCHAR"/>
      </field>
    </class>
    <class name="Book" identity-type="datastore" table="BOOKS">
      <inheritance strategy="new-table"/>
      <field name="isbn">
        <column length="20" jdbc-type="VARCHAR"/>
      </field>
      <field name="author">
        <column length="40" jdbc-type="VARCHAR"/>
      </field>
      <field name="publisher">
        <column length="40" jdbc-type="VARCHAR"/>
      </field>
    </class>
  </package>
</orm>
```

With JDO you have various options as far as where this XML MetaData files is placed in the file structure, and whether they refer to a single class, or multiple classes in a package. With the above example, we have both classes specified in the same file `package-hsql.orm`, in the package these classes are in, since we want to persist to HSQL.
111.2.2 Step 7 : Generate any schema required for your domain classes

This step is optional, depending on whether you have an existing database schema. If you haven’t, at this point you can use the SchemaTool to generate the tables where these domain objects will be persisted. DataNucleus SchemaTool is a command line utility (it can be invoked from Maven2/Ant in a similar way to how the Enhancer is invoked). The first thing that you need is to update the persistence.xml file with your database details. Here we have a sample file (for HSQLDB)

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">
    <!-- Tutorial "unit" -->
    <persistence-unit name="Tutorial">
        <class>org.datanucleus.samples.jdo.tutorial.Inventory</class>
        <class>org.datanucleus.samples.jdo.tutorial.Product</class>
        <class>org.datanucleus.samples.jdo.tutorial.Book</class>
        <exclude-unlisted-classes/>
        <properties>
            <property name="javax.jdo.option.ConnectionURL" value="jdbc:hsqldb:mem:datanucleus"/>
            <property name="javax.jdo.option.ConnectionDriverName" value="org.hsqldb.jdbcDriver"/>
            <property name="javax.jdo.option.ConnectionUserName" value="sa"/>
            <property name="javax.jdo.option.ConnectionPassword" value=""/>
            <property name="datanucleus.autoCreateSchema" value="true"/>
            <property name="datanucleus.validateTables" value="false"/>
            <property name="datanucleus.validateConstraints" value="false"/>
        </properties>
    </persistence-unit>
</persistence>
```

Now we need to run DataNucleus SchemaTool. For our case above you would do something like this
Using Ant:
ant createschema

Using Maven2:
mvn datanucleus:schema-create

Manually on Linux/Unix:
java -cp target/classes;lib/datanucleus-core.jar;lib/datanucleus-rdbms.jar;
   lib/datanucleus-jdo-api.jar;lib/jdo-api.jar;lib/{jdbc_driver.jar}
   org.datanucleus.store.schema.SchemaTool
   -create -pu Tutorial

Manually on Windows:
java -cp target\classes;lib/datanucleus-core.jar;lib/datanucleus-rdbms.jar;
   lib/datanucleus-api-jdo.jar;lib\jdo-api.jar;lib\{jdbc_driver.jar}
   org.datanucleus.store.schema.SchemaTool
   -create -pu Tutorial

This will generate the required tables, indexes, and foreign keys for the classes defined in the JDO Meta-Data file. The generated schema in this case will be as follows

<table>
<thead>
<tr>
<th>INVENTORIES</th>
<th>INVENTORY_PRODUCTS</th>
<th>PRODUCTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>+INVENTORY_NAME</td>
<td>+INVENTORY_NAME_OID</td>
<td>+PRODUCT_ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PRODUCT_NAME</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DESCRIPTION</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PRICE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BOOKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>+BOOK_ID</td>
</tr>
<tr>
<td>AUTHOR</td>
</tr>
<tr>
<td>ISBN</td>
</tr>
<tr>
<td>PUBLISHER</td>
</tr>
</tbody>
</table>

111.2.3 Any questions?

If you have any questions about this tutorial and how to develop applications for use with DataNucleus please read the online documentation since answers are to be found there. If you don’t find what you’re looking for go to our Forums.

The DataNucleus Team
112 Tutorial with ODF

112.1 DataNucleus - Tutorial for JDO using ODF

Download

Source Code (GitHub)

112.1.1 Background
An application can be JDO-enabled via many routes depending on the development process of the project in question. For example the project could use Eclipse as the IDE for developing classes. In that case the project would typically use the DataNucleus Eclipse plugin. Alternatively the project could use Ant, Maven or some other build tool. In this case this tutorial should be used as a guiding way for using DataNucleus in the application. The JDO process is quite straightforward.

1. **Prerequisite**: Download DataNucleus AccessPlatform
2. **Step 1**: Define their persistence definition using Meta-Data.
3. **Step 2**: Define the "persistence-unit"
4. **Step 3**: Compile your classes, and instrument them (using the DataNucleus enhancer).
5. **Step 4**: Write your code to persist your objects within the DAO layer.
6. **Step 5**: Run your application.

The tutorial guides you through this. You can obtain the code referenced in this tutorial from SourceForge (one of the files entitled "datanucleus-samples-jdo-tutorial-*").

112.1.2 Prerequisite : Download DataNucleus AccessPlatform

You can download DataNucleus in many ways, but the simplest is to download the distribution zip appropriate to your datastore (ODF in this case). You can do this from SourceForge DataNucleus download page. When you open the zip you will find DataNucleus jars in the lib directory, and dependency jars in the deps directory.

112.1.3 Step 1 : Take your model classes and mark which are persistable

For our tutorial, say we have the following classes representing a store of products for sale.
package org.datanucleus.samples.jdo.tutorial;

public class Inventory
{
    String name = null;
    Set<Product> products = new HashSet();

    public Inventory(String name)
    {
        this.name = name;
    }

    public Set<Product> getProducts() {return products;}
}

package org.datanucleus.samples.jdo.tutorial;

public class Product
{
    long id;
    String name = null;
    String description = null;
    double price = 0.0;

    public Product(String name, String desc, double price)
    {
        this.name = name;
        this.description = desc;
        this.price = price;
    }
}

package org.datanucleus.samples.jdo.tutorial;

public class Book extends Product
{
    String author=null;
    String isbn=null;
    String publisher=null;

    public Book(String name, String desc, double price, String author,
    String isbn, String publisher)
    {
        super(name,desc,price);
        this.author = author;
        this.isbn = isbn;
        this.publisher = publisher;
    }
}
So we have a relationship (Inventory having a set of Products), and inheritance (Product-Book). Now we need to be able to persist objects of all of these types, so we need to **define persistence for them**. There are many things that you can define when deciding how to persist objects of a type but the essential parts are

- Mark the class as **PersistenceCapable** so it is visible to the persistence mechanism
- Identify which field(s) represent the identity of the object (or use datastore-identity if no field meets this requirement).

So this is what we do now. Note that we could define persistence using XML metadata, annotations or via the JDO API. In this tutorial we will use annotations.

```java
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Inventory
{
    @PrimaryKey
    String name = null;

    ...
}
```

```java
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Product
{
    @PrimaryKey
    @Persistent(valueStrategy=IdGeneratorStrategy.INCREMENT)
    long id;

    ...
}
```

```java
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Book extends Product
{
    ...
}
```

Note that we mark each class that can be persisted with `@PersistenceCapable` and their primary key field(s) with `@PrimaryKey`. In addition we defined a `valueStrategy` for Product field `id` so that it will have its values generated automatically. In this tutorial we are using application identity which means that all objects of these classes will have their identity defined by the primary key field(s). You can read more in **datastore identity** and **application identity** when designing your systems persistence.
112.1.4 Step 2 : Define the 'persistence-unit'

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted. You do this via a file META-INF/persistence.xml at the root of the CLASSPATH. Like this

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                                  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">

<!-- JDO tutorial "unit" -->
<persistence-unit name="Tutorial">
    <class>org.datanucleus.samples.jdo.tutorial.Inventory</class>
    <class>org.datanucleus.samples.jdo.tutorial.Product</class>
    <class>org.datanucleus.samples.jdo.tutorial.Book</class>
    <exclude-unlisted-classes/>
    <properties>
        <property name="javax.jdo.option.ConnectionURL" value="odf:file:test.ods"/>
        <property name="datanucleus.autoCreateSchema" value="true"/>
        <property name="datanucleus.validateTables" value="false"/>
        <property name="datanucleus.validateConstraints" value="false"/>
    </properties>
</persistence-unit>
</persistence>
```

Note that you could equally use a properties file to define the persistence with JDO, but in this tutorial we use persistence.xml for convenience.

112.1.5 Step 3 : Enhance your classes

JDO relies on the classes that you want to persist implementing PersistenceCapable. You could write your classes manually to do this but this would be laborious. Alternatively you can use a post-processing step to compilation that “enhances” your compiled classes, adding on the necessary extra methods to make them PersistenceCapable. There are several ways to do this, most notably at post-compile, or at runtime. We use the post-compile step in this tutorial. DataNucleus JDO provides its own byte-code enhancer for instrumenting/enhancing your classes (in datanucleus-core) and this is included in the DataNucleus AccessPlatform zip file prerequisite.

To understand on how to invoke the enhancer you need to visualise where the various source and jdo files are stored
The first thing to do is compile your domain/model classes. You can do this in any way you wish, but the downloadable JAR provides an Ant task, and a Maven2 project to do this for you.

Using Ant:
```bash
ant compile
```

Using Maven2:
```bash
mvn compile
```

To enhance classes using the DataNucleus Enhancer, you need to invoke a command something like this from the root of your project.

Using Ant:
```bash
ant enhance
```

Using Maven: (this is usually done automatically after the "compile" goal)
```bash
mvn datanucleus:enhance
```

Manually on Linux/Unix:
```bash
org.datanucleus.enhancer.DataNucleusEnhancer -pu Tutorial
```

Manually on Windows:
```bash
java -cp target\classes;lib\datanucleus-core.jar;lib\datanucleus-api-jdo.jar;lib\jdo-api.jar
org.datanucleus.enhancer.DataNucleusEnhancer -pu Tutorial
```

This command enhances the .class files that have @PersistenceCapable annotations. If you accidentally omitted this step, at the point of running your application and trying to persist an object, you would get a `ClassNotPersistenceCapableException` thrown. The use of the enhancer is documented in more detail in the Enhancer Guide. The output of this step are a set of class files that represent PersistenceCapable classes.
112.1.6 Step 4: Write the code to persist objects of your classes

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted, and when. Interaction with the persistence framework of JDO is performed via a PersistenceManager. This provides methods for persisting of objects, removal of objects, querying for persisted objects, etc. This section gives examples of typical scenarios encountered in an application.

The initial step is to obtain access to a PersistenceManager, which you do as follows:

```java
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory("Tutorial");
PersistenceManager pm = pmf.getPersistenceManager();
```

Now that the application has a PersistenceManager it can persist objects. This is performed as follows:

```java
Transaction tx=pm.currentTransaction();
try {
   tx.begin();
   Inventory inv = new Inventory("My Inventory");
   Product product = new Product("Sony Discman", "A standard discman from Sony", 49.99);
   inv.getProducts().add(product);
   pm.makePersistent(inv);
   tx.commit();
}
finally {
   if (tx.isActive()) {
      tx.rollback();
   }
   pm.close();
}
```

Note the following:

- We have persisted the `Inventory` but since this referenced the `Product` then that is also persisted.
- The `finally` step is important to tidy up any connection to the datastore, and close the PersistenceManager.

If you want to retrieve an object from persistent storage, something like this will give what you need. This uses a "Query", and retrieves all Product objects that have a price below 150.00, ordering them in ascending price order.
Transaction tx = pm.currentTransaction();
try {
    tx.begin();

    Query q = pm.newQuery("SELECT * + Product.class.getName() + "
                           " WHERE price < 150.00 ORDER BY price ASC");
    List<Product> products = (List<Product>)q.execute();
    Iterator<Product> iter = products.iterator();
    while (iter.hasNext()) {
        Product p = iter.next();

        // ... (use the retrieved objects)
    }
    tx.commit();
}
finally {
    if (tx.isActive())
        { tx.rollback();
    pm.close();
}

If you want to delete an object from persistence, you would perform an operation something like

Transaction tx = pm.currentTransaction();
try {
    tx.begin();

    // ... (retrieval of objects etc)

    pm.deletePersistent(product);

    tx.commit();
}
finally {
    if (tx.isActive())
        { tx.rollback();
    pm.close();
}

Clearly you can perform a large range of operations on objects. We can't hope to show all of these here. Any good JDO book will provide many examples.
112.1.7 Step 5: Run your application

To run your JDO-enabled application will require a few things to be available in the Java CLASSPATH, these being

- Any persistence.xml file for the PersistenceManagerFactory creation
- Any JDO XML MetaData files for your persistable classes (not used in this example)
- ODFDOM driver class(es) needed for accessing your datastore
- The JDO API JAR (defining the JDO interface)
- The DataNucleus Core, DataNucleus JDO API and DataNucleus ODF JARs

After that it is simply a question of starting your application and all should be taken care of. You can access the DataNucleus Log file by specifying the `logging` configuration properties, and any messages from DataNucleus will be output in the normal way. The DataNucleus log is a very powerful way of finding problems since it can list all SQL actually sent to the datastore as well as many other parts of the persistence process.
Using Ant (you need the included "persistence.xml" to specify your database)

ant run

Using Maven:

mvn exec:java

Manually on Linux/Unix:

```java
java -cp lib/jdo-api.jar:lib/datanucleus-core.jar:lib/datanucleus-odf.jar:
   lib/datanucleus-api-jdo.jar:lib/odfdom.jar:target/classes/:
   org.datanucleus.samples.jdo.tutorial.Main
```

Manually on Windows:

```java
java -cp lib\jdo-api.jar;lib\datanucleus-core.jar;lib\datanucleus-odf.jar;
   lib\datanucleus-api-jdo.jar;lib\odfdom.jar;target\classes\;
   org.datanucleus.samples.jdo.tutorial.Main
```

Output:

DataNucleus Tutorial

Persisting products

Product and Book have been persisted

Retrieving Extent for Products

> Product : Sony Discman [A standard discman from Sony]
> Book : JRR Tolkien - Lord of the Rings by Tolkien

Executing Query for Products with price below 150.00

> Book : JRR Tolkien - Lord of the Rings by Tolkien

Deleting all products from persistence

Deleted 2 products

End of Tutorial

112.2 Part 2 : Next steps

In the above simple tutorial we showed how to employ JDO and persist objects to ODF. Obviously this just scratches the surface of what you can do, and to use JDO requires minimal work from the user. In this second part we show some further things that you are likely to want to do.

1. **Step 6** : Controlling the schema.
2. **Step 7** : Generate the database tables where your classes are to be persisted using SchemaTool.
112.2.1 Step 6: Controlling the schema

In the above simple tutorial we didn’t look at controlling the schema generated for these classes. Now let’s pay more attention to this part by defining XML Metadata for the schema.

```xml
<?xml version="1.0"?>
<!DOCTYPE orm PUBLIC
"-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
"http://java.sun.com/dtd/orm_2_0.dtd">
<orm>
  <package name="org.datanucleus.samples.jdo.tutorial">
    <class name="Inventory" table="Inventories">
      <field name="name">
        <column name="Name" length="100"/>
      </field>
      <field name="products"/>
    </class>
    <class name="Product" table="Products">
      <extension vendor-name="datanucleus" key="include-column-headers" value="true"/>
      <inheritance strategy="complete-table"/>
      <field name="id">
        <column name="Id" position="0"/>
      </field>
      <field name="name">
        <column name="Name" position="1"/>
      </field>
      <field name="description">
        <column name="Description" position="2"/>
      </field>
      <field name="price">
        <column name="Price" position="3"/>
      </field>
    </class>
    <class name="Book" table="Books">
      <extension vendor-name="datanucleus" key="include-column-headers" value="true"/>
      <inheritance strategy="complete-table"/>
      <field name="Product.id">
        <column name="Id" position="0"/>
      </field>
      <field name="author">
        <column name="Author" position="4"/>
      </field>
      <field name="isbn">
        <column name="ISBN" position="5"/>
      </field>
      <field name="publisher">
        <column name="Publisher" position="6"/>
      </field>
    </class>
  </package>
</orm>
```
With JDO you have various options as far as where this XML MetaData files is placed in the file structure, and whether they refer to a single class, or multiple classes in a package. With the above example, we have both classes specified in the same file package-odf.orm, in the package these classes are in, since we want to persist to ODF.

112.2.2 Step 7 : Generate any schema required for your domain classes

This step is optional, depending on whether you have an existing database schema. If you haven't, at this point you can use the SchemaTool to generate the tables where these domain objects will be persisted. DataNucleus SchemaTool is a command line utility (it can be invoked from Maven2/Ant in a similar way to how the Enhancer is invoked). The first thing that you need is to update the persistence.xml file with your database details.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">

<!-- Tutorial "unit" -->
<persistence-unit name="Tutorial">
    <class>org.datanucleus.samples.jdo.tutorial.Inventory</class>
    <class>org.datanucleus.samples.jdo.tutorial.Product</class>
    <class>org.datanucleus.samples.jdo.tutorial.Book</class>
    <exclude-unlisted-classes/>
    <properties>
        <property name="javax.jdo.option.ConnectionURL" value="odf:file:test.ods"/>
        <property name="datanucleus.autoCreateSchema" value="true"/>
        <property name="datanucleus.validateTables" value="false"/>
        <property name="datanucleus.validateConstraints" value="false"/>
    </properties>
</persistence-unit>
</persistence>
```

Now we need to run DataNucleus SchemaTool. For our case above you would do something like this
Using Ant:

ant createschema

Using Maven2:

mvn datanucleus:schema-create

Manually on Linux/Unix:

```
java -cp target/classes:lib/datanucleus-core.jar:lib/datanucleus-odf.jar:
   lib/datanucleus-jdo-api.jar:lib/jdo-api.jar:lib/odfdom.jar
   org.datanucleus.store.schema.SchemaTool
   -create -pu Tutorial
```

Manually on Windows:

```
java -cp target\classes;lib\datanucleus-core.jar;lib\datanucleus-odf.jar;
   lib\datanucleus-api-jdo.jar;lib\jdo-api.jar;lib\odfdom.jar
   org.datanucleus.store.schema.SchemaTool
   -create -pu Tutorial
```

[Command shown on many lines to aid reading. Should be on single line]

This will generate the required tables, etc for the classes defined in the JDO Meta-Data file.

### 112.2.3 Any questions?

If you have any questions about this tutorial and how to develop applications for use with **DataNucleus** please read the online documentation since answers are to be found there. If you don't find what you're looking for go to our [Forums](#).

**The DataNucleus Team**
113 Tutorial with Excel

113.1 DataNucleus - Tutorial for JDO using Excel

Download
Source Code (GitHub)

113.1.1 Background
An application can be JDO-enabled via many routes depending on the development process of the project in question. For example the project could use Eclipse as the IDE for developing classes. In that case the project would typically use the DataNucleus Eclipse plugin. Alternatively the project could use Ant, Maven or some other build tool. In this case this tutorial should be used as a guiding way for using DataNucleus in the application. The JDO process is quite straightforward.

1. Prerequisite : Download DataNucleus AccessPlatform
2. Step 1 : Define their persistence definition using Meta-Data.
3. Step 2 : Define the "persistence-unit"
4. Step 3 : Compile your classes, and instrument them (using the DataNucleus enhancer).
5. Step 4 : Write your code to persist your objects within the DAO layer.
6. Step 5 : Run your application.

The tutorial guides you through this. You can obtain the code referenced in this tutorial from SourceForge (one of the files entitled "datanucleus-samples-jdo-tutorial-*").

113.1.2 Prerequisite : Download DataNucleus AccessPlatform
You can download DataNucleus in many ways, but the simplest is to download the distribution zip appropriate to your datastore (Excel in this case). You can do this from SourceForge DataNucleus download page. When you open the zip you will find DataNucleus jars in the lib directory, and dependency jars in the deps directory.

113.1.3 Step 1 : Take your model classes and mark which are persistable
For our tutorial, say we have the following classes representing a store of products for sale.
package org.datanucleus.samples.jdo.tutorial;

public class Inventory {
    String name = null;
    Set<Product> products = new HashSet();

    public Inventory(String name) {
        this.name = name;
    }

    public Set<Product> getProducts() { return products; }
}

package org.datanucleus.samples.jdo.tutorial;

public class Product {
    long id;
    String name = null;
    String description = null;
    double price = 0.0;

    public Product(String name, String desc, double price) {
        this.name = name;
        this.description = desc;
        this.price = price;
    }
}

package org.datanucleus.samples.jdo.tutorial;

public class Book extends Product {
    String author=null;
    String isbn=null;
    String publisher=null;

    public Book(String name, String desc, double price, String author, String isbn, String publisher) {
        super(name,desc,price);
        this.author = author;
        this.isbn = isbn;
        this.publisher = publisher;
    }
}
So we have a relationship (Inventory having a set of Products), and inheritance (Product-Book). Now we need to be able to persist objects of all of these types, so we need to define persistence for them. There are many things that you can define when deciding how to persist objects of a type but the essential parts are

- Mark the class as PersistenceCapable so it is visible to the persistence mechanism
- Identify which field(s) represent the identity of the object (or use datastore-identity if no field meets this requirement).

So this is what we do now. Note that we could define persistence using XML metadata, annotations or via the JDO API. In this tutorial we will use annotations.

```java
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Inventory {
    @PrimaryKey
    String name = null;
    ...
}
```

```java
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Product {
    @PrimaryKey
    @Persistent(valueStrategy=IdGeneratorStrategy.INCREMENT)
    long id;
    ...
}
```

```java
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Book extends Product {
    ...
}
```

Note that we mark each class that can be persisted with @PersistenceCapable and their primary key field(s) with @PrimaryKey. In addition we defined a valueStrategy for Product field id so that it will have its values generated automatically. In this tutorial we are using application identity which means that all objects of these classes will have their identity defined by the primary key field(s). You can read more in datastore identity and application identity when designing your systems persistence.
113.1.4 Step 2 : Define the 'persistence-unit'

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted. You do this via a file META-INF/persistence.xml at the root of the CLASSPATH. Like this

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">
    <!-- JDO tutorial "unit" -->
    <persistence-unit name="Tutorial">
        <class>org.datanucleus.samples.jdo.tutorial.Inventory</class>
        <class>org.datanucleus.samples.jdo.tutorial.Product</class>
        <class>org.datanucleus.samples.jdo.tutorial.Book</class>
        <exclude-unlisted-classes/>
        <properties>
            <property name="javax.jdo.option.ConnectionURL" value="excel:file:test.xml"/>
            <property name="datanucleus.autoCreateSchema" value="true"/>
            <property name="datanucleus.validateTables" value="false"/>
            <property name="datanucleus.validateConstraints" value="false"/>
        </properties>
    </persistence-unit>
</persistence>
```

Note that you could equally use a properties file to define the persistence with JDO, but in this tutorial we use persistence.xml for convenience.

113.1.5 Step 3 : Enhance your classes

JDO relies on the classes that you want to persist implementing PersistenceCapable. You could write your classes manually to do this but this would be laborious. Alternatively you can use a post-processing step to compilation that "enhances" your compiled classes, adding on the necessary extra methods to make them PersistenceCapable. There are several ways to do this, most notably at post-compile, or at runtime. We use the post-compile step in this tutorial. DataNucleus JDO provides its own byte-code enhancer for instrumenting/enhancing your classes (in datanucleus-core) and this is included in the DataNucleus AccessPlatform zip file prerequisite.

To understand on how to invoke the enhancer you need to visualise where the various source and jdo files are stored
The first thing to do is compile your domain/model classes. You can do this in any way you wish, but the downloadable JAR provides an Ant task, and a Maven2 project to do this for you.

Using Ant:
`ant compile`

Using Maven2:
`mvn compile`

To enhance classes using the DataNucleus Enhancer, you need to invoke a command something like this from the root of your project.

Using Ant:
`ant enhance`

Using Maven2: (this is usually done automatically after the "compile" goal)
`mvn datanucleus:enhance`

Manually on Linux/Unix:
`java -cp target/classes:lib/datanucleus-core.jar:`
`lib/datanucleus-api-jdo.jar:lib/jdo-api.jar`
`org.datanucleus.enhancer.DataNucleusEnhancer -pu Tutorial`

Manually on Windows:
`java -cp target\classes;lib\datanucleus-core.jar;`  
`lib\datanucleus-api-jdo.jar;lib\jdo-api.jar`  
`org.datanucleus.enhancer.DataNucleusEnhancer -pu Tutorial`

This command enhances the .class files that have @PersistenceCapable annotations. If you accidentally omitted this step, at the point of running your application and trying to persist an object, you would get a `ClassNotPersistenceCapableException` thrown. The use of the enhancer is documented in more detail in the Enhancer Guide. The output of this step are a set of class files that represent `PersistenceCapable` classes.
113.1.6 Step 4: Write the code to persist objects of your classes

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted, and when. Interaction with the persistence framework of JDO is performed via a PersistenceManager. This provides methods for persisting of objects, removal of objects, querying for persisted objects, etc. This section gives examples of typical scenarios encountered in an application.

The initial step is to obtain access to a PersistenceManager, which you do as follows:

```java
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory("Tutorial");
PersistenceManager pm = pmf.getPersistenceManager();
```

Now that the application has a PersistenceManager it can persist objects. This is performed as follows:

```java
Transaction tx=pm.currentTransaction();
try {
    tx.begin();
    Inventory inv = new Inventory("My Inventory");
    Product product = new Product("Sony Discman", "A standard discman from Sony", 49.99);
    inv.getProducts().add(product);
    pm.makePersistent(inv);
    tx.commit();
} finally {
    if (tx.isActive()) {
        tx.rollback();
    }
    pm.close();
}
```

Note the following:

- We have persisted the `Inventory` but since this referenced the `Product` then that is also persisted.
- The `finally` step is important to tidy up any connection to the datastore, and close the PersistenceManager.

If you want to retrieve an object from persistent storage, something like this will give what you need. This uses a "Query", and retrieves all Product objects that have a price below 150.00, ordering them in ascending price order.
Transaction tx = pm.currentTransaction();
try {
    tx.begin();
    Query q = pm.newQuery("SELECT * FROM " + Product.class.getName() + " WHERE price < 150.00 ORDER BY price ASC");
    List<Product> products = (List<Product>)q.execute();
    Iterator<Product> iter = products.iterator();
    while (iter.hasNext()) {
        Product p = iter.next();
        ...
    }
    tx.commit();
} finally {
    if (tx.isActive()) {
        tx.rollback();
    }
    pm.close();
}

If you want to delete an object from persistence, you would perform an operation something like

Transaction tx = pm.currentTransaction();
try {
    tx.begin();
    ...
    pm.deletePersistent(product);
    tx.commit();
} finally {
    if (tx.isActive()) {
        tx.rollback();
    }
    pm.close();
}

Clearly you can perform a large range of operations on objects. We can't hope to show all of these here. Any good JDO book will provide many examples.
113.1.7 Step 5 : Run your application

To run your JDO-enabled application will require a few things to be available in the Java CLASSPATH, these being

- Any persistence.xml file for the PersistenceManagerFactory creation
- Any JDO XML MetaData files for your persistable classes (not used in this example)
- Apache POI driver class(es) needed for accessing your datastore
- The JDO API JAR (defining the JDO interface)
- The DataNucleus Core, DataNucleus JDO API and DataNucleus Excel JARs

After that it is simply a question of starting your application and all should be taken care of. You can access the DataNucleus Log file by specifying the logging configuration properties, and any messages from DataNucleus will be output in the normal way. The DataNucleus log is a very powerful way of finding problems since it can list all SQL actually sent to the datastore as well as many other parts of the persistence process.
Using Ant (you need the included "persistence.xml" to specify your database)

ant run

Using Maven:
mvn exec:java

Manually on Linux/Unix:
java -cp lib/jdo-api.jar:lib/datanucleus-core.jar:lib/datanucleus-excel.jar:
  lib/datanucleus-api-jdo.jar:lib/poi.jar:target/classes/:
  org.datanucleus.samples.jdo.tutorial.Main

Manually on Windows:
java -cp lib\jdo-api.jar;lib\datanucleus-core.jar;lib\datanucleus-excel.jar;
  lib\datanucleus-api-jdo.jar;lib\poi.jar;target\classes\;
  org.datanucleus.samples.jdo.tutorial.Main

Output:

DataNucleus Tutorial

Persisting products
Product and Book have been persisted

Retrieving Extent for Products
> Product : Sony Discman [A standard discman from Sony]
> Book : JRR Tolkien - Lord of the Rings by Tolkien

Executing Query for Products with price below 150.00
> Book : JRR Tolkien - Lord of the Rings by Tolkien

Deleting all products from persistence
Deleted 2 products

End of Tutorial

113.2 Part 2 : Next steps

In the above simple tutorial we showed how to employ JDO and persist objects to Excel. Obviously this just scratches the surface of what you can do, and to use JDO requires minimal work from the user. In this second part we show some further things that you are likely to want to do.

1. Step 6 : Controlling the schema.
2. Step 7 : Generate the database tables where your classes are to be persisted using SchemaTool.
113.2.1 Step 6: Controlling the schema

In the above simple tutorial we didn’t look at controlling the schema generated for these classes. Now let’s pay more attention to this part by defining XML Metadata for the schema.

```xml
<?xml version="1.0"?>
<!DOCTYPE orm PUBLIC
 "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
 "http://java.sun.com/dtd/orm_2_0.dtd">
<orm>
  <package name="org.datanucleus.samples.jdo.tutorial">
    <class name="Inventory" table="Inventories">
      <field name="name">
        <column name="Name" length="100"/>
      </field>
      <field name="products"/>
    </class>

    <class name="Product" table="Products">
      <inheritance strategy="complete-table"/>
      <field name="id">
        <column name="Id" position="0"/>
      </field>
      <field name="name">
        <column name="Name" position="1"/>
      </field>
      <field name="description">
        <column name="Description" position="2"/>
      </field>
      <field name="price">
        <column name="Price" position="3"/>
      </field>
    </class>

    <class name="Book" table="Books">
      <inheritance strategy="complete-table"/>
      <field name="Product.id">
        <column name="Id" position="0"/>
      </field>
      <field name="author">
        <column name="Author" position="4"/>
      </field>
      <field name="isbn">
        <column name="ISBN" position="5"/>
      </field>
      <field name="publisher">
        <column name="Publisher" position="6"/>
      </field>
    </class>
  </package>
</orm>
```
With JDO you have various options as far as where this XML MetaData files is placed in the file structure, and whether they refer to a single class, or multiple classes in a package. With the above example, we have both classes specified in the same file `package-excel.orm`, in the package these classes are in, since we want to persist to Excel.

### 113.2.2 Step 7 : Generate any schema required for your domain classes

This step is optional, depending on whether you have an existing database schema. If you haven’t, at this point you can use the SchemaTool to generate the tables where these domain objects will be persisted. DataNucleus SchemaTool is a command line utility (it can be invoked from Maven2/Ant in a similar way to how the Enhancer is invoked). The first thing that you need is to update the `persistence.xml` file with your database details.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">
    <!-- Tutorial "unit" -->
    <persistence-unit name="Tutorial">
        <class>org.datanucleus.samples.jdo.tutorial.Inventory</class>
        <class>org.datanucleus.samples.jdo.tutorial.Product</class>
        <class>org.datanucleus.samples.jdo.tutorial.Book</class>
        <exclude-unlisted-classes/>
        <properties>
            <property name="javax.jdo.option.ConnectionURL" value="excel:file:test.xml"/>
            <property name="datanucleus.autoCreateSchema" value="true"/>
            <property name="datanucleus.validateTables" value="false"/>
            <property name="datanucleus.validateConstraints" value="false"/>
        </properties>
    </persistence-unit>
</persistence>
```

Now we need to run DataNucleus SchemaTool. For our case above you would do something like this
Using Ant:
`ant createschema`

Using Maven2:
`mvn datanucleus:schema-create`

Manually on Linux/Unix:

```
java -cp target/classes:lib/datanucleus-core.jar:lib/datanucleus-excel.jar:
    lib/datanucleus-jdo-api.jar:lib/jdo-api.jar:lib/poi.jar
    org.datanucleus.store.schema.SchemaTool
    -create -pu Tutorial
```

Manually on Windows:

```
javac -cp target\classes;lib\datanucleus-core.jar;lib\datanucleus-excel.jar;
    lib\datanucleus-api-jdo.jar;lib\jdo-api.jar;lib\poi.jar
    org.datanucleus.store.schema.SchemaTool
    -create -pu Tutorial
```

This will generate the required tables, etc for the classes defined in the JDO Meta-Data file.

### 113.2.3 Any questions?

If you have any questions about this tutorial and how to develop applications for use with DataNucleus please read the online documentation since answers are to be found there. If you don't find what you're looking for go to our Forums.

The DataNucleus Team
114 Tutorial with MongoDB

114.1 DataNucleus - Tutorial for JDO using MongoDB

114.1.1 Background
An application can be JDO-enabled via many routes depending on the development process of the project in question. For example the project could use Eclipse as the IDE for developing classes. In that case the project would typically use the DataNucleus Eclipse plugin. Alternatively the project could use Ant, Maven or some other build tool. In this case this tutorial should be used as a guiding way for using DataNucleus in the application. The JDO process is quite straightforward.

1. Prerequisite: Download DataNucleus AccessPlatform
2. Step 1: Define their persistence definition using Meta-Data.
3. Step 2: Define the "persistence-unit"
4. Step 3: Compile your classes, and instrument them (using the DataNucleus enhancer).
5. Step 4: Write your code to persist your objects within the DAO layer.
6. Step 5: Run your application.

The tutorial guides you through this. You can obtain the code referenced in this tutorial from SourceForge (one of the files entitled "datanucleus-samples-jdo-tutorial-*").

114.1.2 Prerequisite: Download DataNucleus AccessPlatform
You can download DataNucleus in many ways, but the simplest is to download the distribution zip appropriate to your datastore (MongoDB in this case). You can do this from SourceForge DataNucleus download page. When you open the zip you will find DataNucleus jars in the lib directory, and dependency jars in the deps directory.

114.1.3 Step 1: Take your model classes and mark which are persistable
For our tutorial, say we have the following classes representing a store of products for sale.
package org.datanucleus.samples.jdo.tutorial;

public class Inventory
{
    String name = null;
    Set<Product> products = new HashSet();

    public Inventory(String name)
    {
        this.name = name;
    }

    public Set<Product> getProducts() {return products;}
}

package org.datanucleus.samples.jdo.tutorial;

public class Product
{
    long id;
    String name = null;
    String description = null;
    double price = 0.0;

    public Product(String name, String desc, double price)
    {
        this.name = name;
        this.description = desc;
        this.price = price;
    }
}

package org.datanucleus.samples.jdo.tutorial;

public class Book extends Product
{
    String author=null;
    String isbn=null;
    String publisher=null;

    public Book(String name, String desc, double price, String author,
                 String isbn, String publisher)
    {
        super(name,desc,price);
        this.author = author;
        this.isbn = isbn;
        this.publisher = publisher;
    }
}
So we have a relationship (Inventory having a set of Products), and inheritance (Product-Book). Now we need to be able to persist objects of all of these types, so we need to **define persistence for them**. There are many things that you can define when deciding how to persist objects of a type but the essential parts are

- Mark the class as *PersistenceCapable* so it is visible to the persistence mechanism
- Identify which field(s) represent the identity of the object (or use datastore-identity if no field meets this requirement).

So this is what we do now. Note that we could define persistence using XML metadata, annotations or via the JDO API. In this tutorial we will use annotations.

```java
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Inventory {
    @PrimaryKey
    String name = null;
    ...
}
```

```java
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Product {
    @PrimaryKey
    @Persistent(valueStrategy=IdGeneratorStrategy.INCREMENT)
    long id;
    ...
}
```

```java
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Book extends Product {
    ...
}
```

Note that we mark each class that can be persisted with [@PersistenceCapable](#) and their primary key field(s) with [@PrimaryKey](#). In addition we defined a `valueStrategy` for Product field `id` so that it will have its values generated automatically. In this tutorial we are using application identity which means that all objects of these classes will have their identity defined by the primary key field(s). You can read more in [datastore identity](#) and [application identity](#) when designing your systems persistence.
114.1.4 Step 2: Define the 'persistence-unit'

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted. You do this via a file META-INF/persistence.xml at the root of the CLASSPATH. Like this

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">
    <!-- JDO tutorial "unit" -->
    <persistence-unit name="Tutorial">
        <class>org.datanucleus.samples.jdo.tutorial.Inventory</class>
        <class>org.datanucleus.samples.jdo.tutorial.Product</class>
        <class>org.datanucleus.samples.jdo.tutorial.Book</class>
        <exclude-unlisted-classes/>
        <properties>
            <property name="javax.jdo.option.ConnectionURL" value="mongodb:/nucleus"/>
            <property name="datanucleus.autoCreateSchema" value="true"/>
            <property name="datanucleus.validateTables" value="false"/>
            <property name="datanucleus.validateConstraints" value="false"/>
        </properties>
    </persistence-unit>
</persistence>
```

Note that you could equally use a properties file to define the persistence with JDO, but in this tutorial we use persistence.xml for convenience.

114.1.5 Step 3: Enhance your classes

JDO relies on the classes that you want to persist implementing PersistenceCapable. You could write your classes manually to do this but this would be laborious. Alternatively you can use a post-processing step to compilation that ‘enhances’ your compiled classes, adding on the necessary extra methods to make them PersistenceCapable. There are several ways to do this, most notably at post-compile, or at runtime. We use the post-compile step in this tutorial. DataNucleus JDO provides its own byte-code enhancer for instrumenting/enhancing your classes (in datanucleus-core) and this is included in the DataNucleus AccessPlatform zip file prerequisite.

To understand on how to invoke the enhancer you need to visualise where the various source and jdo files are stored.
The first thing to do is compile your domain/model classes. You can do this in any way you wish, but the downloadable JAR provides an Ant task, and a Maven2 project to do this for you.

Using Ant:
`ant compile`

Using Maven2:
`mvn compile`

To enhance classes using the DataNucleus Enhancer, you need to invoke a command something like this from the root of your project.

Using Ant:
`ant enhance`

Using Maven2: (this is usually done automatically after the "compile" goal)
`mvn datanucleus:enhance`

Manually on Linux/Unix:

Manually on Windows:
`java -cp target\classes;\bin\datanucleus-core.jar;\bin\datanucleus-api-jdo.jar;\bin\jdo-api.jar org.datanucleus.enhancer.DataNucleusEnhancer -pu Tutorial`

[Command shown on many lines to aid reading - should be on single line]

This command enhances the .class files that have @PersistenceCapable annotations. If you accidentally omitted this step, at the point of running your application and trying to persist an object, you would get a `ClassNotFoundException` thrown. The use of the enhancer is documented in more detail in the Enhancer Guide. The output of this step are a set of class files that represent `PersistenceCapable` classes.
114.1.6 Step 4 : Write the code to persist objects of your classes

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted, and when. Interaction with the persistence framework of JDO is performed via a PersistenceManager. This provides methods for persisting of objects, removal of objects, querying for persisted objects, etc. This section gives examples of typical scenarios encountered in an application.

The initial step is to obtain access to a PersistenceManager, which you do as follows

```java
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory("Tutorial");
PersistenceManager pm = pmf.getPersistenceManager();
```

Now that the application has a PersistenceManager it can persist objects. This is performed as follows

```java
Transaction tx=pm.currentTransaction();
try
{
    tx.begin();
    Inventory inv = new Inventory("My Inventory");
    Product product = new Product("Sony Discman", "A standard discman from Sony", 49.99);
    inv.getProducts().add(product);
    pm.makePersistent(inv);
    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
    pm.close();
}
```

Note the following

- We have persisted the Inventory but since this referenced the Product then that is also persisted.
- The finally step is important to tidy up any connection to the datastore, and close the PersistenceManager

If you want to retrieve an object from persistent storage, something like this will give what you need. This uses a "Query", and retrieves all Product objects that have a price below 150.00, ordering them in ascending price order.
Transaction tx = pm.currentTransaction();
try
{
    tx.begin();

    Query q = pm.newQuery("SELECT " + Product.class.getName() + 
        " WHERE price < 150.00 ORDER BY price ASC");
    List<Product> products = (List<Product>)q.execute();
    Iterator<Product> iter = products.iterator();
    while (iter.hasNext())
    {
        Product p = iter.next();
        ...
        // use the retrieved objects
    }

    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }

    pm.close();
}

If you want to delete an object from persistence, you would perform an operation something like

Transaction tx = pm.currentTransaction();
try
{
    tx.begin();

    // retrieval of objects etc

    pm.deletePersistent(product);

    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }

    pm.close();
}
Clearly you can perform a large range of operations on objects. We can’t hope to show all of these here. Any good JDO book will provide many examples.

114.1.7 Step 5: Run your application

To run your JDO-enabled application will require a few things to be available in the Java CLASSPATH, these being

- Any persistence.xml file for the PersistenceManagerFactory creation
- Any JDO XML MetaData files for your persistable classes (not used in this example)
- MongoDB driver class needed for accessing your datastore
- The JDO API JAR (defining the JDO interface)
- The DataNucleus Core, DataNucleus JDO API and DataNucleus MongoDB JARs

After that it is simply a question of starting your application and all should be taken care of. You can access the DataNucleus Log file by specifying the logging configuration properties, and any messages from DataNucleus will be output in the normal way. The DataNucleus log is a very powerful way of finding problems since it can list all SQL actually sent to the datastore as well as many other parts of the persistence process.
Using Ant (you need the included "persistence.xml" to specify your database)

ant run

Using Maven:

mvn exec:java

Manually on Linux/Unix:

java -cp lib/jdo-api.jar:lib/datanucleus-core.jar:lib/datanucleus-mongodb.jar:
   lib/datanucleus-api-jdo.jar:lib/{mongodb_jars}:target/classes/.
   org.datanucleus.samples.jdo.tutorial.Main

Manually on Windows:

java -cp lib\jdo-api.jar;lib\datanucleus-core.jar;lib\datanucleus-mongodb.jar;
   lib\datanucleus-api-jdo.jar;lib\{mongodb_jars};target\classes\.
   org.datanucleus.samples.jdo.tutorial.Main

Output:

DataNucleus Tutorial

Persisting products

Product and Book have been persisted

Retrieving Extent for Products

> Product : Sony Discman [A standard discman from Sony]
> Book : JRR Tolkien - Lord of the Rings by Tolkien

Executing Query for Products with price below 150.00

> Book : JRR Tolkien - Lord of the Rings by Tolkien

Deleting all products from persistence

Deleted 2 products

End of Tutorial

114.2 Part 2 : Next steps

In the above simple tutorial we showed how to employ JDO and persist objects to MongoDB. Obviously this just scratches the surface of what you can do, and to use JDO requires minimal work from the user. In this second part we show some further things that you are likely to want to do.

1. Step 6 : Controlling the schema.
2. Step 7 : Generate the database tables where your classes are to be persisted using SchemaTool.
114.2.1 Step 6: Controlling the schema

In the above simple tutorial we didn't look at controlling the schema generated for these classes. Now let's pay more attention to this part by defining XML Metadata for the schema.

```xml
<?xml version="1.0"?>
<!DOCTYPE orm PUBLIC
"-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
"http://java.sun.com/dtd/orm_2_0.dtd">
<orm>
  <package name="org.datanucleus.samples.jdo.tutorial">
    <class name="Inventory" identity-type="datastore" table="INVENTORIES">
      <inheritance strategy="new-table"/>
      <field name="name">
        <column name="INVENTORY_NAME" length="100" jdbc-type="VARCHAR"/>
      </field>
      <field name="products">
        <join/>
      </field>
    </class>
    <class name="Product" identity-type="datastore" table="PRODUCTS">
      <inheritance strategy="new-table"/>
      <field name="name">
        <column name="PRODUCT_NAME" length="100" jdbc-type="VARCHAR"/>
      </field>
      <field name="description">
        <column length="255" jdbc-type="VARCHAR"/>
      </field>
    </class>
    <class name="Book" identity-type="datastore" table="BOOKS">
      <inheritance strategy="new-table"/>
      <field name="isbn">
        <column length="20" jdbc-type="VARCHAR"/>
      </field>
      <field name="author">
        <column length="40" jdbc-type="VARCHAR"/>
      </field>
      <field name="publisher">
        <column length="40" jdbc-type="VARCHAR"/>
      </field>
    </class>
  </package>
</orm>
```

With JDO you have various options as far as where this XML MetaData file is placed in the file structure, and whether they refer to a single class, or multiple classes in a package. With the above example, we have both classes specified in the same file `package-mongodb.orm`, in the package these classes are in, since we want to persist to MongoDB.
114.2.2 Step 7: Generate any schema required for your domain classes

This step is optional, depending on whether you have an existing database schema. If you haven’t, at this point you can use the SchemaTool to generate the tables where these domain objects will be persisted. DataNucleus SchemaTool is a command line utility (it can be invoked from Maven2/Ant in a similar way to how the Enhancer is invoked). The first thing that you need is to update the persistence.xml file with your database details.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">

    <!-- Tutorial "unit" -->
    <persistence-unit name="Tutorial">
        <class>org.datanucleus.samples.jdo.tutorial.Inventory</class>
        <class>org.datanucleus.samples.jdo.tutorial.Product</class>
        <class>org.datanucleus.samples.jdo.tutorial.Book</class>
        <exclude-unlisted-classes/>
        <properties>
            <property name="javax.jdo.option.ConnectionURL" value="mongodb:/nucleus"/>
            <property name="datanucleus.autoCreateSchema" value="true"/>
            <property name="datanucleus.validateTables" value="false"/>
            <property name="datanucleus.validateConstraints" value="false"/>
        </properties>
    </persistence-unit>
</persistence>
```

Now we need to run DataNucleus SchemaTool. For our case above you would do something like this
Using Ant:
ant createschema

Using Maven2:
mvn datanucleus:schema-create

Manually on Linux/Unix:
java -cp target/classes:lib/datanucleus-core.jar:lib/datanucleus-mongodb.jar:
   lib/datanucleus-jdo-api.jar:lib/jdo-api.jar:lib/{mongodb_driver.jar}
org.datanucleus.store.schema.SchemaTool
   -create -pu Tutorial

Manually on Windows:
java -cp target\classes;lib\datanucleus-core.jar;lib\datanucleus-mongodb.jar;\
   lib\datanucleus-api-jdo.jar;lib\jdo-api.jar;lib\{mongodb_driver.jar}
org.datanucleus.store.schema.SchemaTool
   -create -pu Tutorial

This will generate the required tables, etc for the classes defined in the JDO Meta-Data file.

114.2.3 Any questions?

If you have any questions about this tutorial and how to develop applications for use with DataNucleus please read the online documentation since answers are to be found there. If you don't find what you're looking for go to our Forums.

The DataNucleus Team
115 Tutorial with HBase

115.1 DataNucleus - Tutorial for JDO using HBase

115.1.1 Background
An application can be JDO-enabled via many routes depending on the development process of the project in question. For example the project could use Eclipse as the IDE for developing classes. In that case the project would typically use the DataNucleus Eclipse plugin. Alternatively the project could use Ant, Maven or some other build tool. In this case this tutorial should be used as a guiding way for using DataNucleus in the application. The JDO process is quite straightforward.

1. Prerequisite: Download DataNucleus AccessPlatform
2. Step 1: Define their persistence definition using Meta-Data.
3. Step 2: Define the "persistence-unit"
4. Step 3: Compile your classes, and instrument them (using the DataNucleus enhancer).
5. Step 4: Write your code to persist your objects within the DAO layer.
6. Step 5: Run your application.

The tutorial guides you through this. You can obtain the code referenced in this tutorial from SourceForge (one of the files entitled "datanucleus-samples-jdo-tutorial-*").

115.1.2 Prerequisite: Download DataNucleus AccessPlatform
You can download DataNucleus in many ways, but the simplest is to download the distribution zip appropriate to your datastore (HBase in this case). You can do this from SourceForge DataNucleus download page. When you open the zip you will find DataNucleus jars in the lib directory, and dependency jars in the deps directory.

115.1.3 Step 1: Take your model classes and mark which are persistable
For our tutorial, say we have the following classes representing a store of products for sale.
package org.datanucleus.samples.jdo.tutorial;

public class Inventory {
    String name = null;
    Set<Product> products = new HashSet();

    public Inventory(String name) {
        this.name = name;
    }

    public Set<Product> getProducts() { return products; }
}

package org.datanucleus.samples.jdo.tutorial;

public class Product {
    long id;
    String name = null;
    String description = null;
    double price = 0.0;

    public Product(String name, String desc, double price) {
        this.name = name;
        this.description = desc;
        this.price = price;
    }
}

package org.datanucleus.samples.jdo.tutorial;

public class Book extends Product {
    String author = null;
    String isbn = null;
    String publisher = null;

    public Book(String name, String desc, double price, String author,
                String isbn, String publisher) {
        super(name, desc, price);
        this.author = author;
        this.isbn = isbn;
        this.publisher = publisher;
    }
So we have a relationship (Inventory having a set of Products), and inheritance (Product-Book). Now we need to be able to persist objects of all of these types, so we need to define persistence for them. There are many things that you can define when deciding how to persist objects of a type but the essential parts are

- Mark the class as PersistenceCapable so it is visible to the persistence mechanism
- Identify which field(s) represent the identity of the object (or use datastore-identity if no field meets this requirement).

So this is what we do now. Note that we could define persistence using XML metadata, annotations or via the JDO API. In this tutorial we will use annotations.

```java
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Inventory
{
    @PrimaryKey
    String name = null;
    ...
}
```

```java
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Product
{
    @PrimaryKey
    @Persistent(valueStrategy=IdGeneratorStrategy.INCREMENT)
    long id;
    ...
}
```

```java
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Book extends Product
{
    ...
}
```

Note that we mark each class that can be persisted with @PersistenceCapable and their primary key field(s) with @PrimaryKey. In addition we defined a valueStrategy for Product field id so that it will have its values generated automatically. In this tutorial we are using application identity which means that all objects of these classes will have their identity defined by the primary key field(s). You can read more in datastore identity and application identity when designing your systems persistence.
115.1.4 Step 2: Define the 'persistence-unit'

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted. You do this via a file META-INF/persistence.xml at the root of the CLASSPATH. Like this

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">
  <!-- JDO tutorial "unit" -->
  <persistence-unit name="Tutorial">
    <class>org.datanucleus.samples.jdo.tutorial.Inventory</class>
    <class>org.datanucleus.samples.jdo.tutorial.Product</class>
    <class>org.datanucleus.samples.jdo.tutorial.Book</class>
    <exclude-unlisted-classes/>
    <properties>
      <property name="javax.jdo.option.ConnectionURL" value="hbase:"/>
    </properties>
  </persistence-unit>
</persistence>
```

Note that you could equally use a properties file to define the persistence with JDO, but in this tutorial we use persistence.xml for convenience.

115.1.5 Step 3: Enhance your classes

JDO relies on the classes that you want to persist implementing PersistenceCapable. You could write your classes manually to do this but this would be laborious. Alternatively you can use a post-processing step to compilation that "enhances" your compiled classes, adding on the necessary extra methods to make them PersistenceCapable. There are several ways to do this, most notably at post-compile, or at runtime. We use the post-compile step in this tutorial. DataNucleus JDO provides its own byte-code enhancer for instrumenting/enhancing your classes (in datanucleus-core) and this is included in the DataNucleus AccessPlatform zip file prerequisite.

To understand on how to invoke the enhancer you need to visualise where the various source and jdo files are stored
The first thing to do is compile your domain/model classes. You can do this in any way you wish, but the downloadable JAR provides an Ant task, and a Maven2 project to do this for you.

Using Ant:
ant compile
Using Maven2:
mvn compile

To enhance classes using the DataNucleus Enhancer, you need to invoke a command something like this from the root of your project.

Using Ant:
ant enhance
Using Maven: (this is usually done automatically after the "compile" goal)
mvn datanucleus:enhance

Manually on Linux/Unix:
java -cp target/classes:lib/datanucleus-core.jar:
lib/datanucleus-api-jdo.jar:lib/jdo-api.jar
org.datanucleus.enhancer.DataNucleusEnhancer -pu Tutorial

Manually on Windows:
java -cp target\classes;lib\datanucleus-core.jar;
lib\datanucleus-api-jdo.jar;lib\jdo-api.jar
org.datanucleus.enhancer.DataNucleusEnhancer -pu Tutorial

This command enhances the .class files that have @PersistenceCapable annotations. If you accidentally omitted this step, at the point of running your application and trying to persist an object, you would get a ClassNotPersistenceCapableException thrown. The use of the enhancer is documented in more detail in the Enhancer Guide. The output of this step are a set of class files that represent PersistenceCapable classes.
115.1.6 Step 4 : Write the code to persist objects of your classes

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted, and when. Interaction with the persistence framework of JDO is performed via a PersistenceManager. This provides methods for persisting of objects, removal of objects, querying for persisted objects, etc. This section gives examples of typical scenarios encountered in an application.

The initial step is to obtain access to a PersistenceManager, which you do as follows

```java
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory("Tutorial");
PersistenceManager pm = pmf.getPersistenceManager();
```

Now that the application has a PersistenceManager it can persist objects. This is performed as follows

```java
Transaction tx = pm.currentTransaction();
try {
    tx.begin();
    Inventory inv = new Inventory("My Inventory");
    Product product = new Product("Sony Discman", "A standard discman from Sony", 49.99);
    inv.getProducts().add(product);
    pm.makePersistent(inv);
    tx.commit();
} finally {
    if (tx.isActive())
    {
        tx.rollback();
    }
    pm.close();
}
```

Note the following

- We have persisted the `Inventory` but since this referenced the `Product` then that is also persisted.
- The `finally` step is important to tidy up any connection to the datastore, and close the PersistenceManager

If you want to retrieve an object from persistent storage, something like this will give what you need. This uses a "Query", and retrieves all Product objects that have a price below 150.00, ordering them in ascending price order.
Transaction tx = pm.currentTransaction();
try {
    tx.begin();

    Query q = pm.newQuery("SELECT * FROM Product WHERE price < 150.00 ORDER BY price ASC");
    List<Product> products = (List<Product>)q.execute();
    Iterator<Product> iter = products.iterator();
    while (iter.hasNext())
    {
        Product p = iter.next();

        ... (use the retrieved objects)
    }
    tx.commit();
} finally {
    if (tx.isActive())
    {
        tx.rollback();
    }
    pm.close();
}

If you want to delete an object from persistence, you would perform an operation something like

Transaction tx = pm.currentTransaction();
try {
    tx.begin();

    ... (retrieval of objects etc)

    pm.deletePersistent(product);
    tx.commit();
} finally {
    if (tx.isActive())
    {
        tx.rollback();
    }
    pm.close();
}
Clearly you can perform a large range of operations on objects. We can’t hope to show all of these here. Any good JDO book will provide many examples.

115.1.7 Step 5: Run your application

To run your JDO-enabled application will require a few things to be available in the Java CLASSPATH, these being

- Any persistence.xml file for the PersistenceManagerFactory creation
- Any JDO XML MetaData files for your persistable classes (not used in this example)
- HBase driver class(es) needed for accessing your datastore
- The JDO API JAR (defining the JDO interface)
- The DataNucleus Core, DataNucleus JDO API and DataNucleus HBase JARs

After that it is simply a question of starting your application and all should be taken care of. You can access the DataNucleus Log file by specifying the logging configuration properties, and any messages from DataNucleus will be output in the normal way. The DataNucleus log is a very powerful way of finding problems since it can list all SQL actually sent to the datastore as well as many other parts of the persistence process.
Using Ant (you need the included "persistence.xml" to specify your database)

ant run

Using Maven:

mvn exec:java

Manually on Linux/Unix:

java -cp lib/jdo-api.jar:lib/datanucleus-core.jar:lib/datanucleus-hbase.jar:
   lib/datanucleus-api-jdo.jar:lib/{hbase_jars}:target/classes/:
   org.datanucleus.samples.jdo.tutorial.Main

Manually on Windows:

java -cp lib\jdo-api.jar;lib\datanucleus-core.jar;lib\datanucleus-hbase.jar;
   lib\datanucleus-api-jdo.jar;lib\{hbase_jars};target\classes\;
   org.datanucleus.samples.jdo.tutorial.Main

Output:

DataNucleus Tutorial
====================
Persisting products
Product and Book have been persisted

Retrieving Extent for Products
> Product : Sony Discman [A standard discman from Sony]
> Book : JRR Tolkien - Lord of the Rings by Tolkien

Executing Query for Products with price below 150.00
> Book : JRR Tolkien - Lord of the Rings by Tolkien

Deleting all products from persistence
Deleted 2 products

End of Tutorial

115.2 Part 2 : Next steps

In the above simple tutorial we showed how to employ JDO and persist objects to HBase. Obviously this just scratches the surface of what you can do, and to use JDO requires minimal work from the user. In this second part we show some further things that you are likely to want to do.

1. Step 6 : Controlling the schema.
2. Step 7 : Generate the database tables where your classes are to be persisted using SchemaTool.
115.2.1 Step 6: Controlling the schema

In the above simple tutorial we didn’t look at controlling the schema generated for these classes. Now let’s pay more attention to this part by defining XML Metadata for the schema.

```xml
<?xml version="1.0"?>
<!DOCTYPE orm PUBLIC
  "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
  "http://java.sun.com/dtd/orm_2_0.dtd">
<orm>
  <package name="org.datanucleus.samples.jdo.tutorial">
    <class name="Inventory" identity-type="datastore" table="INVENTORIES">
      <inheritance strategy="new-table"/>
      <field name="name">
        <column name="INVENTORY_NAME" length="100" jdbc-type="VARCHAR"/>
      </field>
      <field name="products">
        <join/>
      </field>
    </class>
    <class name="Product" identity-type="datastore" table="PRODUCTS">
      <inheritance strategy="new-table"/>
      <field name="name">
        <column name="PRODUCT_NAME" length="100" jdbc-type="VARCHAR"/>
      </field>
      <field name="description">
        <column length="255" jdbc-type="VARCHAR"/>
      </field>
    </class>
    <class name="Book" identity-type="datastore" table="BOOKS">
      <inheritance strategy="new-table"/>
      <field name="isbn">
        <column length="20" jdbc-type="VARCHAR"/>
      </field>
      <field name="author">
        <column length="40" jdbc-type="VARCHAR"/>
      </field>
      <field name="publisher">
        <column length="40" jdbc-type="VARCHAR"/>
      </field>
    </class>
  </package>
</orm>
```

With JDO you have various options as far as where this XML MetaData files is placed in the file structure, and whether they refer to a single class, or multiple classes in a package. With the above example, we have both classes specified in the same file `package-hbase.orm`, in the package these classes are in, since we want to persist to HBase.
115.2.2 Step 7: Generate any schema required for your domain classes

This step is optional, depending on whether you have an existing database schema. If you haven’t, at this point you can use the SchemaTool to generate the tables where these domain objects will be persisted. DataNucleus SchemaTool is a command line utility (it can be invoked from Maven2/Ant in a similar way to how the Enhancer is invoked). The first thing that you need is to update the persistence.xml file with your database details

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">

    <!-- Tutorial "unit" -->
    <persistence-unit name="Tutorial">
        <class>org.datanucleus.samples.jdo.tutorial.Inventory</class>
        <class>org.datanucleus.samples.jdo.tutorial.Product</class>
        <class>org.datanucleus.samples.jdo.tutorial.Book</class>
        <exclude-unlisted-classes/>
        <properties>
            <property name="javax.jdo.option.ConnectionURL" value="hbase:="/>
            <property name="datanucleus.autoCreateSchema" value="true"/>
            <property name="datanucleus.validateTables" value="false"/>
            <property name="datanucleus.validateConstraints" value="false"/>
        </properties>
    </persistence-unit>
</persistence>
```

Now we need to run DataNucleus SchemaTool. For our case above you would do something like this
Using Ant:
ant createschema

Using Maven2:
mvn datanucleus:schema-create

Manually on Linux/Unix:
java -cp target/classes:lib/datanucleus-core.jar:lib/datanucleus-hbase.jar:
   lib/datanucleus-jdo-api.jar:lib/jdo-api.jar:lib/{hbase_driver.jar}
   org.datanucleus.store.schema.SchemaTool
   -create -pu Tutorial

Manually on Windows:
java -cp target\classes;lib\datanucleus-core.jar;lib\datanucleus-hbase.jar;
   lib\datanucleus-api-jdo.jar;lib\jdo-api.jar;lib\{hbase_driver.jar}
   org.datanucleus.store.schema.SchemaTool
   -create -pu Tutorial

Note that "hbase_driver" typically means hbase.jar, hadoop-core.jar, zookeeper.jar and commons-logging.jar

[Command shown on many lines to aid reading. Should be on single line]

This will generate the required tables, etc for the classes defined in the JDO Meta-Data file.

115.2.3 Any questions?
If you have any questions about this tutorial and how to develop applications for use with DataNucleus please read the online documentation since answers are to be found there. If you don't find what you're looking for go to our Forums.

The DataNucleus Team
## 116 Tutorial with Neo4j

### 116.1 DataNucleus - Tutorial for JDO using Neo4J

[Download](#)

[Source Code (GitHub)](#)

#### 116.1.1 Background

An application can be JDO-enabled via many routes depending on the development process of the project in question. For example the project could use Eclipse as the IDE for developing classes. In that case the project would typically use the DataNucleus Eclipse plugin. Alternatively the project could use Ant, Maven or some other build tool. In this case this tutorial should be used as a guiding way for using DataNucleus in the application. The JDO process is quite straightforward.

1. **Prerequisite**: Download DataNucleus AccessPlatform
2. **Step 1**: Define their persistence definition using Meta-Data.
3. **Step 2**: Define the "persistence-unit"
4. **Step 3**: Compile your classes, and instrument them (using the DataNucleus enhancer).
5. **Step 4**: Write your code to persist your objects within the DAO layer.
6. **Step 5**: Run your application.

The tutorial guides you through this. You can obtain the code referenced in this tutorial from SourceForge (one of the files entitled "datanucleus-samples-jdo-tutorial-*").

#### 116.1.2 Prerequisite: Download DataNucleus AccessPlatform

You can download DataNucleus in many ways, but the simplest is to download the distribution zip appropriate to your datastore (Neo4J in this case, so get the full download). You can do this from SourceForge DataNucleus download page. When you open the zip you will find DataNucleus jars in the `lib` directory, and dependency jars in the `deps` directory.

#### 116.1.3 Step 1: Take your model classes and mark which are persistable

For our tutorial, say we have the following classes representing a store of products for sale.
package org.datanucleus.samples.jdo.tutorial;

public class Inventory
{
    String name = null;
    Set<Product> products = new HashSet();

    public Inventory(String name)
    {
        this.name = name;
    }

    public Set<Product> getProducts() {return products;}
}

package org.datanucleus.samples.jdo.tutorial;

public class Product
{
    long id;
    String name = null;
    String description = null;
    double price = 0.0;

    public Product(String name, String desc, double price)
    {
        this.name = name;
        this.description = desc;
        this.price = price;
    }
}

package org.datanucleus.samples.jdo.tutorial;

public class Book extends Product
{
    String author=null;
    String isbn=null;
    String publisher=null;

    public Book(String name, String desc, double price, String author,
                String isbn, String publisher)
    {
        super(name,desc,price);
        this.author = author;
        this.isbn = isbn;
        this.publisher = publisher;
    }
}
So we have a relationship (Inventory having a set of Products), and inheritance (Product-Book). Now we need to be able to persist objects of all of these types, so we need to define persistence for them. There are many things that you can define when deciding how to persist objects of a type but the essential parts are

- Mark the class as `PersistenceCapable` so it is visible to the persistence mechanism
- Identify which field(s) represent the identity of the object (or use datastore-identity if no field meets this requirement).

So this is what we do now. Note that we could define persistence using XML metadata, annotations or via the JDO API. In this tutorial we will use annotations.

```java
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Inventory {
    @PrimaryKey
    String name = null;
    ...
}
```

```java
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Product {
    @PrimaryKey
    @Persistent(valueStrategy=IdGeneratorStrategy.NATIVE)
    long id;
    ...
}
```

```java
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Book extends Product {
    ...
}
```

Note that we mark each class that can be persisted with `@PersistenceCapable` and their primary key field(s) with `@PrimaryKey`. In addition we defined a `valueStrategy` for Product field `id` so that it will have its values generated automatically. In this tutorial we are using application identity which means that all objects of these classes will have their identity defined by the primary key field(s). You can read more in datastore identity and application identity when designing your systems persistence.
116.1.4 Step 2: Define the 'persistence-unit'

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted. You do this via a file META-INF/persistence.xml at the root of the CLASSPATH. Like this

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">

<!-- JDO tutorial "unit" -->
<persistence-unit name="Tutorial">
    <class>org.datanucleus.samples.jdo.tutorial.Inventory</class>
    <class>org.datanucleus.samples.jdo.tutorial.Product</class>
    <class>org.datanucleus.samples.jdo.tutorial.Book</class>
    <exclude-unlisted-classes/>
    <properties>
        <property name="javax.jdo.option.ConnectionURL" value="neo4j:testDB"/>
    </properties>
</persistence-unit>
</persistence>
```

Note that you could equally use a properties file to define the persistence with JDO, but in this tutorial we use persistence.xml for convenience.

116.1.5 Step 3: Enhance your classes

JDO relies on the classes that you want to persist implementing PersistenceCapable. You could write your classes manually to do this but this would be laborious. Alternatively you can use a post-processing step to compilation that "enhances" your compiled classes, adding on the necessary extra methods to make them PersistenceCapable. There are several ways to do this, most notably at post-compile, or at runtime. We use the post-compile step in this tutorial. **DataNucleus JDO** provides its own byte-code enhancer for instrumenting/enhancing your classes (in datanucleus-core) and this is included in the DataNucleus AccessPlatform zip file prerequisite.

To understand on how to invoke the enhancer you need to visualise where the various source and jdo files are stored
src/main/java/org/datanucleus/samples/jdo/tutorial/Book.java
src/main/java/org/datanucleus/samples/jdo/tutorial/Inventory.java
src/main/java/org/datanucleus/samples/jdo/tutorial/Product.java
src/main/resources/META-INF/persistence.xml

target/classes/org/datanucleus/samples/jdo/tutorial/Book.class
target/classes/org/datanucleus/samples/jdo/tutorial/Inventory.class
target/classes/org/datanucleus/samples/jdo/tutorial/Product.class

[when using Ant]
lib/jdo-api.jar
lib/datanucleus-core.jar
lib/datanucleus-api-jdo.jar

The first thing to do is compile your domain/model classes. You can do this in any way you wish, but the downloadable JAR provides an Ant task, and a Maven2 project to do this for you.

Using Ant :
ant compile

Using Maven2 :
mvn compile

To enhance classes using the DataNucleus Enhancer, you need to invoke a command something like this from the root of your project.

Using Ant :
ant enhance

Using Maven : (this is usually done automatically after the "compile" goal)
mvn datanucleus:enhance

Manually on Linux/Unix :
java -cp target/classes:lib/datanucleus-core.jar;
   lib/datanucleus-api-jdo.jar:lib/jdo-api.jar
   org.datanucleus.enhancer.DataNucleusEnhancer -pu Tutorial

Manually on Windows :
java -cp target\classes;lib\datanucleus-core.jar;
   lib\datanucleus-api-jdo.jar;lib\jdo-api.jar
   org.datanucleus.enhancer.DataNucleusEnhancer -pu Tutorial

[Command shown on many lines to aid reading - should be on single line]

This command enhances the .class files that have @PersistenceCapable annotations. If you accidentally omitted this step, at the point of running your application and trying to persist an object, you would get a ClassNotPersistenceCapableException thrown. The use of the enhancer is documented in more detail in the Enhancer Guide. The output of this step are a set of class files that represent PersistenceCapable classes.
116.1.6 Step 4: Write the code to persist objects of your classes

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted, and when. Interaction with the persistence framework of JDO is performed via a PersistenceManager. This provides methods for persisting of objects, removal of objects, querying for persisted objects, etc. This section gives examples of typical scenarios encountered in an application.

The initial step is to obtain access to a PersistenceManager, which you do as follows:

```java
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory("Tutorial");
PersistenceManager pm = pmf.getPersistenceManager();
```

Now that the application has a PersistenceManager it can persist objects. This is performed as follows:

```java
Transaction tx = pm.currentTransaction();
try {
    tx.begin();
    Inventory inv = new Inventory("My Inventory");
    Product product = new Product("Sony Discman", "A standard discman from Sony", 49.99);
    inv.getProducts().add(product);
    pm.makePersistent(inv);
    tx.commit();
} finally {
    if (tx.isActive()) {
        tx.rollback();
    }
    pm.close();
}
```

Note the following:

- We have persisted the `Inventory` but since this referenced the `Product` then that is also persisted.
- The `finally` step is important to tidy up any connection to the datastore, and close the PersistenceManager.

If you want to retrieve an object from persistent storage, something like this will give what you need. This uses a "Query", and retrieves all Product objects that have a price below 150.00, ordering them in ascending price order.
Transaction tx = pm.currentTransaction();
try {
    tx.begin();
    Query q = pm.newQuery("SELECT " + Product.class.getName() + " WHERE price < 150.00 ORDER BY price ASC");
    List<Product> products = (List<Product>)q.execute();
    Iterator<Product> iter = products.iterator();
    while (iter.hasNext())
    {
        Product p = iter.next();
        // ... (use the retrieved objects)
    }
    tx.commit();
} finally {
    if (tx.isActive())
    {
        tx.rollback();
    }
    pm.close();
}

If you want to delete an object from persistence, you would perform an operation something like:

Transaction tx = pm.currentTransaction();
try {
    tx.begin();
    // ... (retrieval of objects etc)
    pm.deletePersistent(product);
    tx.commit();
} finally {
    if (tx.isActive())
    {
        tx.rollback();
    }
    pm.close();
}
Clearly you can perform a large range of operations on objects. We can't hope to show all of these here. Any good JDO book will provide many examples.

116.1.7 Step 5: Run your application

To run your JDO-enabled application will require a few things to be available in the Java CLASSPATH, these being

- Any persistence.xml file for the PersistenceManagerFactory creation
- Any JDO XML MetaData files for your persistable classes (not used in this example)
- Neo4J jar needed for accessing your datastore
- The JDO API JAR (defining the JDO interface)
- The DataNucleus Core, DataNucleus JDO API and DataNucleus Neo4J JARs

After that it is simply a question of starting your application and all should be taken care of. You can access the DataNucleus Log file by specifying the logging configuration properties, and any messages from DataNucleus will be output in the normal way. The DataNucleus log is a very powerful way of finding problems since it can list all SQL actually sent to the datastore as well as many other parts of the persistence process.
Using Ant (you need the included "persistence.xml" to specify your database)
```
ant run
```

Using Maven:
```
mvn exec:java
```

Manually on Linux/Unix :
```
java -cp lib/jdo-api.jar:lib/datanucleus-core.jar:lib/datanucleus-neo4j.jar:
    lib/datanucleus-api-jdo.jar:lib/{neo4j_jars}:target/classes/:.
    org.datanucleus.samples.jdo.tutorial.Main
```

Manually on Windows :
```
javac -cp lib\jdo-api.jar;lib\datanucleus-core.jar;lib\datanucleus-neo4j.jar;
    lib\datanucleus-api-jdo.jar;lib\{neo4j_jars};target\classes\;
    org.datanucleus.samples.jdo.tutorial.Main
```

Output :

```
DataNucleus Tutorial
====================
Persisting products
Product and Book have been persisted

Retrieving Extent for Products
> Product : Sony Discman [A standard discman from Sony]
> Book : JRR Tolkien - Lord of the Rings by Tolkien

Executing Query for Products with price below 150.00
> Book : JRR Tolkien - Lord of the Rings by Tolkien

Deleting all products from persistence
Deleted 2 products

End of Tutorial
```

116.2 Part 2 : Next steps

In the above simple tutorial we showed how to employ JDO and persist objects to Neo4J. Obviously this just scratches the surface of what you can do, and to use JDO requires minimal work from the user. In this second part we show some further things that you are likely to want to do.

1. Step 6 : Controlling the property names.

116.2.1 Step 6 : Controlling the schema

In the above simple tutorial we didn’t look at controlling the property names of the Nodes generated for these classes. Now let’s pay more attention to this part by defining XML Metadata for the schema.
With JDO you have various options as far as where this XML MetaData files is placed in the file structure, and whether they refer to a single class, or multiple classes in a package. With the above example, we have both classes specified in the same file package-neo4j.orm, in the package these classes are in, since we want to persist to Neo4J.

116.2.2 Any questions?

If you have any questions about this tutorial and how to develop applications for use with DataNucleus please read the online documentation since answers are to be found there. If you don’t find what you’re looking for go to our Forums.

The DataNucleus Team
117 1-N Bidir FK Relation

117.1 JDO Guides : 1-N Bidirectional Relation using Foreign-Key

This guide demonstrates a 1-N collection relationship between 2 classes. In this sample we have Pack and Card such that each Pack can contain many Cards. In addition each Card has a Pack that it belongs to. We demonstrate the classes themselves, and the MetaData necessary to persist them to the datastore in the way that we require. In this case we are going to persist the relation to an RDBMS using a ForeignKey.

1. **Classes** - Design your Java classes to represent what you want to model in your system. Persistence doesn't have much of an impact on this stage, but we'll analyse the very minor influence it does have.

2. **Object Identity** - Decide how the identities of your objects of these classes will be defined. Do you want JDO to give them id's or will you do it yourself.

3. **Meta-Data** - Define how your objects of these classes will be persisted.

   1. **New Database Schema** - you have a clean sheet of paper and can have them persisted with no constraints.
   2. **Existing Database Schema** - you have existing tables that you need the objects persisted to.

117.1.1 The Classes

Lets look at our initial classes for the example. We want to represent a pack of cards.
package org.datanucleus.samples.packofcards.inverse;

public class Pack
{
    String name=null;
    String description=null;

    Set    cards=new HashSet();

    public Pack(String name, String desc)
    {
        this.name = name;
        this.description = desc;
    }

    public void addCard(Card card)
    {
        cards.add(card);
    }

    public void removeCard(Card card)
    {
        cards.remove(card);
    }

    public Set getCards()
    {
        return cards;
    }

    public int getNumberOfCards()
    {
        return cards.size();
    }
}

public class Card
{
    String suit=null;
    String number=null;
    Pack   pack=null;

    public Card(String suit,String number)
    {
        this.suit = suit;
        this.number = number;
    }

    public String getSuit()
    {
        return suit;
    }

    public String getNumber()
    {
        return number;
    }

    public Pack getPack()
    {
        return pack;
    }
}
The first thing that we need to do is add a default constructor. This is a requirement of JDO. This can be private if we wish, so we add

```java
public class Pack {
    private Pack() {
    }
    ...
}
public class Card {
    private Card() {
    }
    ...
}
```

### 117.1.2 Object Identity

The next thing to do is decide if we want to allow DataNucleus to generate the identities of our objects, or whether we want to do it ourselves. In our case we will allow DataNucleus to create the identities for our `Pack`s and also for our `Card`s.

In the case of `Pack` there is nothing more to code since DataNucleus will handle the identities. Similarly, in the case of `Card` there is nothing more to add.

### 117.1.3 MetaData for New Schema

Now that we've decided on our classes and how we want to define their identities we can decide on the precise persistence definition in the datastore. In this section we'll describe how to persist these objects to a new database schema where we can create new tables and don't need to write to some existing table.

Some JDO tools provide an IDE to generate Meta-Data files, but DataNucleus doesn't currently. Either way it is a good idea to become familiar with the structure of these files since they define how your classes are persisted. Let's start with the header area. You add a block like this to define that the file is JDO Meta-Data

```xml
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC
   "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
   "http://java.sun.com/dtd/jdo_2_0.dtd">
</jdo>
```

Now let's define the persistence for our `Pack` class. We are going to use datastore identity here, meaning that DataNucleus will assign id's to each `Pack` object persisted. We define it as follows...
Here we've defined that our name field will be persisted to a VARCHAR(100) column, our description field will be persisted to a VARCHAR(255) column, and that our cards field is a Collection containing org.datanucleus.samples.packofcards.inverse.Card objects. In addition, it specifies that there is a pack field in the Card class (the mapped-by attribute) that gives the related pack (with the Pack being the owner of the relationship). This final information is to inform DataNucleus to link the table for this class (via a foreign key) to the table for Card class. This is what is termed a ForeignKey relationship. Please refer to the 1-N Relationships Guide for more details on this. We'll discuss join table relationships in a different example.

Now lets define the persistence for our Card class. We are going to use datastore identity here, meaning that DataNucleus will assign the id's for any object of type Card. We define it as follows:

Here we've defined that our suit field will be persisted to a VARCHAR(10) column, our number field will be persisted to a VARCHAR(20) column.

We finally terminate the Meta-Data file with the closing tag:

117.1.4 MetaData for Existing Schema

Now that we’ve decided on our classes and how we want to define their identities we can decide on the precise persistence definition. In this section we’ll describe how to persist these objects to an existing database schema where we already have some database tables from a previous persistence.
mechanism and we want to use those tables (because they have data in them). Our existing tables are shown below.

We will take the Meta-Data that was described in the previous section (New Schema) and continue from there. To recap, here is what we arrived at

```xml
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC 
 "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN" 
 "http://java.sun.com/dtd/jdo_2_0.dtd">
</jdo>

<package name="org.datanucleus.samples.packofcards.inverse">
    <class name="Pack" identity-type="datastore">
        <field name="name" persistence-modifier="persistent">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="description" persistence-modifier="persistent">
            <column length="255" jdbc-type="VARCHAR"/>
        </field>
        <field name="cards" persistence-modifier="persistent" mapped-by="pack">
            <collection element-type="org.datanucleus.samples.packofcards.inverse.Card">
            </collection>
        </field>
    </class>
</package>
```

The first thing we need to do is map the Pack class to the table that we have in our database. It needs to be mapped to a table called "DECK", with columns "IDENTIFIERNAME" and "DETAILS", and the identity column that DataNucleus uses needs to be called IDENTIFIER_ID. We do this by changing the Meta-Data to be
So we made use of the attribute `table` (of element `class`) and `name` (of element `column`) to align to the table that is there. In addition we made use of the `datastore-identity` element to map the identity column name. Lets now do the same for the class `Card`. In our database we want this to map to a table called "PLAYINGCARD", with columns "SET" and "VALUE". So we do the same thing to its Meta-

OK, so we've now mapped our 2 classes to their tables. This completes our job. The only other aspect that is likely to be met is where a column in the database is of a particular type, but we'll cover that in a different example.

One thing worth mentioning is the difference if our Collection class was a List, ArrayList, Vector, etc. In this case we need to specify the ordering column for maintaining the order within the List. In our case we want to specify this column to be called "IDX", so we do it like this.
<class name="Card" identity-type="datastore" table="PLAYINGCARD">
    <datastore-identity>
        <column name="PLAYINGCARD_ID"/>
    </datastore-identity>
    <field name="suit">
        <column name="SET" length="10" jdbc-type="VARCHAR"/>
    </field>
    <field name="number">
        <column name="VALUE" length="20" jdbc-type="VARCHAR"/>
    </field>
    <field name="pack">
        <column name="DECK_ID"/>
        <order column="IDX"/>
    </field>
</class>
118 M-N Relation

118.1 JDO Guides: M-N Relation

This guide demonstrates an M-N collection relationship between 2 classes. In this sample we have Supplier and Customer such that each Customer can contain many Suppliers. In addition each Supplier can have many Customers. We demonstrate the classes themselves, and the MetaData necessary to persist them to the datastore in the way that we require. In this example we use XML metadata, but you could easily use annotations

1. **Classes** - Design your Java classes to represent what you want to model in your system. JDO doesn't have much of an impact on this, but we'll analyse the very minor influence it does have.
2. **Meta-Data** - Define how your objects of these classes will be persisted.
   1. **New Database Schema** - you have a clean sheet of paper and can have them persisted with no constraints.
   2. **Existing Database Schema** - you have existing tables that you need the objects persisted to.
3. **Managing the Relationship** - How we add/remove elements to/from the M-N relation.

118.1.1 The Classes

Lets look at our initial classes for the example. We want to represent the relation between a customer and a supplier.
package org.datanucleus.samples.m_to_n;

public class Customer
{
    String name = null;
    String description = null;
    Collection suppliers = new HashSet();

    public Customer(String name, String desc)
    {
        this.name = name;
        this.description = desc;
    }

    public void addSupplier(Supplier supplier)
    {
        suppliers.add(supplier);
    }

    public void removeSupplier(Supplier supplier)
    {
        suppliers.remove(supplier);
    }

    public Collection getSuppliers()
    {
        return suppliers;
    }

    public int getNumberOfSuppliers()
    {
        return suppliers.size();
    }
}
public class Supplier
{
    String name = null;
    String address = null;
    Collection customers = new HashSet();

    public Supplier(String name, String address)
    {
        this.name = name;
        this.address = address;
    }

    public String getName()
    {
        return name;
    }

    public String getAddress()
    {
        return address;
    }

    public void addCustomer(Customer customer)
    {
        customers.add(customer);
    }

    public void removeCustomer(Customer customer)
    {
        customers.remove(customer);
    }

    public Collection getCustomers()
    {
        return customers;
    }

    public int getNumberOfCustomers()
    {
        return customers.size();
    }
}

The first thing that we need to do is add a default constructor. This is a requirement of JDO. In our case we are using the DataNucleus enhancer and this will automatically add the default constructor when not present, so we omit this.

In this example we don't care about the "identity" type chosen so we will use datastore-identity. Please refer to the documentation for examples of application and datastore identity for how to specify them.
118.1.2 MetaData for New Schema

Now that we've decided on our classes and how we want to define their identities we can decide on the precise persistence definition. In this section we'll describe how to persist these objects to a new database schema where we can create new tables and don't need to write to some existing table.

Some JDO tools provide an IDE to generate Meta-Data files, but DataNucleus doesn't currently. Either way it is a good idea to become familiar with the structure of these files since they define how your classes are persisted. Lets start with the header area. You add a block like this to define that the file is JDO Meta-Data

```xml
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC
   "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
   "http://java.sun.com/dtd/jdo_2_0.dtd">
</jdo>
```

Now let's define the persistence for our **Customer** class. We define it as follows

```xml
<package name="org.datanucleus.samples.m_to_n">
  <class name="Customer" identity-type="datastore">
    <field name="name" persistence-modifier="persistent">
      <column length="100" jdbc-type="VARCHAR"/>
    </field>
    <field name="description" persistence-modifier="persistent">
      <column length="255" jdbc-type="VARCHAR"/>
    </field>
    <field name="suppliers" persistence-modifier="persistent" mapped-by="customers">
      <collection element-type="org.datanucleus.samples.m_to_n.Supplier"/>
      <join/>
    </field>
  </class>
</package>
```

Here we've defined that our **name** field will be persisted to a VARCHAR(100) column, our **description** field will be persisted to a VARCHAR(255) column, and that our **suppliers** field is a Collection containing `org.datanucleus.samples.m_to_n.Supplier` objects. In addition, it specifies that there is a **customers** field in the **Supplier** class (the mapped-by attribute) that gives the related customers for the Supplier. This final information is to inform DataNucleus to link the table for this class to the table for **Supplier** class. This is what is termed an M-N relationship. Please refer to the **M-N Relationships Guide** for more details on this.

Now let's define the persistence for our **Supplier** class. We define it as follows

```xml
<package name="org.datanucleus.samples.m_to_n">
  <class name="Supplier" identity-type="datastore">
    <field name="name" persistence-modifier="persistent">
      <column length="100" jdbc-type="VARCHAR"/>
    </field>
    <field name="description" persistence-modifier="persistent">
      <column length="255" jdbc-type="VARCHAR"/>
    </field>
    <field name="customers" persistence-modifier="persistent" mapped-by="suppliers">
      <collection element-type="org.datanucleus.samples.m_to_n.Customer"/>
      <join/>
    </field>
  </class>
</package>
```
Here we've defined that our *name* field will be persisted to a VARCHAR(100) column, our *address* field will be persisted to a VARCHAR(100) column.

We finally terminate the Meta-Data file with the closing tag

</jdo>

### 118.1.3 MetaData for Existing Schema

Now that we've decided on our classes and how we want to define their identities we can decide on the precise persistence definition. In this section we'll describe how to persist these objects to an existing database schema where we already have some database tables from a previous persistence mechanism and we want to use those tables (because they have data in them). Our existing tables are shown below.

We will take the Meta-Data that was described in the previous section (New Schema) and continue from there. To recap, here is what we arrived at
The first thing we need to do is map the **Customer** class to the table that we have in our database. It needs to be mapped to a table called "CUSTOMER", with columns "NAME" and "DESC", and the identity column that DataNucleus uses needs to be called CUST_ID. We do this by changing the Meta-Data to be
We now need to define the mapping for the join table storing the relationship information. So we make use of the "table" attribute of field to define this, and use the join and element subelements to define the columns of the join table. Like this

```xml
<class name="Customer" identity-type="datastore" table="CUSTOMER">
    <datastore-identity>
        <column name="CUST_ID"/>
    </datastore-identity>
    <field name="name" persistence-modifier="persistent">
        <column name="NAME" length="100" jdbc-type="VARCHAR"/>
    </field>
    <field name="description" persistence-modifier="persistent">
        <column name="DESC" length="100" jdbc-type="VARCHAR"/>
    </field>
    <field name="suppliers" persistence-modifier="persistent" mapped-by="customers">
        <collection element-type="org.datanucleus.samples.m_to_n.Supplier"/>
        <join>
            <column name="CUST_ID"/>
        </join>
    </field>
</class>
```

Let's now do the same for the class Supplier. In our database we want this to map to a table called "SUPPLIER", with columns "SUPP_ID" (identity), "NAME" and "ADDR". We need to do nothing more for the join table since it is shared and we have defined its table/columns above.
OK, so we've now mapped our 2 classes to their tables. This completes our job. The only other aspect that is likely to be met is where a column in the database is of a particular type, but we'll cover that in a different example.

118.1.4 Management of the Relation

We now have our classes and the definition of persistence and we need to use our classes in the application. This section defines how we maintain the relation between the objects. Let's start by creating a few objects.
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
Object[] custIds = new Object[2];
Object[] suppIds = new Object[3];
try {
    tx.begin();
    Customer cust1 = new Customer("DFG Stores", "Small shop in London");
    Customer cust2 = new Customer("Kevins Cards", "Gift shop");

    Supplier supp1 = new Supplier("Stationery Direct", "123 The boulevard, Milton Keynes, UK");
    Supplier supp2 = new Supplier("Grocery Wholesale", "56 Jones Industrial Estate, London, UK");
    Supplier supp3 = new Supplier("Makro", "1 Parkville, Wembley, UK");

    pm.makePersistent(cust1);
    pm.makePersistent(cust2);
    pm.makePersistent(supp1);
    pm.makePersistent(supp2);
    pm.makePersistent(supp3);
    tx.commit();
    custIds[0] = JDOHelper.getObjectId(cust1);
    custIds[1] = JDOHelper.getObjectId(cust2);
    suppIds[0] = JDOHelper.getObjectId(supp1);
    suppIds[1] = JDOHelper.getObjectId(supp2);
    suppIds[2] = JDOHelper.getObjectId(supp3);
} catch (Exception e) {
    // Handle any errors
} finally {
    if (tx.isActive()) {
        tx.rollback();
    }
    pm.close();
}

OK. We’ve now persisted some Customers and Suppliers into our datastore. We now need to establish the relations.
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try {
    tx.begin();
    Customer cust1 = (Customer) pm.getObjectById(custIds[0]);
    Customer cust2 = (Customer) pm.getObjectById(custIds[1]);
    Supplier supp1 = (Supplier) pm.getObjectById(suppIds[0]);
    Supplier supp2 = (Supplier) pm.getObjectById(suppIds[1]);

    // Establish the relation customer1 uses supplier2
    cust1.addSupplier(supp2);
    supp2.addCustomer(cust1);

    // Establish the relation customer2 uses supplier1
    cust2.addSupplier(supp1);
    supp1.addCustomer(cust2);

    tx.commit();
} catch (Exception e) {
    // Handle any errors
} finally {
    if (tx.isActive()) {
        tx.rollback();
    }
    pm.close();
}

You note that we set both sides of the relation. This is important since JDO doesn't define support for "managed relations" before JDO2.1. We could have adapted the Customer method addSupplier to add both sides of the relation (or alternatively via Supplier method addCustomer) to simplify this process.

Let's now assume that over time we want to change our relationships.
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try
{
    tx.begin();

    // Retrieve the objects
    Customer cust1 = (Customer)pm.getObjectById(custIds[0]);
    Customer cust2 = (Customer)pm.getObjectById(custIds[1]);
    Supplier supp1 = (Supplier)pm.getObjectById(suppIds[0]);
    Supplier supp2 = (Supplier)pm.getObjectById(suppIds[1]);
    Supplier supp2 = (Supplier)pm.getObjectById(suppIds[1]);
    Supplier supp2 = (Supplier)pm.getObjectById(suppIds[1]);

    // Remove the relation from customer1 to supplier2, and add relation to supplier3
    cust1.removeSupplier(supp2);
    supp2.removeCustomer(cust1);
    cust1.addSupplier(supp3);
    supp3.addCustomer(cust1);

    // Add a relation customer2 uses supplier3
    cust2.addSupplier(supp3);
    supp3.addCustomer(cust2);

    tx.commit();
}
catch (Exception e)
{
    // Handle any errors
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
    pm.close();
}

So now we have customer1 with a relation to supplier3, and we have customer2 with relations to supplier1 and supplier3. That should give enough idea of how to manage the relations. The most important thing with any bidirectional relation is to set both sides of the relation.
119 M-N Attributed Relation

119.1 JDO Guides : M-N Attributed Relation

DataNucleus provides support for standard JDO M-N relations where we have a relation between, for example, Customer and Supplier, where a Customer has many Suppliers and a Supplier has many Customers. A slight modification on this is where you have the relation carrying some additional attributes of the relation. Let’s take some classes

```java
public class Customer {
    private long id; // PK
    private String name;
    private Set supplierRelations = new HashSet();
    // ...
}

public class Supplier {
    private long id; // PK
    private String name;
    private Set customerRelations = new HashSet();
    // ...
}
```

Now we obviously cant define an "attributed relation" using Java and just these classes so we invent an intermediate "associative" class, that will also contain the attributes.

```java
public class BusinessRelation {
    private Customer customer; // PK
    private Supplier supplier; // PK
    private String relationLevel;
    private String meetingLocation;

    public BusinessRelation(Customer cust, Supplier supp, String level, String meeting) {
        this.customer = cust;
        this.supplier = supp;
        this.relationLevel = level;
        this.meetingLocation = meeting;
    }
    // ...
}
```
So we define the metadata like this

```xml
<jdo>
  <package name="mydomain.business">
    <class name="Customer" detachable="true" table="CUSTOMER">
      <field name="id" primary-key="true" value-strategy="increment" column="ID"/>
      <field name="name" column="NAME"/>
      <field name="supplierRelations" persistence-modifier="persistent" mapped-by="customer">
        <collection element-type="BusinessRelation"/>
      </field>
    </class>
  </package>
</jdo>
```

So we’ve used a 1-N "CompoundIdentity" relation between Customer and BusinessRelation, and similarly between Supplier and BusinessRelation meaning that BusinessRelation has a composite PK define like this

```
<jdo>
  <package name="mydomain.business">
    <class name="Supplier" detachable="true" table="SUPPLIER">
      <field name="id" primary-key="true" value-strategy="increment" column="ID"/>
      <field name="name" column="NAME"/>
      <field name="customerRelations" persistence-modifier="persistent" mapped-by="supplier">
        <collection element-type="BusinessRelation"/>
      </field>
    </class>
  </package>
</jdo>
```
This arrangement will result in the following schema

![Diagram showing relationships between Customer, BusinessRelation, and Supplier]

So all we need to do now is persist some objects using these classes
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory("jpox.properties");
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
Object holderId = null;
try {
    tx.begin();
    Customer cust1 = new Customer("Web design Inc");
    Supplier supp1 = new Supplier("DataNucleus Corporation");
    pm.makePersistent(cust1);
    pm.makePersistent(supp1);
    BusinessRelation rel_1_1 = new BusinessRelation(cust1, supp1, "Very Friendly", "Hilton Hotel, London");
    cust1.addRelation(rel_1_1);
    supp1.addRelation(rel_1_1);
    pm.makePersistent(rel_1_1);
    tx.commit();
}
finally {
    if (tx.isActive()) {
        tx.rollback();
    }
    pm.close();
}

This will now have persisted an entry in table "CUSTOMER", an entry in table "SUPPLIER", and an entry in table "BUSINESSRELATION". We can now utilise the BusinessRelation objects to update the attributes of the M-N relation as we wish.
120 Spatial Types Tutorial

120.1 Persistence of Spatial Data using JDO

120.1.1 Background

dataNucleus-spatial allows the use of DataNucleus as persistence layer for geospatial applications in an environment that supports the OGC SFA specification. It allows the persistence of the Java geometry types from the JTS topology suite as well as those from the PostGIS project.

In this tutorial, we perform the basic persistence operations over spatial types using MySQL/MariaDB and Postgis products.

1. Step 1: Install the database server and spatial extensions.
2. Step 2: Download DataNucleus and PostGis libraries.
3. Step 3: Design and implement the persistent data model.
4. Step 4: Design and implement the persistent code.
5. Step 5: Run your application.

120.1.2 Step 1: Install the database server and spatial extensions

Download MySQL/MariaDB database and PostGIS. Install MySQL/MariaDB and PostGis. During PostGis installation, you will be asked to select the database schema where the spatial extensions will be enabled. You will use this schema to run the tutorial application.

120.1.3 Step 2: Download DataNucleus and PostGis libraries

Download the DataNucleus core, RDBMS and Spatial jars and any dependencies. Configure your development environment by adding the PostGIS and JDO jars to the classpath.
120.1.4 Step 3: Design and implement the persistent data model

```java
package org.datanucleus.samples.spatial;

import org.postgis.Point;

public class Position {
    private String name;
    private Point point;

    public Position(String name, Point point) {
        this.name = name;
        this.point = point;
    }

    public String getName() {
        return name;
    }

    public Point getPoint() {
        return point;
    }

    public String toString() {
        return "\[name\] \+ name + " \[point\] \+point;"
    }
}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "file:/javax/jdo/jdo.dtd">
<jdo>
    <extension vendor-name="datanucleus" key="spatial-dimension" value="2"/>
    <extension vendor-name="datanucleus" key="spatial-srid" value="4326"/>
    <class name="Position" table="spatialpostut" detachable="true">
        <field name="name"/>
        <field name="point" persistence-modifier="persistent"/>
    </class>
</package>
</jdo>
```

The above JDO metadata has two extensions `spatial-dimension` and `spatial-srid`. These settings specifies the format of the spatial data. SRID stands for spatial referencing system identifier and Dimension the number of coordinates.
120.1.5 Step 4: Design and implement the persistent code

In this tutorial, we query for all locations where the X coordinate is greater than 10 and Y coordinate is 0.
package org.datanucleus.samples.spatial;
import java.sql.SQLException;
import java.util.List;
import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;
import javax.jdo.Query;
import javax.jdo.Transaction;
import org.postgis.Point;

public class Main
{
    public static void main(String args[]) throws SQLException
    {
        // Create a PersistenceManagerFactory for this datastore
        PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory("MyUnit");
        System.out.println("DataNucleus JDO Spatial Sample");
        System.out.println("================================================");

        // Persistence of a Product and a Book.
        PersistenceManager pm = pmf.getPersistenceManager();
        Transaction tx = pm.currentTransaction();
        try
        {
            //create objects
            tx.begin();
            Position[] sps = new Position[3];
            Point[] points = new Point[3];
            points[0] = new Point("SRID=4326;POINT(5 0)");
            points[1] = new Point("SRID=4326;POINT(10 0)");
            points[2] = new Point("SRID=4326;POINT(20 0)");
            sps[0] = new Position("market",points[0]);
            sps[1] = new Position("rent-a-car",points[1]);
            Point homepoint = new Point("SRID=4326;POINT(0 0)");
            Position home = new Position("home",homepoint);
            System.out.println("Persisting spatial data...");
            System.out.println(home);
            System.out.println(sps[0]);
            System.out.println(sps[1]);
            System.out.println(sps[2]);
            System.out.println(""");
            pm.makePersistentAll(sps);
            pm.makePersistent(home);
            tx.commit();

            //query for the distance
            tx.begin();
            Double distance = new Double(12.0);
            System.out.println("Retriving position where distance to home is less than " + distance + " ... Found:");
            Query query = pm.newQuery(Position.class, "name != 'home' && Spatial.distance(this.point, :homepoint) < :distance");
            List list = (List) query.execute(homepoint, distance);
            for (int i = 0; i < list.size(); i++)
            {
                System.out.println(list.get(i));
            }
            //clean up database.. just for fun :)
            pm.newQuery(Position.class).deletePersistentAll();
            tx.commit();
        }
        finally
        {
            if (tx.isActive())
            {
                tx.rollback();
            }
            pm.close();
        }
    }
}

©2015. DataNucleus • ALL RIGHTS RESERVED.
We define a *persistence.xml* file with connection properties to MySQL

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
version="1.0">
  <persistence-unit name="MyTest">
    <mapping-file>org/datanucleus/samples/jdo/spatial/package.jdo</mapping-file>
    <exclude-unlisted-classes />
    <properties>
      <property name="javax.jdo.option.ConnectionURL" value="jdbc:mysql://127.0.0.1/nucleus"/>
      <property name="javax.jdo.option.ConnectionDriverName" value="com.mysql.jdbc.Driver"/>
      <property name="javax.jdo.option.ConnectionUserName" value="mysql"/>
      <property name="javax.jdo.option.ConnectionPassword" value=""/>

      <property name="datanucleus.autoCreateSchema" value="true"/>
      <property name="datanucleus.autoCreateColumns" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

120.1.6 Step 5: Run your application

Before running the application, you must *enhance* the persistent classes. Finally, configure the application classpath with the DataNucleus Core, DataNucleus RDBMS, DataNucleus Spatial, JDO2, MySQL and PostGis libraries and run the application as any other Java application.

The output for the application is:

```
DataNucleus JDO Spatial Sample
================================
Persisting spatial data...
[name] home [point] SRID=4326;POINT(0 0)
[name] market [point] SRID=4326;POINT(5 0)
[name] rent-a-car [point] SRID=4326;POINT(10 0)
[name] pizza shop [point] SRID=4326;POINT(20 0)

Retrieving position where X position is > 10 and Y position is 0 ... Found:
[name] pizza shop [point] SRID=4326;POINT(20 0)

End of Sample
```
121 DAO Layer Design

121.1 DataNucleus - Design of a DAO Layer with JDO

121.1.1 Introduction
The design of an application will involve many choices, and often compromises. What is generally accepted as good practice is to layer the application tiers and provide interfaces between these. For example a typical web application can have 3 tiers - web tier, business-logic tier, and data-access tier. DataNucleus provides data persistence and so, following this model, should only be present in the data access layer. One pattern to achieve this is the data access object (DAO) pattern.

A typical DAO provides an interface that defines its contract with the outside world. This takes the form of a series of data access and data update methods. In this tutorial we follow this and describe how to implement a DAO using DataNucleus and JDO using some of the new features introduced in JDO 2.0.

While this guide demonstrates how to write a DAO layer, it does not propose use of the DAO pattern, just explaining how you could achieve it.

121.1.2 The DAO contract
To highlight our strategy for DAO's we introduce 3 simple classes.
So we have 3 dependent classes, and we have a 1-N relationship between Owner and Pet, and a N-1 relationship between Pet and PetType.

We now generate an outline DAO object containing the main methods that we expect to need

```java
public interface ClinicDAO {
    public Collection<Owner> getOwners();
    public Collection<PetType> getPetTypes();
    public CollectionOWNER findOwners(String lastName);
    public Owner loadOwner(long id);
    public void storeOwner(Owner owner);
    public void storePet(Pet pet);
}
```

Clearly we could have defined more methods, but these will demonstrate the basic operations performed in a typical application. Note that we defined our DAO as an interface. This has various benefits, and the one we highlight here is that we can now provide an implementation using
DataNucleus and JDO. We could, in principle, provide a DAO implementation of this interface using JDBC for example, or one for whatever persistence technology. It demonstrates a flexible design strategy allowing components to be swapped at a future date.

We now define an outline DAO implementation using DataNucleus. We will implement just a few of the methods defined in the interface, just to highlight the style used. So we choose one method that retrieves data and one that stores data.
public class MyDAO implements ClinicDAO
{
    PersistenceManagerFactory pmf;

    /** Constructor, defining the PersistenceManagerFactory to use. */
    public MyDAO(PersistenceManagerFactory pmf)
    {
        this.pmf = pmf;
    }

    /** Accessor for a PersistenceManager */
    protected PersistenceManager getPersistenceManager()
    {
        return pmf.getPersistenceManager();
    }

    public Collection<Owner> getOwners()
    {
        Collection owners = null;
        PersistenceManager pm = getPersistenceManager();
        Transaction tx = pm.currentTransaction();
        try
        {
            tx.begin();

            Query q = pm.newQuery(mydomain.model.Owner.class);
            Collection query_owners = q.execute();

            // *** TODO Copy "query_owners" into "owners" ***

            tx.commit();
        }
        finally
        {
            if (tx.isActive())
            {
                tx.rollback();
            }
            pm.close();
        }
        return owners;
    }

    public void storeOwner(Owner owner)
    {
        PersistenceManager pm = getPersistenceManager();
        Transaction tx = pm.currentTransaction();
        try
        {
            tx.begin();

            // Owner is new, so persist it
            if (owner.id() == null)
            {
                pm.makePersistent(owner);
            }

            // Owner exists, so update it
            else
            {
                // *** TODO Store the updated owner ***
            }
        }
        finally
        {
            if (tx.isActive())
            {
                tx.rollback();
            }
            pm.close();
        }
    }
}
So here we've seen the typical DAO and how, for each method, we retrieve a PersistenceManager, obtain a transaction, and perform our operation(s). Notice above there are a couple of places where we have left "TODO" comments. These will be populated in the next section, using the JDO 2.0 feature attach/detach.

121.1.3 Use of attach/detach

We saw in the previous section our process for the DAO methods. The problem we have with JDO 1.0 is that as soon as we leave the transaction our object would move back to "Hollow" state (hence losing its field values, and hence the object would have been unusable in the rest of our application. With JDO 2.0 we have a feature called attach/detach that allows us to detach objects for use elsewhere, and then attach them when we want to persist any changed data within the object. So we now go back to our DataNucleus DAO and add the necessary code to use this
public Collection<Owner> getOwners()
{
    Collection<Owner> owners;
    PersistenceManager pm = getPersistenceManager();
    Transaction tx = pm.currentTransaction();
    try
    {
        tx.begin();

        Query q = pm.newQuery(mydomain.model.Owner.class);
        Collection query_owners = q.execute();

        // Detach our owner objects for use elsewhere
        owners = pm.detachCopyAll(query_owners);

        tx.commit();
    }
    finally
    {
        if (tx.isActive())
        {
            tx.rollback();
        }
        pm.close();
    }
    return owners;
}

public void storeOwner(Owner owner)
{
    PersistenceManager pm = getPersistenceManager();
    Transaction tx = pm.currentTransaction();
    try
    {
        tx.begin();

        // Persist our changes back to the datastore
        pm.makePersistent(owner);

        tx.commit();
    }
    finally
    {
        if (tx.isActive())
        {
            tx.rollback();
        }
        pm.close();
    }
}
So we have added 2 very simple method calls. These facilitate making our objects usable outside the DAO layer and so give us much flexibility. Please note that instead of detachCopy you could set the PMF option "javax.jdo.option.DetachAllOnCommit" and this would silently migrate all enlisted instances to detached state at commit of the transaction, and is probably a more convenient way of detaching.

121.1.4 Definition of fetch-groups

In the previous section we have described how to design our basic DAO layer. We have an interface to this layer and provide a DataNucleus implementation. The DataNucleus/JDO calls are restricted to the DAO layer. What we haven't yet considered is what actually is made usable in the rest of the application when we do our detach. By default with this feature PersistenceCapable fields will not be detached with the owning object. This means that our "pets" field of our detached "owner" object will not be available for use. In many situations we would want to give access to other parts of our object. To do this we make use of another JDO 2.0 feature, called fetch-groups. This is defined both in the Meta-Data for our classes, and in the DAO layer where we perform the detaching. Let's start with the MetaData for our 3 classes.

```
<class name="Owner" detachable="true">
  <field name="id" primary-key="true"/>
  <field name="pets" mapped-by="owner">
    <collection element-type="mydomain.model.Pet"/>
  </field>
  <fetch-group name="detach_owner_pets">
    <field name="pets"/>
  </fetch-group>
</class>
<class name="PetType" detachable="true">
  <field name="id" primary-key="true"/>
  <field name="name">
    <column length="80" jdbc-type="VARCHAR"/>
  </field>
</class>
<class name="Pet" detachable="true">
  <field name="id" primary-key="true"/>
  <field name="name">
    <column length="30" jdbc-type="VARCHAR"/>
  </field>
  <field name="type" persistence-modifier="persistent"/>
  <fetch-group name="detach_pet_type">
    <field name="type"/>
  </fetch-group>
</class>
```

Here we've marked the classes as detachable, and added a fetch-group to Owner for the "pets" field, and to Pet for the "type" field. Doing these for each field adds extra flexibility to our ability to specify them. Lets now update our DAO layer method using detach to use these fetch-groups so that when we use the detached objects in our application, they have the necessary components available.
public Collection getOwners()
{
    Collection owners;
    PersistenceManager pm = getPersistenceManager();
    Transaction tx = pm.currentTransaction();
    try
    {
        tx.begin();

        Query q = pm.newQuery(mydomain.model.Owner.class);
        Collection query_owners = q.execute();

        // Define the objects to be detached with our owner objects.
        pm.getFetchPlan().addGroup("detach_owner_pets");
        pm.getFetchPlan().addGroup("detach_pet_type");
        pm.getFetchPlan().setMaxFetchDepth(3);

        // Detach our owner objects for use elsewhere
        owners = pm.detachCopyAll(query_owners);

        tx.commit();
    }
    finally
    {
        if (tx.isActive())
        {
            tx.rollback();
        }
        pm.close();
    }
    return owners;
}

So you see that when we detach our Owner objects, we also detach the Pet objects for each owner and for each Pet object we will also detach the PetType object. This means that we can access all of these objects with no problem outside of the DAO layer.

121.1.5 Summary

In this tutorial we've demonstrated a way of separating the persistence code from the rest of the application by providing a DAO layer interface. We've demonstrated how to write the associated methods within this layer and, with the use of attach/detach and fetch-groups we can make our persisted objects available outside of the DAO layer in a seamless way. Developers of the remainder of the system don't need to know any details of the persistence, they simply code to the interface contract provided by the DAO.
122 JPA API

122.1 JPA : API

JPA defines an interface (or API) to persist normal Java objects (or POJO's in some peoples terminology) to an RDBMS datastore. While DataNucleus allows you to use JPA against any of its supported datastores if you are intent on using JPA for persistence to a non-RDBMS datastore we highly recommend that you think deeply about that decision, and consider JDO instead since the design of JPA and in particular JPQL force assumptions to be made in how the persistence/query process operates.

Note that this version of DataNucleus requires JPA 2.1 API

JPA uses a definition of how the users Java objects map to the chosen datastore structure. This mapping can be provided by way of XML metadata, or alternatively by having Java annotations in the code. The whole point of having a standard mapping and API is that users can, in principle, swap between implementations of JPA without changing their code. Make sure you have datanucleus-api-jpa.jar in your CLASSPATH for this API. The process of mapping a class can be split into the following areas

- The first thing to do is to mark the classes that are to be persisted as such
- JPA allows fields/properties to be defined for persistence, and you can control which of these are persisted, and how they are persisted.
- Since JPA is oriented to RDBMS datastores only you now need to define the Object-Relational Mapping (ORM)

Note that with DataNucleus, you can map your classes using JDO MetaData (XML/Annotations) OR using JPA MetaData (XML/Annotations) and still use the JPA API with these classes.

At runtime the JPA code can be split into several sections.

- You firstly need to create an EntityManagerFactory to connect to a datastore
- You then need to create an EntityManager to provide the interface to persisting/accessing objects
- Controlling the transaction
- Accessing persisted object via queries, using JPQL, or SQL

If in doubt about how things fit together, please make use of the JPA Tutorial

If you just want to get the JPA API javadocs, then you can access those here

122.1.1 JPA References

- JPA 2.1 Specification
- JPA 2.1 Javadocs
- JPA Group mailing lists
- ORM comparison : JDO .v. JPA
123 Class Mapping

123.1 JPA : Class Mapping

The first thing to decide when implementing your persistence layer is which classes are to be persisted. If you need to persist a field/property then you must mark that class as persistable. In JPA there are three types of persistable classes.

- **Entity** - persistable class with full control over its persistence.
- **MappedSuperclass** - persistable class that will not be persisted into its own table simply providing some fields to be persisted. Consequently an inheritance tree cannot just have a mapped superclass on its own. [Read more](#)
- **Embeddable** - persistable class that is only persistable embedded into an entity class. [Read more](#)

Let's take a sample class (`Hotel`) as an example. We can define a class as persistable using either annotations in the class, or XML metadata.

To achieve the above aim with XML metadata, we do this:

```xml
<entity class="org.datanucleus.test.Hotel">
  ...
</entity>
```

Alternatively, using JPA Annotations, like this:

```java
@Entity
public class Hotel {
  ...
}
```

In the above example we have marked the class as an **entity**. We could equally have marked it as **mapped-superclass** (using annotation `@MappedSuperclass`, or XML element `<mapped-superclass>`) or as **embeddable** (using annotation `@Embeddable`, or XML element `<embeddable>`).

See also:

- JPA XML reference
- JPA Annotations reference

123.1.1 Persistence Aware

With JPA you cannot access public fields of classes. DataNucleus allows an extension to permit this, but such classes need special enhancement. To allow this you need to:

- Annotate the class that will access these public fields (assuming it isn't an Entity) with the DataNucleus extension annotation `@PersistenceAware`

You perform the annotation of the class as follows:

```java
@PersistenceAware
public class Hotel {
  ...
}
```
@PersistenceAware
public class MyClassThatAccessesPublicFields
{
    ...
}

See also :-

- Annotations reference for @PersistenceAware

### 123.1.2 Read-Only

You can, if you wish, make a class *read-only*. This is a DataNucleus extension and you set it as follows

```java
@Extension
@Entity
@Extension(vendorName="datanucleus", key="read-only", value="true")
public class MyClass
{
    ...
}
```
124 Application Identity

124.1 JPA : Application Identity

With application identity you are taking control of the specification of id's to DataNucleus. Application identity requires a primary key class (unless using SingleFieldIdentity, where one is provided for you), and each persistent capable class may define a different class for its primary key, and different persistent capable classes can use the same primary key class, as appropriate. With application identity the field(s) of the primary key will be present as field(s) of the class itself. To specify that a class is to use application identity, you add the following to the MetaData for the class.

```xml
<entity class="org.mydomain.MyClass">
  <id-class class="org.mydomain.MyIdClass"/>
  <attributes>
    <id name="myPrimaryKeyField"/>
  </attributes>
</entity>
```

For JPA we specify the id field and id-class. Alternatively, if we are using annotations

```java
@Entity
@IdClass(class=MyIdClass.class)
public class MyClass
{
    @Id
    private long myPrimaryField;
}
```

When you have an inheritance hierarchy, you should specify the identity type in the base instantiable class for the inheritance tree. This is then used for all persistent classes in the tree. This means that you can have MappedSuperclass without any identity fields/properties as superclass, and then the base instantiable class is the first persistable class which has the identity field(s). This is a change from DataNucleus 2.2 where you had to have identity fields in the base persistable class of the inheritance tree.

See also :

- MetaData reference for <id> element
- Annotations reference for @Id

124.1.1 Primary Key

Using application identity requires the use of a Primary Key class. With JPA when you have a single-field you don't need to provide a primary key class. Where the class has multiple fields that form the primary key a Primary Key class must be provided (via the id-class).

See also :

- Primary Key Guide - user-defined and built-in primary keys
124.1.2 Generating identities

By choosing **application identity** you are controlling the process of identity generation for this class. This does not mean that you have a lot of work to do for this. JPA defines many ways of generating these identities and DataNucleus supports all of these and provides some more of its own besides.

See also :-
- **Identity Generation Guide** - strategies for generating ids

124.1.3 Changing Identities

JPA doesn't define what happens if you change the identity (an identity field) of an object once persistent. **DataNucleus doesn't currently support changes to identities.**

124.1.4 Accessing objects by Identity

You access an object from its object class name and identity "value" as follows:

```java
Object obj = em.find(MyClass.class, mykey);
```

If you have defined your own "IdClass" then the `mykey` is the toString() form if the identity of your PK class.

124.2 JPA : PrimaryKey Classes

When you choose application identity you are defining which fields of the class are part of the primary key, and you are taking control of the specification of id's to DataNucleus. Application identity requires a primary key (PK) class, and each persistent capable class may define a different class for its primary key, and different persistent capable classes can use the same primary key class, as appropriate. If you have only a single primary-key field then there are builtin PK classes so you can forget this section. Where you have more than 1 primary key field, you would define the PK class like this:

```xml
<entity class="MyClass">
  <id-class class="MyIdClass"/>
  ...
</entity>
```

or using annotations

```java
@Entity
@IdClass(class=MyIdClass.class)
public class MyClass {
  ...
}
```

You now need to define the PK class to use. This is simplified for you because **if you have only one PK field then you dont need to define a PK class** and you only define it when you have a composite PK.

An important thing to note is that the PK can only be made up of fields of the following Java types
- Primitives : boolean, byte, char, int, long, short
• java.lang: **Boolean, Byte, Character, Integer, Long, Short, String, Enum, StringBuffer**
• java.math: **BigInteger**
• java.sql: **Date, Time, Timestamp**
• java.util: **Date, Currency, Locale, TimeZone, UUID**
• java.net: **URI, URL**
• javax.jdo.spi: **PersistenceCapable**

Note that the types in **bold** are JPA standard types. Any others are DataNucleus extensions and, as always, check the specific datastore docs to see what is supported for your datastore.

### 124.2.1 Single PrimaryKey field

The simplest way of using application identity is where you have a single PK field, and in this case you use an inbuilt primary key class that DataNucleus provides, so you don't need to specify the `objectid-class`. Let's take an example

```java
public class MyClass {
    long id;
    ...
}
```

```xml
<entity class="MyClass">
    <attributes>
        <id name="id"/>
    ...
</attributes>
</entity>
```

or using annotations

```java
@Entity
public class MyClass {
    @Id
    long id;
    ...
}
```

So we didn't specify the JPA "id-class". You will, of course, have to give the field a value before persisting the object, either by setting it yourself, or by using a value-strategy on that field.

### 124.2.2 Multiple PrimaryKey field

Since there are many possible combinations of primary-key fields it is impossible for JPA to provide a series of builtin composite primary key classes. However the DataNucleus enhancer provides a mechanism for auto-generating a primary-key class for a persistable class. It follows the rules listed below and should work for all cases. Obviously if you want to tailor the output of things like the PK
toString() method then you ought to define your own. The enhancer generation of primary-key class is only enabled if you don't define your own class.

124.2.3 Rules for User-Defined PrimaryKey classes
If you wish to use application identity and don't want to use the "SingleFieldIdentity" builtin PK classes then you must define a Primary Key class of your own. You can't use classes like java.lang.String, or java.lang.Long directly. You must follow these rules when defining your primary key class.

- the Primary Key class must be public
- the Primary Key class must implement Serializable
- the Primary Key class must have a public no-arg constructor, which might be the default constructor
- The PrimaryKey class can have a constructor taking the primary key fields, or can use Java bean setters/getters
- the field types of all non-static fields in the Primary Key class must be serializable, and are recommended to be primitive, String, Date, or Number types
- all serializable non-static fields in the Primary Key class can be public, but package/protected/private should also be fine
- the names of the non-static fields in the Primary Key class must include the names of the primary key fields in the Entity, and the types of the common fields must be identical
- the equals() and hashCode() methods of the Primary Key class must use the value(s) of all the fields corresponding to the primary key fields in the JDO class
- if the Primary Key class is an inner class, it must be static
- the Primary Key class must override the toString() method defined in Object, and return a String that can be used as the parameter of a constructor
- the Primary Key class must provide a String constructor that returns an instance that compares equal to an instance that returned that String by the toString() method.
- the Primary Key class must be only used within a single inheritance tree.

Please note that if one of the fields that comprises the primary key is in itself a persistable object then you have Compound Identity and should consult the documentation for that feature which contains its own example.

124.2.4 PrimaryKey Example - Multiple Field
Here's an example of a composite (multiple field) primary key class
@Entity
@IdClass(ComposedIdKey.class)
public class MyClass
{
    @Id
    String field1;

    @Id
    String field2;
    ...
}

public class ComposedIdKey implements Serializable
{
    public String field1;
    public String field2;

    /**
     * Default constructor.
     */
    public ComposedIdKey()
    {
    }

    /**
     * Constructor accepting same input as generated by toString().
     */
    public ComposedIdKey(String value)
    {
        StringTokenizer token = new StringTokenizer(value, "::");
        //field1
        this.field1 = token.nextToken();
        //field2
        this.field2 = token.nextToken();
    }

    public boolean equals(Object obj)
    {
        if (obj == this)
        {
            return true;
        }
        if (!(obj instanceof ComposedIdKey))
        {
            return false;
        }
        ComposedIdKey c = (ComposedIdKey)obj;
        return field1.equals(c.field1) && field2.equals(c.field2);
    }

    public int hashCode()
    {
        return this.field1.hashCode() ^ this.field2.hashCode();
    }

    public String toString()
    {
        // Give output expected by String constructor
        return "" + this.field1 + "::" + this.field2;
    }
}
125 Datastore Identity

125.1 JPA: Datastore Identity

While JPA defines support for application identity only DataNucleus also provides support for datastore identity. With datastore identity you are leaving the assignment of id's to DataNucleus and your class will not have a field for this identity - it will be added to the datastore representation by DataNucleus. It is, to all extents and purposes a surrogate key that will have its own column in the datastore. To specify that a class is to use datastore identity with JPA, you define the following annotations on your class

```java
@Entity
@org.datanucleus.api.jpa.annotations.DatastoreIdentity
public class MyClass {
    ...
}
```

Please note that since the JPA XML metadata is poorly designed it is not possible to specify datastore identity using XML, you have to use the annotations.

When you have an inheritance hierarchy, you should specify the identity type in the base class for the inheritance tree. This is then used for all persistent classes in the tree.

125.1.1 Generating identities

By choosing datastore identity you are handing the process of identity generation to the DataNucleus. This does not mean that you haven't got any control over how it does this. JPA defines many ways of generating these identities and DataNucleus supports all of these and provides some more of its own besides.

Defining which one to use is a simple matter of adding a MetaData element to your classes definition, like this

```java
@Entity
@org.datanucleus.api.jpa.annotations.DatastoreIdentity(generationType=GenerationType.TABLE)
public class MyClass {
    ...
}
```

See also:

- Identity Generation Guide - strategies for generating ids
- Annotations reference for @DatastoreIdentity
125.1.2 Accessing the Identity

When using **datastore identity**, the class has no associated field so you can't just access a field of the class to see its identity - if you need a field to be able to access the identity then you should be using **application identity**. There are, however, ways to get the identity for the datastore identity case, if you have the object.

```java
Object id = JDOHelper.getObjectId(obj);
```

You should be aware however that the "identity" is in a complicated form, and is not available as a simple integer value for example. Again, if you want an identity of that form then you should use **application identity**

125.1.3 DataNucleus Implementation

When implementing **datastore identity** DataNucleus provides a public class that represents this identity. If you call `JDOHelper.getObjectId(...)` for a class using datastore identity you will be passed an object which, in the case of DataNucleus will be of type `org.datanucleus.identity.OIDImpl`. If you were to call "toString()" on this object you would get something like

```java
1[OID]mydomain.myclass
```

This is made up of :-

- 1 = identity number of this object
- class-name

DataNucleus allows you the luxury of being able to **provide your own datastore identity class** so you can have whatever formatting you want for identities.

125.1.4 Accessing objects by Identity

You can also access the object from the object class name and the toString() form of the datastore identity (e.g "101[OID]mydomain.MyClass") like this

```java
Object obj = em.find(MyClass.class, mykey);
```

or in "datanucleus-api-jpa" 3.0.7+ you can also just pass in the "key" value, like this

```java
Object obj = em.find(MyClass.class, 101);
```
126 Compound Identity

126.1 JPA: Compound Identity Relationships
An identifying relationship (or "compound identity relationship") is a relationship between two objects of two classes in which the child object must coexist with the parent object and where the primary key of the child includes the Entity object of the parent. So effectively the key aspect of this type of relationship is that the primary key of one of the classes includes a Entity field (hence why is is referred to as Compound Identity). This type of relation is available in the following forms

- 1-1 unidirectional
- 1-N collection bidirectional using ForeignKey
- 1-N map bidirectional using ForeignKey (key stored in value)

126.1.1 1-1 Relationship
Let's take the same classes as we have in the 1-1 Relationships. In the 1-1 relationships guide we note that in the datastore representation of the User and Account the ACCOUNT table has a primary key as well as a foreign-key to USER. In our example here we want to just have a primary key that is also a foreign-key to USER. To do this we need to modify the classes slightly and add primary-key fields and use "application-identity".

In addition we need to define primary key classes for our User and Account classes.
public class User
{
    long id;

    ... (remainder of User class)

    /**
    * Inner class representing Primary Key
    */
    public static class PK implements Serializable
    {
        public long id;

        PK()
        {
        }

        PK(String s)
        {
            this.id = Long.valueOf(s).longValue();
        }

        public String toString()
        {
            return "" + id;
        }

        public int hashCode()
        {
            return (int)id;
        }

        public boolean equals(Object other)
        {
            if (other != null && (other instanceof PK))
            {
                PK otherPK = (PK)other;
                return otherPK.id == this.id;
            }
            return false;
        }
    }
}

public class Account
{
    User user;

    ... (remainder of Account class)

    /**
    * Inner class representing Primary Key
    */
    public static class PK implements Serializable
    {
        public User.PK user; // Use same name as the real field above

        PK()
        {
        }

        PK(String s)
        {
            StringTokenizer token = new StringTokenizer(s, "::");
            this.user = new User.PK(token.nextToken());
        }

        public String toString()
        {
            return this.user.toString();
        }

        public int hashCode()
        {
            return user.hashCode();
        }

        public boolean equals(Object other)
        {
            if (other != null && (other instanceof PK))
            {
                PK otherPK = (PK)other;
                return this.user.equals(otherPK.user);
            }
            return false;
        }
    }
}
To achieve what we want with the datastore schema we define the MetaData like this

```
<entity-mappings>
  <entity class="mydomain.User">
    <table name="USER"/>
    <id-class class="mydomain.User.PK"/>
    <attributes>
      <id name="id">
        <column name="USER_ID"/>
      </id>
      <basic name="login">
        <column name="LOGIN" length="20"/>
      </basic>
    </attributes>
  </entity>
  <entity class="mydomain.Account">
    <table name="ACCOUNT"/>
    <id-class class="mydomain.Account.PK"/>
    <attributes>
      <id name="user">
        <column name="USER_ID"/>
      </id>
      <basic name="firstName">
        <column name="FIRSTNAME" length="50"/>
      </basic>
      <basic name="secondName">
        <column name="LASTNAME" length="50"/>
      </basic>
      <one-to-one name="user"/>
    </attributes>
  </entity>
</entity-mappings>
```

So now we have the following datastore schema

```
USER
+USER_ID
LOGIN

ACCOUNT
+USER_ID
FIRSTNAME
LASTNAME
```

Things to note :-

- In the child Primary Key class, you must have a field with the same name as the relationship in the child class, and the field in the child Primary Key class must be the same type as the Primary Key class of the parent
- See also the general instructions for Primary Key classes
• You can only have one "Account" object linked to a particular "User" object since the FK to the "User" is now the primary key of "Account". To remove this restriction you could also add a "long id" to "Account" and make the "Account.PK" a composite primary-key

126.1.2 1-N Collection Relationship

Let's take the same classes as we have in the 1-N Relationships (FK). In the 1-N relationships guide we note that in the datastore representation of the Account and Address classes the ADDRESS table has a primary key as well as a foreign-key to ACCOUNT. In our example here we want to have the primary-key to ACCOUNT to include the foreign-key. To do this we need to modify the classes slightly, adding primary-key fields to both classes, and use "application-identity" for both.

In addition we need to define primary key classes for our Account and Address classes
public class Account
{
    long id; // PK field

    Set addresses = new HashSet();

    ... (remainder of Account class)

    /**
     * Inner class representing Primary Key
     */
    public static class PK implements Serializable
    {
        public long id;

        public PK()
        {
        }

        public PK(String s)
        {
            this.id = Long.valueOf(s).longValue();
        }

        public String toString()
        {
            return "" + id;
        }

        public int hashCode()
        {
            return (int)id;
        }

        public boolean equals(Object other)
        {
            if (other != null && (other instanceof PK))
            {
                PK otherPK = (PK)other;
                return otherPK.id == this.id;
            }
            return false;
        }
    }
}

public class Address
{
    long id;
    Account account;

    .. (remainder of Address class)

    /**
     * Inner class representing Primary Key
     */
    public static class PK implements Serializable
    {
        public long id; // Same name as real field above
        public Account.PK account; // Same name as the real field above
To achieve what we want with the datastore schema we define the MetaData like this

```
<entity-mappings>
  <entity class="mydomain.Account">
    <table name="ACCOUNT"/>
    <id-class class="mydomain.Account.PK"/>
    <attributes>
      <id name="id">
        <column name="ACCOUNT_ID"/>
      </id>
      <basic name="firstName">
        <column name="FIRSTNAME" length="50"/>
      </basic>
      <basic name="secondName">
        <column name="LASTNAME" length="50"/>
      </basic>
      <one-to-many name="addresses" mapped-by="account"/>
    </attributes>
  </entity>

  <entity class="mydomain.Address">
    <table name="ADDRESS"/>
    <id-class class="mydomain.Address.PK"/>
    <attributes>
      <id name="id">
        <column name="ID"/>
      </id>
      <id name="account">
        <column name="ACCOUNT_ID"/>
      </id>
      <basic name="city">
        <column name="CITY"/>
      </basic>
      <basic name="street">
        <column name="STREET"/>
      </basic>
      <many-to-one name="account"/>
    </attributes>
  </entity>
</entity-mappings>
```

So now we have the following datastore schema

![Datastore Schema Diagram]

Things to note:-
• In the child Primary Key class, you must have a field with the same name as the relationship in the child class, and the field in the child Primary Key class must be the same type as the Primary Key class of the parent
• See also the general instructions for Primary Key classes
• If we had omitted the "id" field from "Address" it would have only been possible to have one "Address" in the "Account" "addresses" collection due to PK constraints. For that reason we have the "id" field too.

126.1.3 1-N Map Relationship

Lets take the same classes as we have in the 1-N Relationships (FK). In this guide we note that in the datastore representation of the Account and Address classes the ADDRESS table has a primary key as well as a foreign-key to ACCOUNT. In our example here we want to have the primary-key to ACCOUNT to include the foreign-key. To do this we need to modify the classes slightly, adding primary-key fields to both classes, and use "application-identity" for both.

In addition we need to define primary key classes for our Account and Address classes
126 Compound Identity

public class Account
{
long id; // PK field
Set addresses = new HashSet();
... (remainder of Account class)
/**
* Inner class representing Primary Key
*/
public static class PK implements Serializable
{
public long id;
public PK()
{
}
public PK(String s)
{
this.id = Long.valueOf(s).longValue();
}
public String toString()
{
return "" + id;
}
public int hashCode()
{
return (int)id;
}
public boolean equals(Object other)
{
if (other != null && (other instanceof PK))
{
PK otherPK = (PK)other;
return otherPK.id == this.id;
}
return false;
}
}
}
public class Address
{
String alias;
Account account;
.. (remainder of Address class)

©2015,

/**
* Inner class representing Primary Key
*/
public static class PK implements Serializable
DataNucleus • ALL RIGHTS RESERVED.
{
public String alias; // Same name as real field above
public Account.PK account; // Same name as the real field above

751


To achieve what we want with the datastore schema we define the MetaData like this

```
<entity-mappings>
  <entity class="mydomain.Account">
    <table name="ACCOUNT"/>
    <id-class class="mydomain.Account.PK"/>
    <attributes>
      <id name="id">
        <column name="ACCOUNT_ID"/>
      </id>
      <basic name="firstName">
        <column name="FIRSTNAME" length="50"/>
      </basic>
      <basic name="secondName">
        <column name="LASTNAME" length="50"/>
      </basic>
      <one-to-many name="addresses" mapped-by="account">
        <map-key name="alias"/>
      </one-to-many>
    </attributes>
  </entity>
  <entity class="mydomain.Address">
    <table name="ADDRESS"/>
    <id-class class="mydomain.Address.PK"/>
    <attributes>
      <id name="account">
        <column name="ACCOUNT_ID"/>
      </id>
      <id name="alias">
        <column name="KEY"/>
      </id>
      <basic name="city">
        <column name="CITY"/>
      </basic>
      <basic name="street">
        <column name="STREET"/>
      </basic>
      <many-to-one name="account"/>
    </attributes>
  </entity>
</entity-mappings>
```

So now we have the following datastore schema
Things to note :-

- In the child Primary Key class, you must have a field with the same name as the relationship in the child class, and the field in the child Primary Key class must be the same type as the Primary Key class of the parent.
- See also the general instructions for Primary Key classes.
- If we had omitted the "alias" field from "Address" it would have only been possible to have one "Address" in the "Account" "addresses" collection due to PK constraints. For that reason we have the "alias" field too as part of the PK.
127 Versioning

127.1 JPA : Versioning

JPA allows objects of classes to be versioned. The version is typically used as a way of detecting if the object has been updated by another thread or EntityManager since retrieval using the current EntityManager - for use by Optimistic Transactions.

127.1.1 Version Field/Property

JPAs mechanism for versioning of objects is to mark a field of the class to store the version. The field must be Integer/Long based. With JPA you can specify the details of this version field as follows.

```xml
<entity name="mydomain.User">
    <attributes>
        <id name="id"/>
        <version name="version"/>
    </attributes>
</entity>
```

or alternatively using annotations

```java
@Entity
public class User {
    @Id
    long id;

    @Version
    int version;

    ...
}
```

The specification above will use the "version" field for storing the version of the object. DataNucleus will use a "version-number" strategy for populating the value.

127.1.2 Surrogate Version for Class

While the above mechanism should always be used for portability, DataNucleus also supports a surrogate version for objects of a class. With this you don't have a particular field that stores the version and instead DataNucleus persists the version in the datastore with the field values in its own "column" You do this as follows.
import org.datanucleus.api.jpa.annotations.SurrogateVersion;

@Entity
@SurrogateVersion
public class User {
  @Id
  long id;
  ...
}
128 Inheritance

128.1 JPA: Inheritance Strategies

In Java it is a normal situation to have inheritance between classes. With JPA you have choices to make as to how you want to persist your classes for the inheritance tree. For each inheritance tree (for the root class) you select how you want to persist those classes information. You have the following choices.

1. The default strategy is to select a class to have its fields persisted in the table of the base class. There is only one table per inheritance hierarchy. In JPA this is known as SINGLE_TABLE.

2. The next way is to have a table for each class in the inheritance hierarchy, and for each table to only hold columns for the fields of that class. Fields of superclasses are persisted into the table of the superclass. Consequently to get all field values for a subclass object a join is made of all tables of superclasses. In JPA this is referred to as JOINED.

3. The third way is like JOINED except that each table will also contain columns for all inherited fields. In JPA this is referred to as TABLE_PER_CLASS.

In order to demonstrate the various inheritance strategies we need an example. Here are a few simple classes representing products in a (online) store. We have an abstract base class, extending this to provide something that we can represent any product by. We then provide a few specialisations for typical products. We will use these classes later when defining how to persistent these objects in the different inheritance strategies.
The default JPA strategy is "SINGLE_TABLE", namely that the base class will have a table and all subclasses will be persisted into that same table. So if you don't specify an "inheritance strategy" in your root class this is what you will get.

Please note that you must specify the identity of objects in the root persistable class of the inheritance hierarchy. You cannot redefine it down the inheritance tree.

See also:

- MetaData reference for <inheritance> element
- MetaData reference for <discriminator-column> element
- Annotations reference for @Inheritance
- Annotations reference for @DiscriminatorColumn
128.1.1 Discriminator

Applicable to RDBMS, HBase, MongoDB

A discriminator is an extra "column" stored alongside data to identify the class of which that information is part. It is useful when storing objects which have inheritance to provide a quick way of determining the object type on retrieval. A discriminator in JPA will store the a specified value (or the class name if you provide no value). You specify a discriminator as follows

```xml
<entity name="mydomain.Product">
  <discriminator-column name="OBJECT" discriminator-type="STRING"/>
  <discriminator-value>MyClass</discriminator-value>
</entity>
```

or with annotations

```java
@Entity
@DiscriminatorColumn(name="OBJECT_TYPE", discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("MyClass")
public class Product {...}
```

128.1.2 SINGLE_TABLE

Applicable to RDBMS

"SINGLE_TABLE" strategy is where the root class has a table and all subclasses are also persisted into that table. This corresponds to JDOs "new-table" for the root class and "superclass-table" for all subclasses. This has the advantage that retrieval of an object is a single DB call to a single table. It also has the disadvantage that the single table can have a very large number of columns, and database readability and performance can suffer, and additionally that a discriminator column is required.

In our example, lets ignore the AbstractProduct class for a moment and assume that Product is the base class (with the "id"). We have no real interest in having separate tables for the Book and CompactDisc classes and want everything stored in a single table PRODUCT. We change our MetaData as follows
<entity name="Product">
  <inheritance strategy="SINGLE_TABLE"/>
  <discriminator-value>PRODUCT</discriminator-value>
  <discriminator-column name="PRODUCT_TYPE" discriminator-type="STRING"/>
  <attributes>
    <id name="id">
      <column name="PRODUCT_ID"/>
    </id>
    <basic name="price">
      <column name="PRICE"/>
    </basic>
  </attributes>
</entity>

<entity name="Book">
  <discriminator-value>BOOK</discriminator-value>
  <attributes>
    <basic name="isbn">
      <column name="ISBN"/>
    </basic>
    <basic name="author">
      <column name="AUTHOR"/>
    </basic>
    <basic name="title">
      <column name="TITLE"/>
    </basic>
  </attributes>
</entity>

<entity name="TravelGuide">
  <discriminator-value>TRAVELGUIDE</discriminator-value>
  <attributes>
    <basic name="country">
      <column name="COUNTRY"/>
    </basic>
  </attributes>
</entity>

<entity name="CompactDisc">
  <discriminator-value>COMPACTDISC</discriminator-value>
  <attributes>
    <basic name="artist">
      <column name="ARTIST"/>
    </basic>
    <basic name="title">
      <column name="DISCTITLE"/>
    </basic>
  </attributes>
</entity>

or using annotations
This change of use of the `inheritance` element has the effect of using the PRODUCT table for all classes, containing the fields of `Product`, `Book`, `CompactDisc`, and `TravelGuide`. You will also note that we used a `discriminator-column` element for the `Product` class. The specification above will result in an extra column (called `PRODUCT_TYPE`) being added to the PRODUCT table, and containing the "discriminator-value" of the object stored. So for a Book it will have "BOOK" in that column for example. This column is used in discriminating which row in the database is of which type. The final thing to note is that in our classes `Book` and `CompactDisc` we have a field that is identically named. With `CompactDisc` we have defined that its column will be called `DISCTITLE` since both of these fields will be persisted into the same table and would have had identical names otherwise - this gets around the problem.

In the above example, when we insert a TravelGuide object into the datastore, a row will be inserted into the PRODUCT table only.

128.1.3 JOINED

"JOINED" strategy means that each entity in the inheritance hierarchy has its own table and that the table of each class only contains columns for that class. Inherited fields are persisted into the tables of the superclass(es). This corresponds to JDO2s "new-table" (for all classes in the inheritance hierarchy). This has the advantage of being the most normalised data definition. It also has the disadvantage of being slower in performance since multiple tables will need to be accessed to retrieve an object of a sub-type. Let's try an example using the simplest to understand strategy JOINED. We have the classes defined above, and we want to persist our classes each in their own table. We define the Meta-Data for our classes like this
<entity class="AbstractProduct">
   <inheritance strategy="JOINED"/>
   <attributes>
      <id name="id">
         <column name="PRODUCT_ID"/>
      </id>
      <basic name="name">
         <column name="NAME"/>
      </basic>
      <basic name="description">
         <column name="DESCRIPTION"/>
      </basic>
   </attributes>
</entity>

<entity class="Product">
   <attributes>
      <basic name="price">
         <column name="PRICE"/>
      </basic>
   </attributes>
</entity>

<entity class="Book">
   <attributes>
      <basic name="isbn">
         <column name="ISBN"/>
      </basic>
      <basic name="author">
         <column name="AUTHOR"/>
      </basic>
      <basic name="title">
         <column name="TITLE"/>
      </basic>
   </attributes>
</entity>

<entity class="TravelGuide">
   <attributes>
      <basic name="country">
         <column name="COUNTRY"/>
      </basic>
   </attributes>
</entity>

<entity class="CompactDisc">
   <attributes>
      <basic name="artist">
         <column name="ARTIST"/>
      </basic>
      <basic name="title">
         <column name="TITLE"/>
      </basic>
   </attributes>
</entity>
or using annotations

```java
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Product {...}
```

So we will have 5 tables - ABSTRACTPRODUCT, PRODUCT, BOOK, COMPACTDISC, and TRAVELGUIDE. They each contain just the fields for that class (and not any inherited fields, except the identity to join with).

In the above example, when we insert a TravelGuide object into the datastore, a row will be inserted into ABSTRACTPRODUCT, PRODUCT, BOOK, and TRAVELGUIDE.

128.1.4 TABLE_PER_CLASS

Applicable to all datastores

This strategy is like "JOINED" except that in addition to each class having its own table, the table also holds columns for all inherited fields. So taking the same classes as used above
<entity class="AbstractProduct">
    <inheritance strategy="TABLE_PER_CLASS"/>
    <attributes>
        <id name="id">
            <column name="PRODUCT_ID"/>
        </id>
        <basic name="name">
            <column name="NAME"/>
        </basic>
        <basic name="description">
            <column name="DESCRIPTION"/>
        </basic>
    </attributes>
</entity>
<entity class="Product">
    <attributes>
        <basic name="price">
            <column name="PRICE"/>
        </basic>
    </attributes>
</entity>
<entity class="Book">
    <attributes>
        <basic name="isbn">
            <column name="ISBN"/>
        </basic>
        <basic name="author">
            <column name="AUTHOR"/>
        </basic>
        <basic name="title">
            <column name="TITLE"/>
        </basic>
    </attributes>
</entity>
<entity class="TravelGuide">
    <attributes>
        <basic name="country">
            <column name="COUNTRY"/>
        </basic>
    </attributes>
</entity>
<entity class="CompactDisc">
    <attributes>
        <basic name="artist">
            <column name="ARTIST"/>
        </basic>
        <basic name="title">
            <column name="TITLE"/>
        </basic>
    </attributes>
</entity>
or using annotations

```java
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Product {...}
```

This then implies a datastore schema as follows

- **Product**
  - +PRODUCT_ID
  - NAME
  - DESCRIPTION

- **Book**
  - +PRODUCT_ID
  - NAME
  - DESCRIPTION
  - ISBN
  - AUTHOR
  - TITLE

- **CompactDisc**
  - +PRODUCT_ID
  - NAME
  - DESCRIPTION
  - ARTIST
  - TITLE

So any object of explicit type **Book** is persisted into the table BOOK. Similarly any **TravelGuide** is persisted into the table TRAVELGUIDE, etc. In addition if any class in the inheritance tree is abstract then it won't have a table since there cannot be any instances of that type. **DataNucleus currently has limitations when using a class using this inheritance as the element of a collection.**

### 128.1.5 Mapped Superclasses

JPA defines entities called "mapped superclasses" for the situation where you don't persist an actual object of a superclass type but that all subclasses of that type that are entities will also persist the values for the fields of the "mapped superclass". That is a "mapped superclass" has no table to store its objects in a datastore. Instead its fields are stored in the tables of its subclasses. Let's take an example...
In this case we will have a table for **Product** and the fields of **AbstractProduct** will be stored in this table. If the mapping information (column names etc) for these fields need setting then you should use `<attribute-override>` in the MetaData for **Product**.
129 Fields/Properties

129.1 JPA : Persistent Fields or Properties
Now that we have defined the class as persistable we need to define how to persist the different fields/properties that are to be persisted. Please note that JPA cannot persist static or final fields. There are two distinct modes of persistence definition; the most common uses fields, whereas an alternative uses properties.

129.1.1 Persistent Fields
The most common form of persistence is where you have a field in a class and want to persist it to the datastore. With this mode of operation DataNucleus will persist the values stored in the fields into the datastore, and will set the values of the fields when extracting it from the datastore.

Requirement: you have a field in the class. This can be public, protected, private or package access, but cannot be static or final.

An example of how to define the persistence of a field is shown below

```java
@Entity
public class MyClass
{
    @Basic
    Date birthday;

    @Transient
    String someOtherField;
}
```

So, using annotations, we have marked this class as persistent, and the field `birthday` also as persistent, whereas field `someOtherField` is not persisted. Using XML MetaData we would have done

```xml
<entity name="mydomain.MyClass">
    <attributes>
        <basic name="birthday"/>
        <transient name="someOtherField"/>
    </attributes>
</entity>
```

Please note that the field Java type defines whether it is, by default, persistable. Look at the Types Guide and if the type has a tick in the column "Persistent?" then you can omit the "basic" specification.

129.1.2 Persistent Properties
A second mode of operation is where you have Java Bean-style getter/setter for a property. In this situation you want to persist the output from `getXXX` to the datastore, and use the `setXXX` to load up the value into the object when extracting it from the datastore.

Requirement: you have a property in the class with Java Bean getter/setter methods. These methods can be public, protected, private or package access, but cannot be static. The class must have BOTH getter AND setter methods.
An example of how to define the persistence of a property is shown below

```java
@Entity
public class MyClass {
    @Basic
    Date getBirthday()
    {
        ...
    }

    void setBirthday(Date date)
    {
        ...
    }
}
```

So, using annotations, we have marked this class as persistent, and the getter is marked as persistent. By default a property is non-persistent, so we have no need in specifying the `someOtherField` as transient. Using XML MetaData we would have done

```xml
<entity name="mydomain.MyClass">
    <attributes>
        <basic name="birthday"/>
    </attributes>
</entity>
```

### 129.1.3 Field/Property positioning

With some datastores (notably spreadsheets) it is desirable to be able to specify the relative position of a column. The default (for DataNucleus) is just to put them in ascending alphabetical order. JPA doesn’t allow configuration of this, but DataNucleus provides the following vendor extension. It is currently only possible using (DataNucleus) annotations

```java
@Entity
@Table(name="People")
public class Person {
    @Id
    @ColumnPosition(0)
    long personNum;

    @ColumnPosition(1)
    String firstName;

    @ColumnPosition(2)
    String lastName;
}
```
130 Java Types

130.1 JPA: Persistable Java Types

When persisting a class, a persistence solution needs to know how to persist the types of each field in the class. Clearly a persistence solution can only support a finite number of Java types; it cannot know how to persist every possible type creatable. The JPA specification define lists of types that are required to be supported by all implementations of those specifications. This support can be conveniently split into two parts

130.1.1 First-Class (FCO) Types

An object that can be referred to (object reference, providing a relation) and that has an "identity" is termed a First Class Object (FCO). DataNucleus supports the following Java types as FCO

- **persistable**: any class marked for persistence (annotations or XML) can be persisted with its own identity in the datastore
- **interface** where the field represents a persistable object
- **java.lang.Object** where the field represents a persistable object

130.1.2 Supported Second-Class (SCO) Types

An object that does not have an "identity" is termed a Second Class Object (SCO). This is something like a String or Date field in a class, or alternatively a Collection (that contains other objects). The table below shows the currently supported SCO java types in DataNucleus. The table shows

- **Extension?**: whether the type is JPA standard, or is a DataNucleus extension
- **EAGER**: whether the field is retrieved by default when retrieving the object itself (this equates to the "default-fetch-group" concept in JDO)
- **persistence-modifier**: whether the field is persisted by default, or whether the user has to mark the field as persistent in XML/annotations to persist it
- **proxied**: whether the field is represented by a "proxy" that intercepts any operations to detect whether it has changed internally.
- **primary-key**: whether the field can be used as part of the primary-key

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>datanucleus-core</td>
</tr>
<tr>
<td>byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>datanucleus-core</td>
</tr>
<tr>
<td>char</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>datanucleus-core</td>
</tr>
<tr>
<td>double</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>datanucleus-core</td>
</tr>
<tr>
<td>float</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>datanucleus-core</td>
</tr>
<tr>
<td>int</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>datanucleus-core</td>
</tr>
<tr>
<td>long</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>datanucleus-core</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean[]</td>
<td>Byte[]</td>
<td>Char[]</td>
<td>Double[]</td>
<td>Float[]</td>
<td>Int[]</td>
</tr>
<tr>
<td>------------</td>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
<td>----------</td>
<td>---------</td>
<td>-----------</td>
</tr>
<tr>
<td>short</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>boolean[]</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>byte[]</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>char[]</td>
<td></td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>double[]</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>float[]</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int[]</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>long[]</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>short[]</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Boolean</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Byte</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Character</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Double</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Float</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Integer</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Long</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Short</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Boolean[]</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Byte[]</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Character[]</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Double[]</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Float[]</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Integer[]</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Long[]</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Short[]</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class/Membership</td>
<td>Extension</td>
<td>Compatibility</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>-----------</td>
<td>---------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.lang.Long[]</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.lang.Short[]</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.lang.Object</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.lang.String</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.lang.String</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.lang.Class</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.math.BigDecimal</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.math.BigInteger</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.sql.Date</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.sql.Time</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.sql.Timestamp</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.util.ArrayList</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.util.BitSet</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.util.Calendar</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.util.Collection</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.util.Currency</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.util.Date</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.util.Date[7]</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.util.GregorianCalendar</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.util.LinkedList</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.util.List</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>java.util.Vector</code></td>
<td>![Extension]</td>
<td>![Compat]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** The compatibility status is represented by icons: ![Compat] indicates compatibility, and ![Compat] indicates incompatibility.
<table>
<thead>
<tr>
<th>Class</th>
<th>130 Java Types</th>
<th>© 2015, DataNucleus • ALL RIGHTS RESERVED.</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.util.HashMap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.HashSet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Hashtable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.LinkedHashMap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.LinkedHashSet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.List</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Locale</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Locale[]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Map</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Properties</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.PriorityQueue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Queue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.SortedMap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.SortedSet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.Stack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.TimeZone</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.TreeMap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.TreeSet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.util.UUID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.awt.Color</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2015, DataNucleus • ALL RIGHTS RESERVED.
<table>
<thead>
<tr>
<th>Class</th>
<th>Extension</th>
<th>javanucleus-core</th>
<th>javanucleus-geospatial</th>
<th>javanucleus-core</th>
<th>javanucleus-core</th>
<th>javanucleus-core</th>
<th>javanucleus-core</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.awt.image.BufferedImage</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.awt.Point</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.awt.Rectangle</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.net.URI</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.net.URL</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.io.Serializable</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Enum</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>javax.time.calendar.LocalDateTime</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>javax.time.calendar.LocalTime</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>javax.time.calendar.LocalDate</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.joda.time.DateTime</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.joda.time.LocalTime</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.joda.time.Period</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.joda.time.Interval</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>com.google.common.collect.Multiset</td>
<td><img src="image" alt="Extension" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- [1] javanucleus.jar, javanucleus.TreeSet allow the specification of comparators via the "comparator-name" DataNucleus extension MetaData element (within <collection>). The headSet, tailSet, subSet methods are only supported when using cached collections.
• [2] - `java.util.SortedMap, java.util.TreeMap` allow the specification of comparators via the "comparator-name" DataNucleus extension MetaData element (within `<map>`). The headMap, tailMap, subMap methods are only supported when using cached containers.

• [3] - `java.lang.StringBuffer` dirty check mechanism is limited to immutable mode, it means, if you change a StringBuffer object field, you must reassign it to the owner object field to make sure changes are propagated to the database.

• [4] - `java.lang.Number` will be stored in a column capable of storing a BigDecimal, and will store to the precision of the object to be persisted. On reading back the object will be returned typically as a BigDecimal since there is no mechanism for determining the type of the object that was stored.

• [5] - `java.util.LinkedHashMap` treated as a Map currently. No List-ordering is supported.

• [6] - `java.util.LinkedHashSet` treated as a Set currently. No List-ordering is supported.

• [7] - `java.util.Calendar` can be stored into two columns (milliseconds, Timezone) or into a single column (Timestamp). The single column option is not guaranteed to preserve the TimeZone of the input Calendar.

• [8] - available only for RDBMS, persisted into LONGVARBINARY, and retrieved as streamable so as not to adversely affect memory utilisation, hence suitable for large files. New in 3.2.8 of datanucleus-rdbms

Note that support is available for persisting other types depending on the datastore to which you are persisting

• RDBMS GeoSpatial types via the DataNucleus RDBMS Spatial plugin

If you have support for any additional types and would either like to contribute them, or have them listed here, let us know

130.1.3 JPA Attribute Converters

JPA2.1 introduces an API for conversion of an attribute of an Entity to its datastore value. You can define a "converter" that will convert to the datastore value and back from it, implementing this interface.

```java
public interface AttributeConverter<X, Y>
{
    public Y convertToDatabaseColumn (X attributeObject);

    public X convertToEntityAttribute (Y dbData);
}
```

so if we have a simple converter to allow us to persist fields of type URL in a String form in the datastore, like this
public class URLStringConverter implements AttributeConverter<URL, String> {
    public URL convertToEntityAttribute(String str) {
        if (str == null) {
            return null;
        }
        URL url = null;
        try {
            url = new java.net.URL(str.trim());
        } catch (MalformedURLException mue) {
            throw new IllegalStateException("Error converting the URL", mue);
        }
        return url;
    }
    public String convertToDatabaseColumn(URL url) {
        return url != null ? url.toString() : null;
    }
}

and now in our Entity class we mark any URL field as being converted using this converter

@Entity
public class MyClass {
    @Id
    long id;

    @Basic
    @Convert(converter=URLStringConverter.class)
    URL url;

    ...
}

Note that in strict JPA 2.1 you have to mark all converters with the @Converter annotation. In DataNucleus if you specify the converter class name in the @Convert then we know its a converter so don’t really see why we need a user to annotate the converter too. We only require annotation as @Converter if you want the converter to always be applied to fields of a particular type i.e if you want all URL fields to be persisted using the above converter (without needing to put @Convert on each field of that type) then you would add the annotation
The only other point to make is that if you have some java type with a @Converter registered to "autoApply", you can turn it off on a field-by-field basis with

```java
@Convert(disableConversion=true)
URL url;
```

### 130.1.4 Eclipse EMF models

You could try to persist Eclipse EMF models using the Texo project to generate POJOs
131 Value Generation

131.1 JPA : Value Generation

Fields of a class can either have the values set by you the user, or you can set DataNucleus to generate them for you. This is of particular importance with identity fields where you want unique identities. You can use this value generation process with the identity field(s) in JPA. There are many different "strategies" for generating values, as defined by the JPA specification. Some strategies are specific to a particular datastore, and some are generic. You should choose the strategy that best suits your target datastore. The available strategies are :-

- AUTO - this is the default and allows DataNucleus to choose the most suitable for the datastore
- SEQUENCE - this uses a datastore sequence (if supported by the datastore)
- IDENTITY - these use autoincrement/identity/serial features in the datastore (if supported by the datastore)
- TABLE - this is datastore neutral and increments a sequence value using a table.

See also :-

- JPA MetaData reference for <generated-value>
- JPA Annotation reference for @GeneratedValue

Please note that the JPA spec only requires the ability to generate values for identity fields. DataNucleus allows you to do it for any field. Please bear this in mind when considering portability.

Please note that by defining a value-strategy for a field then it will, by default, always generate a value for that field on persist. If the field can store nulls and you only want it to generate the value at persist when it is null (i.e you haven’t assigned a value yourself) then you can add the extension “strategy-when-null” as false

131.1.1 AUTO

With this strategy DataNucleus will choose the most appropriate strategy for the datastore being used. If you define the field as String-based then it will choose uuid-hex. Otherwise the field is numeric in which case it chooses identity if supported, otherwise sequence if supported, otherwise increment if supported otherwise throws an exception. On RDBMS you can get the behaviour used up until DN v3.0 by specifying the persistence property datanucleus.rdbms.useLegacyNativeValueStrategy as true For a class using application identity you need to set the value-strategy attribute on the primary key field. You can configure the Meta-Data for the class something like this

```xml
<entity class="MyClass">
  <attributes>
    <id name="myId">
      <generated-value strategy="AUTO"/>
    </id>
  </attributes>
</entity>
```

or using annotations
A sequence is a user-defined database function that generates a sequence of unique numeric ids. The unique identifier value returned from the database is translated to a java type: java.lang.Long. DataNucleus supports sequences for the following datastores:

- Oracle
- PostgreSQL
- SAP DB
- DB2
- Firebird
- HSQLDB
- H2
- Derby (from v10.6)
- SQLServer (from v2012)
- NuoDB

To configure a class to use either of these generation methods using application identity you would add the following to the class' Meta-Data

```xml
<sequence-generator name="SEQ1" sequence-name="MY_SEQ" initial-value="5" allocation-size="10"/>
<entity class="MyClass">
  <attributes>
    <id name="myId">
      <generated-value strategy="SEQUENCE" generator="SEQ1"/>
    </id>
  </attributes>
</entity>
```

or using annotations

```java
@Entity
@SequenceGenerator(name="SEQ1", sequenceName="MY_SEQ", initialValue=5, allocationSize=10)
public class MyClass
{
  @Id
  @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="SEQ1")
  private long myId;
  ...
}
```
If the sequence does not yet exist in the database at the time DataNucleus needs a new unique identifier, a new sequence is created in the database based on the JPA Meta-Data configuration. Additional properties for configuring sequences are set in the JPA Meta-Data, see the available properties below. Unsupported properties by a database are silently ignored by DataNucleus.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>key-database-cache-size</td>
<td>specifies how many sequence numbers are to be preallocated and stored in memory for faster access. This is an optimization feature provided by the database</td>
<td>No</td>
</tr>
<tr>
<td>sequence-catalog-name</td>
<td>Name of the catalog where the sequence is.</td>
<td>No</td>
</tr>
<tr>
<td>sequence-schema-name</td>
<td>Name of the schema where the sequence is.</td>
<td>No</td>
</tr>
</tbody>
</table>

This value generator will generate values unique across different JVMs.

### 131.1.3 IDENTITY

Auto-increment/identity/serial are primary key columns that are populated when a row is inserted in the table. These use the databases own keywords on table creation and so rely on having the table structure either created by DataNucleus or having the column with the necessary keyword.

DataNucleus supports auto-increment/identity/serial keys for many databases including:

- DB2 (IDENTITY)
- MySQL (AUTOINCREMENT)
- MSSQL (IDENTITY)
- Sybase (IDENTITY)
- HSQLDB (IDENTITY)
- H2 (IDENTITY)
- PostgreSQL (SERIAL)
- Derby (IDENTITY)
- MongoDB - String based
- Neo4j - long based
- NuoDB (IDENTITY)

This generation strategy should only be used if there is a single "root" table for the inheritance tree. If you have more than 1 root table (e.g using subclass-table inheritance) then you should choose a different generation strategy.

For a class using application identity you need to set the `value-strategy` attribute on the primary key field. You can configure the Meta-Data for the class something like this:
Please be aware that if you have an inheritance tree with the base class defined as using "identity" then the column definition for the PK of the base table will be defined as "AUTO_INCREMENT" or "IDENTITY" or "SERIAL" (dependent on the RDBMS) and all subtables will NOT have this identifier added to their PK column definitions. This is because the identities are assigned in the base table (since all objects will have an entry in the base table).

Please note that if using optimistic transactions, this strategy will mean that the value is only set when the object is actually persisted (i.e at flush() or commit())

This value generator will generate values unique across different JVMs

### 131.1.4 TABLE

This method is database neutral and uses a sequence table that holds an incrementing sequence value. The unique identifier value returned from the database is translated to a java type: java.lang.Long. This strategy will work with any datastore. This method require a sequence table in the database and creates one if doesn't exist.

To configure an application identity class to use this generation method you simply add this to the class' Meta-Data. If your class is in an inheritance tree you should define this for the base class only.

```xml
<entity class="MyClass">
  <attributes>
    <id name="myId">
      <generated-value strategy="TABLE"/>
    </id>
  </attributes>
</entity>
```

or using annotations

```java
@Entity
public class MyClass {
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private long myId;
  ...
}
```
```java
@Entity
public class MyClass {
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE)
    private long myId;
    ...
}
```

Additional properties for configuring this generator are set in the JPA Meta-Data, see the available properties below. Unsupported properties are silently ignored by DataNucleus.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>key-initial-value</td>
<td>First value to be allocated.</td>
<td>No. Defaults to 1</td>
</tr>
<tr>
<td>key-cache-size</td>
<td>number of unique identifiers to cache. The keys are pre-allocated, cached and used on demand. If key-cache-size is greater than 1, it may generate holes in the object keys in the database, if not all keys are used.</td>
<td>No. Default is 50</td>
</tr>
<tr>
<td>sequence-table-basis</td>
<td>Whether to define uniqueness on the base class name or the base table name. Since there is no “base table name” when the root class has “subclass-table” this should be set to “class” when the root class has “subclass-table” inheritance.</td>
<td>No. Defaults to class, but the other option is table</td>
</tr>
<tr>
<td>sequence-name</td>
<td>name for the sequence (overriding the &quot;sequence-table-basis&quot; above). The row in the table will use this in the PK column.</td>
<td>No</td>
</tr>
<tr>
<td>sequence-table-name</td>
<td>Table name for storing the sequence.</td>
<td>No. Defaults to SEQUENCE_TABLE</td>
</tr>
<tr>
<td>sequence-catalog-name</td>
<td>Name of the catalog where the table is.</td>
<td>No.</td>
</tr>
<tr>
<td>sequence-schema-name</td>
<td>Name of the schema where the table is.</td>
<td>No.</td>
</tr>
<tr>
<td>sequence-name-column-name</td>
<td>Name for the column that represent sequence names.</td>
<td>No. Defaults to SEQUENCE_NAME</td>
</tr>
<tr>
<td>sequence-nextval-column-name</td>
<td>Name for the column that represent incrementing sequence values.</td>
<td>No. Defaults to NEXT_VAL</td>
</tr>
<tr>
<td>table-name</td>
<td>Name of the table whose column we are generating the value for (used when we have no previous sequence value and want a start point.</td>
<td>No.</td>
</tr>
<tr>
<td>column-name</td>
<td>Name of the column we are generating the value for (used when we have no previous sequence value and want a start point.)</td>
<td>No.</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------------------------------------------------------------------------------------------------</td>
<td>-----</td>
</tr>
</tbody>
</table>

This value generator will generate values unique across different JVMs.
132 Embedded Fields

132.1 JPA : Embedded Fields

The JPA persistence strategy typically involves persisting the fields of any class into its own table, and representing any relationships from the fields of that class across to other tables. There are occasions when this is undesirable, maybe due to an existing datastore schema, or because a more convenient datastore model is required. JPA allows the persistence of fields as embedded typically into the same table as the "owning" class.

One important decision when defining objects of a type to be embedded into another type is whether objects of that type will ever be persisted in their own right into their own table, and have an identity. JPA provides a MetaData attribute that you can use to signal this.

```xml
<embeddable name="mydomain.MyClass">
...
</embeddable>
```

or using annotations

```java
@Embeddable
public class MyClass
{
    ...
}
```

With the above MetaData (using the embeddable definition), in our application any objects of the class MyClass can be embedded into other objects.

JPA's definition of embedding encompasses several types of fields. These are described below

- **Embedded Entities** - where you have a 1-1 relationship and you want to embed the other Entity into the same table as the your object.
- **Embedded Nested Entities** - like the first example except that the other object also has another Entity that also should be embedded
- **Embedded Collection elements** - where you want to embed the elements of a collection into a join table (instead of persisting them into their own table)

132.1.1 Embedding entities (1-1)

Applicable to RDBMS, Excel, OOXML, ODF, HBase, MongoDB, Neo4j

In a typical 1-1 relationship between 2 classes, the 2 classes in the relationship are persisted to their own table, and a foreign key is managed between them. With JPA and DataNucleus you can persist the related entity object as embedded into the same table. This results in a single table in the datastore rather than one for each of the 2 classes.

Let's take an example. We are modelling a Computer, and in our simple model our Computer has a graphics card and a sound card. So we model these cards using a ComputerCard class. So our classes become
The traditional (default) way of persisting these classes would be to have a table to represent each class. So our datastore will look like this:

```
<table>
<thead>
<tr>
<th>COMPUTER</th>
<th>COMPUTER_CARD</th>
</tr>
</thead>
<tbody>
<tr>
<td>+COMPUTER_ID</td>
<td>+COMPUTERCARD_ID</td>
</tr>
<tr>
<td>OS_NAME</td>
<td>MANUFACTURER</td>
</tr>
<tr>
<td>#GRAPHICSCARD_ID</td>
<td>TYPE</td>
</tr>
<tr>
<td>#SOUNDCARD_ID</td>
<td></td>
</tr>
</tbody>
</table>
```
However we decide that we want to persist **Computer** objects into a table called COMPUTER and we also want to persist the PC cards into the same table. We define our MetaData like this:

```xml
<entity name="mydomain.Computer">
  <attributes>
    <basic name="operatingSystem">
      <column="OS_NAME"/>
    </basic>
    <embedded name="graphicsCard">
      <attribute-override name="manufacturer">
        <column="GRAPHICS_MANUFACTURER"/>
      </attribute-override>
      <attribute-override name="type">
        <column="GRAPHICS_TYPE"/>
      </attribute-override>
    </embedded>
    <embedded name="soundCard">
      <attribute-override name="manufacturer">
        <column="SOUND_MANUFACTURER"/>
      </attribute-override>
      <attribute-override name="type">
        <column="SOUND_TYPE"/>
      </attribute-override>
    </embedded>
  </attributes>
</entity>

<embeddable name="mydomain.ComputerCard">
  <attributes>
    <basic name="manufacturer"/>
    <basic name="type"/>
  </attributes>
</embeddable>
```

So here we will end up with a TABLE called “COMPUTER” with columns “COMPUTER_ID”, “OS_NAME”, “GRAPHICS_MANUFACTURER”, “GRAPHICS_TYPE”, “SOUND_MANUFACTURER”, “SOUND_TYPE”. If we call persist() on any objects of type **Computer**, they will be persisted into this table.

It should be noted that in this latter (embedded) case we can still persist objects of type **ComputerCard** into their own table - the MetaData definition for **ComputerCard** is used for the table definition in this case.
DataNucleus supports embedded persistable objects with the following proviso :-

- You can represent inheritance of embedded objects using a discriminator (you must define it in the metadata of the embedded type. Note that this is a DataNucleus extension since JPA doesn’t define any support for embedded inherited persistable objects

See also :-

- MetaData reference for <embedded> element
- Annotations reference for @Embeddable
- Annotations reference for @Embedded

132.1.2 Embedding Nested Entities

Applicable to RDBMS, Excel, OOXML, ODF, HBase, MongoDB, Neo4j.

In the above example we had an embedded Entity within a persisted object. What if our embedded PersistenceCapable object also contain another Entity object. So, using the above example what if ComputerCard contains an object of type Connector?

```java
@Embeddable
public class ComputerCard {
    ...
    @Embedded
    Connector connector;
    
    public ComputerCard(String manufacturer, int type, Connector conn) {
        this.manufacturer = manufacturer;
        this.type = type;
        this.connector = conn;
    }
    ...
}

@Embeddable
public class Connector {
    int type;
}
```

Well we want to store all of these objects into the same record in the COMPUTER table.
So we simply nest the embedded definition of the Connector objects within the embedded definition of the ComputerCard definitions for Computer. JPA supports this to as many levels as you require! The Connector objects will be persisted into the GRAPHICS_CONNECTOR_TYPE, and SOUND_CONNECTOR_TYPE columns in the COMPUTER table.
132.1.3 Embedding Collection Elements

In a typical 1-N relationship between 2 classes, the 2 classes in the relationship are persisted to their own table, and either a join table or a foreign key is used to relate them. With JPA and DataNucleus you have a variation on the join table relation where you can persist the objects of the "N" side into the join table itself so that they don't have their own identity, and aren't stored in the table for that class. This is supported in DataNucleus with the following provisos

- You can have inheritance in embedded keys/values and a discriminator is added (you must define the discriminator in the metadata of the embedded type).
- When retrieving embedded elements, all fields are retrieved in one call. That is, fetch plans are not utilised. This is because the embedded element has no identity so we have to retrieve all initially.

It should be noted that where the collection "element" is not PersistenceCapable or of a "reference" type (Interface or Object) it will always be embedded, and this functionality here applies to PersistenceCapable elements only. DataNucleus doesn't support the embedding of reference type objects currently.

Let's take an example. We are modelling a Network, and in our simple model our Network has collection of Devices. So we define our classes as
We decide that instead of `Device` having its own table, we want to persist them into the join table of its relationship with the `Network` since they are only used by the network itself. We define our MetaData like this:
So here we will end up with a table called "NETWORK" with columns "NETWORK_ID", and "NAME", and a table called "NETWORK_DEVICES" with columns "NETWORK_ID", "ADPT_PK_IDX", "DEVICE_NAME", "DEVICE_IP_ADDR". When we persist a Network object, any devices are persisted into the NETWORK_DEVICES table.

See also :-

- MetaData reference for <embeddable> element
- MetaData reference for <embedded> element
- MetaData reference for <element-collection> element
- MetaData reference for <collection-table> element
- Annotations reference for @Embeddable
- Annotations reference for @Embedded
- Annotations reference for @ElementCollection
133 Serialised Fields

133.1 JPA : Serialising Objects

JPA1 provides a way for users to specify that a field will be persisted *serialised*. This is of use, for example, to collections/maps/arrays which typically are stored using join tables or foreign-keys to other records. By specifying that a field is serialised a column will be added to store that field and the field will be serialised into it.

JPA’s definition of serialising applies to any field and all in the same way, unlike the situation with JDO which provides much more flexibility. Perhaps the most important thing to bear in mind when deciding to serialise a field is that that object in the field being serialised must implement `java.io.Serializable`.

133.1.1 Serialised Fields

Applicable to RDBMS, HBase, MongoDB

If you wish to serialise a particular field into a single column (in the table of the class), you need to simply mark the field as a "lob" (large object). Let's take an example. We have the following classes

![Diagram](image)

and we want the `animals` collection to be serialised into a single column in the table storing the `Farm` class, so we define our MetaData like this

```xml
<entity class="Farm">
    <table name="FARM"/>
    <attributes>
        ...
        <basic name="animals">
            <column name="ANIMALS"/>
            <lob/>
        </basic>
        ...
    </attributes>
</entity>
```

So we make use of the `lob` element (or @Lob annotation). This specification results in a table like this
Provisos to bear in mind are

- Queries cannot be performed on collections stored as serialised.

If the field that we want to serialise is of type String, byte[], char[], Byte[] or Character[] then the field will be serialised into a CLOB column rather than BLOB.

See also :-

- MetaData reference for <basic> element
- Annotations reference for @Lob

133.1.2 Serialised Field to Local File

Applicable to RDBMS

Note this is not part of the JPA spec, but is available in DataNucleus to ease your usage. If you have a non-relation field that implements Serializable you have the option of serialising it into a file on the local disk. This could be useful where you have a large file and don’t want to persist very large objects into your RDBMS. Obviously this will mean that the field is no longer queryable, but then if its a large file you likely don’t care about that. So let's give an example

```java
@Entity
public class Person
{
    @Id
    long id;

    @Basic
    @Lob
    @Extension(vendorName="datanucleus", key="serializeToFileLocation"
                            value="person_avatars")
    AvatarImage image;
}
```

Or using XML
So this will now persist a file into a folder `person_avatars` with filename as the String form of the identity of the owning object. In a real world example you likely will specify the extension value as an absolute path name, so you can place it anywhere in the local disk.
134 Interface Fields

134.1 JPA : Interface Fields

JPA doesn't define support for persisting fields of type interface, but DataNucleus provides an extension whereby the implementations of the interface are persistable objects. It follows the same general process as for `java.lang.Object` since both interfaces and `java.lang.Object` are basically references to some persistable object.

To demonstrate interface handling let's introduce some classes. Let's suppose you have an interface with a selection of classes implementing the interface something like this:

![Diagram of Shape, Circle, Rectangle, and Square classes]

You then have a class that contains an object of this interface type:

```java
public class ShapeHolder
{
    protected Shape shape = null;
    ...
}
```

DataNucleus allows the following strategies for mapping this field:

- **per-implementation**: a FK is created for each implementation so that the datastore can provide referential integrity. The other advantage is that since there are FKs then querying can be performed. The disadvantage is that if there are many implementations then the table can become large with many columns not used.

- **identity**: a single column is added and this stores the class name of the implementation stored, as well as the identity of the object. The advantage is that if you have large numbers of implementations then this can cope with no schema change. The disadvantages are that no querying can be performed, and that there is no referential integrity.

- **xcalia**: a slight variation on "identity" whereby there is a single column yet the contents of that column are consistent with what Xcalia XIC JDO implementation stored there.
The user controls which one of these is to be used by specifying the extension `mapping-strategy` on the field containing the interface. The default is "per-implementation".

In terms of the implementations of the interface, you can either leave the field to accept any known about implementation, or you can restrict it to only accept some implementations (see "implementation-classes" metadata extension). If you are leaving it to accept any persistable implementation class, then you need to be careful that such implementations are known to DataNucleus at the point of encountering the interface field. By this we mean, DataNucleus has to have encountered the metadata for the implementation so that it can allow for the implementation when handling the field. You can force DataNucleus to know about a persistable class by using an autostart mechanism, or using `persistence.xml`, or by placement of the `package.jdo` file so that when the owning class for the interface field is encountered so is the metadata for the implementations.

134.1.1 1-1

To allow persistence of this interface field with DataNucleus you have 2 levels of control. The first level is global control. Since all of our `Square`, `Circle`, `Rectangle` classes implement `Shape` then we just define them in the MetaData as we would normally.

```java
public class Square implements Shape {
    ...
}
public class Circle implements Shape {
    ...
}
public class Rectangle implements Shape {
    ...
}
```

The global way means that when mapping that field DataNucleus will look at all Entities it knows about that implement the specified interface.

DataNucleus also allows users to specify a list of classes implementing the interface on a field-by-field basis, defining which of these implementations are accepted for a particular interface field. To do this you define the Meta-Data like this:

```java
@Entity
public class ShapeHolder {
    @OneToOne
    @Extension(key="implementation-classes",
               value="mydomain.Circle,mydomain.Rectangle,mydomain.Square")
    Shape shape;
    ...
}
```

That is, for any interface object in a class to be persisted, you define the possible implementation classes that can be stored there. DataNucleus interprets this information and will map the above example classes to the following in the database.
So DataNucleus adds foreign keys from the containers table to all of the possible implementation tables for the *shape* field.

If we use **mapping-strategy** of "identity" then we get a different datastore schema.

```java
@Entity
public class ShapeHolder{
    @OneToOne
    @Extensions(
        @Extension(key="implementation-classes", 
            value="mydomain.Circle,mydomain.Rectangle,mydomain.Square"),
        @Extension(key="mapping-strategy", value="identity"))
    Shape shape;
    ...
}
```

and the datastore schema becomes
and the column "SHAPE" will contain strings such as `mydomain.Circle:1` allowing retrieval of the related implementation object.

134.1.2 1-N

You can have a Collection/Map containing elements of an interface type. You specify this in the same way as you would any Collection/Map. **You can have a Collection of interfaces as long as you use a join table relation and it is unidirectional.** The "unidirectional" restriction is that the interface is not persistent on its own and so cannot store the reference back to the owner object. Use the 1-N relationship guides for the metadata definition to use.

You need to use a DataNucleus extension tag "implementation-classes" if you want to restrict the collection to only contain particular implementations of an interface. For example

```java
@Entity
public class ShapeHolder
{
    @OneToMany
    @JoinTable
    @Extensions(
        @Extension(key="implementation-classes",
                  value="mydomain.Circle,mydomain.Rectangle,mydomain.Square"),
        @Extension(key="mapping-strategy", value="identity"))
    Collection<Shape> shapes;

    ...
}
```

So the `shapes` field is a Collection of `mydomain.Shape` and it will accept the implementations of type `Circle`, `Rectangle`, `Square` and `Triangle`. If you omit the implementation-classes tag then you have to give DataNucleus a way of finding the metadata for the implementations prior to encountering this field.
134.1.3 Dynamic Schema Updates

The default mapping strategy for interface fields and collections of interfaces is to have separate FK column(s) for each possible implementation of the interface. Obviously if you have an application where new implementations are added over time the schema will need new FK column(s) adding to match. This is possible if you enable the persistence property `datanucleus.rdbms.dynamicSchemaUpdates`, setting it to `true`. With this set, any insert/update operation of an interface related field will do a check if the implementation being stored is known about in the schema and, if not, will update the schema accordingly.
135 Object Fields

135.1 JPA : Fields of type java.lang.Object

JPA doesn't specify support for persisting fields of type java.lang.Object, however DataNucleus does support this where the values of that field are persistable objects themselves. This follows the same general process as for Interfaces since both interfaces and java.lang.Object are basically references to some persistable object.

Java.lang.Object cannot be used to persist non-persistable types with fixed schema datastore (e.g RDBMS). Think of how you would expect it to be stored if you think it ought to

DataNucleus allows the following ways of persisting Object fields :-

- **per-implementation** : a FK is created for each "implementation" so that the datastore can provide referential integrity. The other advantage is that since there are FKs then querying can be performed. The disadvantage is that if there are many implementations then the table can become large with many columns not used.
- **identity** : a single column is added and this stores the class name of the "implementation" stored, as well as the identity of the object. The disadvantages are that no querying can be performed, and that there is no referential integrity.
- **xcalia** : a slight variation on "identity" whereby there is a single column yet the contents of that column are consistent with what Xcalia XIC JDO implementation stored there.

The user controls which one of these is to be used by specifying the extension mapping-strategy on the field containing the interface. The default is "per-implementation"

135.1.1 1-1/N-1 relation

Let's suppose you have a field in a class and you have a selection of possible persistable class that could be stored there, so you decide to make the field a java.lang.Object. So let's take an example. We have the following class

```java
public class ParkingSpace
{
    String location;
    Object occupier;
}
```

So we have a space in a car park, and in that space we have an occupier of the space. We have some legacy data and so can't make the type of this "occupier" an interface type, so we just use java.lang.Object. Now we know that we can only have particular types of objects stored there (since there are only a few types of vehicle that can enter the car park). So we define our MetaData like this
This will result in the following database schema.

So DataNucleus adds foreign keys from the ParkingSpace table to all of the possible implementation tables for the \textit{occupier} field.

In conclusion, when using "per-implementation" mapping for any \texttt{java.lang.Object} field in a class to be persisted (as non-serialised), you must define the possible "implementation" classes that can be stored there.

If we use \texttt{mapping-strategy} of "identity" then we get a different datastore schema.

and the datastore schema becomes
and the column "OCCUPIER" will contain strings such as `com.mydomain.samples.object.Car:1` allowing retrieval of the related implementation object.

### 135.1.2 Collections of Objects

You can have a Collection/Map containing elements of `java.lang.Object`. You specify this in the same way as you would any Collection/Map. DataNucleus supports having a Collection of references with multiple implementation types as long as you use a join table relation.

### 135.1.3 Serialised Objects

By default a field of type `java.lang.Object` is stored as an instance of the underlying PersistenceCapable in the table of that object. If either your Object field represents non-PersistenceCapable objects or you simply wish to serialise the Object into the same table as the owning object, you need to specify the "serialized" attribute, like this

```java
public class MyClass
{
    @Lob
    Object myObject;
}
```

Please refer to the [serialised fields guide](#) for more details of storing objects in this way.
136 Array Fields

136.1 JPA : Array Fields

JPA defines support the persistence of arrays but only arrays of byte[], Byte[], char[], Character[]. Moreover it only defines support for persisting these arrays into CLOB columns, hence they have to be byte-streamed. Namely

- **Single Column** - the array is byte-streamed into a single column in the table of the containing object.

136.1.1 Single Column Arrays

Let's suppose you have a class something like this

```
Account

firstName: String
lastName: String
permissions: byte[]
```

So we have an `Account` and it has a number of permissions, each expressed as a byte. We want to persist the permissions in a single-column into the table of the account (but we don't want them serialised). We then define MetaData something like this

```
<entity class="Account">
  <table name="ACCOUNT"/>
  <attributes>
    ...
    <basic name="permissions">
      <column name="PERMISSIONS"/>
      <lob/>
    </basic>
    ...
  </attributes>
</entity>
```

This results in a datastore schema as follows

```
ACCOUNT
+ACCOUNT_ID
FIRST_NAME
LAST_NAME
PERMISSIONS
```

See also :-

- MetaData reference for `<basic>` element
• Annotations reference for @Basic
137 1-to-1 Relations

137.1 JPA : 1-1 Relationships

You have a 1-to-1 relationship when an object of a class has an associated object of another class (only one associated object). It could also be between an object of a class and another object of the same class (obviously). You can create the relationship in 2 ways depending on whether the 2 classes know about each other (bidirectional), or whether only one of the classes knows about the other class (unidirectional). These are described below.

137.1.1 Unidirectional

For this case you could have 2 classes, User and Account, as below.

![Diagram of User and Account classes with 1-to-1 relationship]

so the Account class knows about the User class, but not vice-versa. If you define the XML metadata for these classes as follows

```xml
<entity-mappings>
  <entity class="User">
    <table name="USER"/>
    <attributes>
      <id name="id">
        <column name="USER_ID"/>
      </id>
      ...
    </attributes>
  </entity>

  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      <id name="id">
        <column name="ACCOUNT_ID"/>
      </id>
      ...
      <one-to-one name="user">
        <join-column name="USER_ID"/>
      </one-to-one>
    </attributes>
  </entity>
</entity-mappings>
```
or alternatively using annotations

```java
public class Account {
    ...
    @OneToOne
    @JoinColumn(name="USER_ID")
    User user;
}

public class User {
    ...
}
```

This will create 2 tables in the database, one for `User` (with name `USER`), and one for `Account` (with name `ACCOUNT` and a column `USER_ID`), as shown below.

![Diagram showing USER and ACCOUNT tables]

Things to note:-

- `Account` has the object reference to `User` (and so is the "owner" of the relation) and so its table holds the foreign-key

- If you call `EntityManager.remove()` on the end of a 1-1 unidirectional relation without the relation and that object is related to another object, an exception will typically be thrown (assuming the RDBMS supports foreign keys). To delete this record you should remove the other objects association first.

### 137.1.2 Bidirectional

For this case you could have 2 classes, `User` and `Account` again, but this time as below. Here the `Account` class knows about the `User` class, and also vice-versa.

![Diagram showing User and Account classes and their relationship]

We create the 1-1 relationship with a single foreign-key. To do this you define the XML metadata as
or alternatively using annotations

```java
class Account
{
    ...
    @OneToOne
    @JoinColumn(name="USER_ID")
    User user;
}
class User
{
    ...
    @OneToOne(mappedBy="user")
    Account account;
    ...
}
```

The difference is that we added `mapped-by` to the field of `User` making it bidirectional (and putting the FK at the other side for RDBMS)
This will create 2 tables in the database, one for User (with name USER), and one for Account (with name ACCOUNT). For RDBMS it includes a USER_ID column in the ACCOUNT table, like this:

For other types of datastore it will have a USER_ID column in the ACCOUNT table and a ACCOUNT column in the USER table.

Things to note:

- When forming the relation please make sure that you set the relation at BOTH sides since DataNucleus would have no way of knowing which end is correct if you only set one end.
138 1-to-N Relations

138.1 JPA : 1-N Relationships

You have a 1-N (one to many) when you have one object of a class that has a Collection/Map of objects of another class. In the `java.util` package there are an assortment of possible collection/map classes and they all have subtly different behaviour with respect to allowing nulls, allowing duplicates, providing ordering, etc. There are two ways in which you can represent a collection or map in a datastore: Join Table (where a join table is used to provide the relationship mapping between the objects), and Foreign-Key (where a foreign key is placed in the table of the object contained in the collection or map).

We split our documentation based on what type of collection/map you are using.

- 1-N using Collection types
- 1-N using Set types
- 1-N using List type
- 1-N using Map type
139 Collections

139.1 JPA : 1-N Relationships with Collections

You have a 1-N (one to many) when you have one object of a class that has a Collection of objects of another class. Please note that Collections allow duplicates, and so the persistence process reflects this with the choice of primary keys. There are two ways in which you can represent this in a datastore: **Join Table** (where a join table is used to provide the relationship mapping between the objects), and **Foreign-Key** (where a foreign key is placed in the table of the object contained in the Collection).

The various possible relationships are described below.

- 1-N Unidirectional using Join Table
- 1-N Unidirectional using Foreign-Key
- 1-N Bidirectional using Join Table
- 1-N Bidirectional using Foreign-Key
- 1-N Unidirectional of non-persistable using Join Table

Please note that RDBMS supports the full range of options on this page, whereas other datastores (ODF, Excel, HBase, MongoDB, etc) persist the Collection in a column in the owner object rather than using join-tables or foreign-keys since those concepts are RDBMS-only.

139.1.1 equals() and hashCode()

Important: The element of a Collection ought to define the methods `equals` and `hashCode` so that updates are detected correctly. This is because any Java Collection will use these to determine equality and whether an element is contained in the Collection. Note also that the `hashCode()` should be consistent throughout the lifetime of a persistable object. By that we mean that it should not use some basis before persistence and then use some other basis (such as the object identity) after persistence, for this reason we do not recommend usage of `JDOHelper.getObjectId(obj)` in the equals/`hashCode` methods.

139.2 1-N Collection Unidirectional

We have 2 sample classes `Account` and `Address`. These are related in such a way as `Account` contains a `Collection` of objects of type `Address`, yet each `Address` knows nothing about the `Account` objects that it relates to. Like this:

![Diagram of Account and Address classes](image)

There are 2 ways that we can persist this relationship. These are shown below
### 139.2.1 Using Join Table

If you define the XML metadata for these classes as follows

```xml
<entity-mappings>
  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      <id name="id">
        <column name="ACCOUNT_ID"/>
      </id>
      ...
      <one-to-many name="addresses" target-entity="com.mydomain.Address">
        <join-table name="ACCOUNT_ADDRESSES">
          <join-column name="ACCOUNT_ID_OID"/>
          <inverse-join-column name="ADDRESS_ID_EID"/>
        </join-table>
      </one-to-many>
    </attributes>
  </entity>
  <entity class="Address">
    <table name="ADDRESS"/>
    <attributes>
      <id name="id">
        <column name="ADDRESS_ID"/>
      </id>
      ...
    </attributes>
  </entity>
</entity-mappings>
```

or alternatively using annotations

```java
public class Account {
  ...

  @OneToMany
  @JoinTable(name="ACCOUNT_ADDRESSES")
  @JoinColumn(name="ACCOUNT_ID_OID")
  @InverseJoinColumn(name="ADDRESS_ID_EID")
  Collection<Address> addresses
}

g

public class Address {
  ...
}
```

The crucial part is the `join-table` element on the field element - this signals to JPA to use a join table.
This will create 3 tables in the database, one for Address, one for Account, and a join table, as shown below.

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the `table` element below the `class` element.
- To specify the names of the columns where the fields of a class are stored, specify the `column` attribute on the `basic` element.
- To specify the name of the join table, specify the `join-table` element below the `one-to-many` element with the collection.
- To specify the names of the join table columns, use the `join-column` and `inverse-join-column` elements below the `join-table` element.
- The join table will NOT be given a primary key (since a Collection can have duplicates).

### 139.2.2 Using Foreign-Key

In this relationship, the Account class has a List of Address objects, yet the Address knows nothing about the Account. In this case we don't have a field in the Address to link back to the Account and so DataNucleus has to use columns in the datastore representation of the Address class. So we define the XML metadata like this
or alternatively using annotations

```java
public class Account {
    ...
    @OneToMany
    @JoinColumn(name="ACCOUNT_ID")
    Collection<Address> addresses
}

public class Address {
    ...
}
```

**Note that you MUST specify the join-column here otherwise it defaults to a join table with JPA!**

There will be 2 tables, one for **Address**, and one for **Account**. If you wish to specify the names of the column(s) used in the schema for the foreign key in the Address table you should use the `join-column` element within the field of the collection.
In terms of operation within your classes of assigning the objects in the relationship. You have to take your Account object and add the Address to the Account collection field since the Address knows nothing about the Account.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the `table` element below the `class` element
- To specify the names of the columns where the fields of a class are stored, specify the `column` attribute on the `basic` element.

**Limitation**: Since each Address object can have at most one owner (due to the "Foreign Key") this mode of persistence will not allow duplicate values in the Collection. If you want to allow duplicate Collection entries, then use the "Join Table" variant above.

### 139.3 1-N Collection Bidirectional

We have 2 sample classes **Account** and **Address**. These are related in such a way as **Account** contains a **Collection** of objects of type **Address**, and each **Address** has a reference to the **Account** object that it relates to. Like this

There are 2 ways that we can persist this relationship. These are shown below

#### 139.3.1 Using Join Table

If you define the XML metadata for these classes as follows
<entity-mappings>
  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      <id name="id">
        <column name="ACCOUNT_ID"/>
      </id>
      ...
      <one-to-many name="addresses" target-entity="com.mydomain.Address" mapped-by="account">
        <join-table name="ACCOUNT_ADDRESSES">
          <join-column name="ACCOUNT_ID_OID"/>
          <inverse-join-column name="ADDRESS_ID_EID"/>
        </join-table>
      </one-to-many>
    </attributes>
  </entity>
  <entity class="Address">
    <table name="ADDRESS"/>
    <attributes>
      <id name="id">
        <column name="ADDRESS_ID"/>
      </id>
      ...
      <many-to-one name="account"/>
    </attributes>
  </entity>
</entity-mappings>

or alternatively using annotations

public class Account
{
  ...

  @OneToMany(mappedBy="account")
  @JoinTable(name="ACCOUNT_ADDRESSES")
  @JoinColumn(name="ACCOUNT_ID_OID")
  @InverseJoinColumn(name="ADDRESS_ID_EID")
  Collection<Address> addresses
}

public class Address
{
  ...

  @ManyToOne
  Account account;

  ...
}
The crucial part is the join element on the field element - this signals to JPA to use a join table.

This will create 3 tables in the database, one for Address, one for Account, and a join table, as shown below.

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips:

- To specify the name of the table where a class is stored, specify the table element below the class element.
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the basic element.
- To specify the name of the join table, specify the join-table element below the one-to-many element with the collection.
- To specify the names of the join table columns, use the join-column and inverse-join-column elements below the join-table element.
- The join table will NOT be given a primary key (since a Collection can have duplicates).
- When forming the relation please make sure that you set the relation at BOTH sides since DataNucleus would have no way of knowing which end is correct if you only set one end.

139.3.2 Using Foreign-Key

Here we have the 2 classes with both knowing about the relationship with the other.

If you define the XML metadata for these classes as follows
or alternatively using annotations

```java
public class Account {
    ...
    @OneToMany(mappedBy="account")
    @JoinColumn(name="ACCOUNT_ID")
    Collection<Address> addresses
}

public class Address {
    ...
    @ManyToOne
    Account account;
    ...
}
```

The crucial part is the *mapped-by* attribute of the field on the "1" side of the relationship. This tells the JPA implementation to look for a field called *account* on the Address class.
This will create 2 tables in the database, one for Address (including an ACCOUNT_ID to link to the ACCOUNT table), and one for Account. Notice the subtle difference to this set-up to that of the Join Table relationship earlier.

If you wish to fully define the schema table and column names etc, follow these tips:

- To specify the name of the table where a class is stored, specify the `table` element below the `class` element.
- To specify the names of the columns where the fields of a class are stored, specify the `column` attribute on the `basic` element.
- When forming the relation please make sure that you set the relation at BOTH sides since DataNucleus would have no way of knowing which end is correct if you only set one end.

**Limitation:** Since each Address object can have at most one owner (due to the "Foreign Key") this mode of persistence will not allow duplicate values in the Collection. If you want to allow duplicate Collection entries, then use the "Join Table" variant above.

### 139.4 1-N Collection of non-Entity objects

In JPA1 you cannot have a 1-N collection of non-Entity objects. All of the examples above show a 1-N relationship between 2 persistable classes. If you want the element to be primitive or Object types then follow this section. For example, when you have a Collection of Strings. This will be persisted in the same way as the "Join Table" examples above. A join table is created to hold the collection elements. Let's take our example. We have an Account that stores a Collection of addresses. These addresses are simply Strings. We define the annotations like this:

```java
@Entity
public class Account
{
    ...

    @ElementCollection
    @CollectionTable(name="ACCOUNT_ADDRESSES")
    Collection<String> addresses;
}
```

or using XML metadata.
<entity class="mydomain.Account">
  <attributes>
    ...
    <element-collection name="addresses">
      <collection-table name="ACCOUNT_ADDRESSES"/>
    </element-collection>
  </attributes>
</entity>

In the datastore the following is created

The ACCOUNT table is as before, but this time we only have the "join table". Use @Column on the field/method to define the column details of the element in the join table.
140 Sets

140.1 JPA : 1-N Relationships with Sets

You have a 1-N (one to many) when you have one object of a class that has a Set of objects of another class. Please note that Sets do not allow duplicates, and so the persistence process reflects this with the choice of primary keys. There are two ways in which you can represent this in a datastore: Join Table (where a join table is used to provide the relationship mapping between the objects), and Foreign-Key (where a foreign key is placed in the table of the object contained in the Set).

The various possible relationships are described below.

- 1-N Unidirectional using Join Table
- 1-N Unidirectional using Foreign-Key
- 1-N Bidirectional using Join Table
- 1-N Bidirectional using Foreign-Key
- 1-N Unidirectional of non-PC using Join Table

This page is aimed at Set fields and so applies to fields of Java type java.util.HashSet, java.util.LinkedHashSet, java.util.Set, java.util.SortedSet, java.util.TreeSet

Please note that RDBMS supports the full range of options on this page, whereas other datastores (ODF, Excel, HBase, MongoDB, etc) persist the Set in a column in the owner object rather than using join-tables or foreign-keys since those concepts are RDBMS-only.

140.1.1 equals() and hashCode()

Important: The element of a Collection ought to define the methods equals and hashCode so that updates are detected correctly. This is because any Java Collection will use these to determine equality and whether an element is contained in the Collection. Note also that the hashCode() should be consistent throughout the lifetime of a persistable object. By that we mean that it should not use some basis before persistence and then use some other basis (such as the object identity) after persistence, for this reason we do not recommend usage of JDOHelper.getObjectId(obj) in the equals/hashCode methods.

140.2 1-N Set Unidirectional

We have 2 sample classes Account and Address. These are related in such a way as Account contains a Set of objects of type Address, yet each Address knows nothing about the Account objects that it relates to. Like this:

There are 2 ways that we can persist this relationship. These are shown below.
140.2.1 Using Join Table

If you define the XML metadata for these classes as follows

```xml
<entity-mappings>
  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      <id name="id">
        <column name="ACCOUNT_ID"/>
      </id>
      ...
      <one-to-many name="addresses" target-entity="com.mydomain.Address">
        <join-table name="ACCOUNT_ADDRESSES">
          <join-column name="ACCOUNT_ID_OID"/>
          <inverse-join-column name="ADDRESS_ID_EID"/>
        </join-table>
      </one-to-many>
    </attributes>
  </entity>
  <entity class="Address">
    <table name="ADDRESS"/>
    <attributes>
      <id name="id">
        <column name="ADDRESS_ID"/>
      </id>
      ...
    </attributes>
  </entity>
</entity-mappings>
```

or alternatively using annotations

```java
public class Account {
  ...

  @OneToMany
  @JoinTable(name="ACCOUNT_ADDRESSES")
  @JoinColumn(name="ACCOUNT_ID_OID")
  @InverseJoinColumn(name="ADDRESS_ID_EID")
  Set<Address> addresses
}

gc
public class Address {
  ...
}
```

The crucial part is the `join-table` element on the field element - this signals to JPA to use a join table.
This will create 3 tables in the database, one for Address, one for Account, and a join table, as shown below.

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the `table` element below the `class` element
- To specify the names of the columns where the fields of a class are stored, specify the `column` attribute on the `basic` element.
- To specify the name of the join table, specify the `join-table` element below the `one-to-many` element with the Set
- To specify the names of the join table columns, use the `join-column` and `inverse-join-column` elements below the `join-table` element.
- The join table will be given a primary key (since a Set can't have duplicates).

### 140.2.2 Using Foreign-Key

In this relationship, the Account class has a List of Address objects, yet the Address knows nothing about the Account. In this case we don't have a field in the Address to link back to the Account and so DataNucleus has to use columns in the datastore representation of the Address class. So we define the XML metadata like this
or alternatively using annotations

```java
public class Account
{
    ...

    @OneToMany
    @JoinColumn(name="ACCOUNT_ID")
    Set<Address> addresses
}

public class Address
{
    ...
}
```

**Note that you MUST specify the join-column here otherwise it defaults to a join table with JPA!**

There will be 2 tables, one for `Address`, and one for `Account`. If you wish to specify the names of the column(s) used in the schema for the foreign key in the Address table you should use the `join-column` element within the field of the Set.
In terms of operation within your classes of assigning the objects in the relationship. You have to take your Account object and add the Address to the Account Set field since the Address knows nothing about the Account.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the `table` element below the `class` element
- To specify the names of the columns where the fields of a class are stored, specify the `column` attribute on the `basic` element.

**Limitation:** Since each Address object can have at most one owner (due to the "Foreign Key") this mode of persistence will not allow duplicate values in the Set. If you want to allow duplicate Set entries, then use the "Join Table" variant above.

### 140.3 1-N Set Bidirectional

We have 2 sample classes `Account` and `Address`. These are related in such a way as `Account` contains a `Set` of objects of type `Address`, and each `Address` has a reference to the `Account` object that it relates to. Like this

There are 2 ways that we can persist this relationship. These are shown below

### 140.3.1 Using Join Table

If you define the XML metadata for these classes as follows
or alternatively using annotations
public class Account
{
    ...

    @OneToMany(mappedBy="account")
    @JoinTable(name="ACCOUNT_ADDRESSES")
    @JoinColumn(name="ACCOUNT_ID_OID")
    @InverseJoinColumn(name="ADDRESS_ID_EID")
    Set<Address> addresses
}

public class Address
{
    ...

    @ManyToOne
    Account account;
    ...
}

The crucial part is the join element on the field element - this signals to JPA to use a join table.

This will create 3 tables in the database, one for Address, one for Account, and a join table, as shown below.

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table element below the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the basic element.
- To specify the name of the join table, specify the join-table element below the one-to-many element with the set.
- To specify the names of the join table columns, use the join-column and inverse-join-column elements below the join-table element.
- The join table will be given a primary key (since a Set can't have duplicates).
- When forming the relation please make sure that you set the relation at BOTH sides since DataNucleus would have no way of knowing which end is correct if you only set one end.
140.3.2 Using Foreign-Key

Here we have the 2 classes with both knowing about the relationship with the other.

If you define the XML metadata for these classes as follows

```xml
<entity-mappings>
  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      <id name="id">
        <column name="ACCOUNT_ID"/>
      </id>
      ...  
      <one-to-many name="addresses" target-entity="com.mydomain.Address" mapped-by="account">
        <join-column name="ACCOUNT_ID"/>
      </one-to-many>
    </attributes>
  </entity>
  <entity class="Address">
    <table name="ADDRESS"/>
    <attributes>
      <id name="id">
        <column name="ADDRESS_ID"/>
      </id>
      ...  
      <many-to-one name="account">
        </many-to-one>
    </attributes>
  </entity>
</entity-mappings>
```

or alternatively using annotations

```java
public class Account
{
  ...
  @OneToMany(mappedBy="account")
  @JoinColumn(name="ACCOUNT_ID")
  Set<Address> addresses
}

public class Address
{
  ...
  @ManyToOne
  Account account;
  ...
}
```
The crucial part is the mapped-by attribute of the field on the "1" side of the relationship. This tells the JPA implementation to look for a field called account on the Address class.

This will create 2 tables in the database, one for Address (including an ACCOUNT_ID to link to the ACCOUNT table), and one for Account. Notice the subtle difference to this set-up to that of the Join Table relationship earlier.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table element below the class element
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the basic element.
- When forming the relation please make sure that you set the relation at BOTH sides since DataNucleus would have no way of knowing which end is correct if you only set one end.

Limitation: Since each Address object can have at most one owner (due to the "Foreign Key") this mode of persistence will not allow duplicate values in the Set. If you want to allow duplicate Set entries, then use the "Join Table" variant above.

140.4 1-N Collection of non-Entity objects

In JPA1 you cannot have a 1-N set of non-Entity objects. This is available in JPA2. All of the examples above show a 1-N relationship between 2 PersistenceCapable classes. If you want the element to be primitive or Object types then follow this section. For example, when you have a Set of Strings. This will be persisted in the same way as the "Join Table" examples above. A join table is created to hold the collection elements. Let's take our example. We have an Account that stores a Collection of addresses. These addresses are simply Strings. We define the annotations like this

```java
@Entity
public class Account{
    
    @ElementCollection
    @CollectionTable(name="ACCOUNT_ADDRESSES")
    Collection<String> addresses;
}
```

or using XML metadata
In the datastore the following is created

The ACCOUNT table is as before, but this time we only have the "join table". Use @Column on the field/method to define the column details of the element in the join table.
141 Lists

141.1 JPA : 1-N Relationships with Lists
You have a 1-N (one to many) when you have one object of a class that has a List of objects of another class. There are two ways in which you can represent this in a datastore. Join Table (where a join table is used to provide the relationship mapping between the objects), and Foreign-Key (where a foreign key is placed in the table of the object contained in the List.

The various possible relationships are described below.
- 1-N Unidirectional using Join Table
- 1-N Unidirectional using Foreign-Key
- 1-N Bidirectional using Join Table
- 1-N Bidirectional using Foreign-Key
- 1-N Unidirectional of non-PC using Join Table

This page is aimed at List fields and so applies to fields of Java type java.util.ArrayList, java.util.LinkedList, java.util.List, java.util.Stack, java.util.Vector

Please note that RDBMS supports the full range of options on this page, whereas other datastores (ODF, Excel, HBase, MongoDB, etc) persist the List in a column in the owner object rather than using join-tables or foreign-keys since those concepts are RDBMS-only

In JPA1 all List relationships are "ordered Lists". If a List is an "ordered List" then the positions of the elements in the List at persistence are not preserved (are not persisted) and instead an ordering is defined for their retrieval. In JPA2 Lists can optionally use a surrogate column to handle the ordering. This means that the positions of the elements in List at persistence are preserved. This is the same situation as JDO provides.

141.1.1 equals() and hashCode()
Important : The element of a Collection ought to define the methods equals and hashCode so that updates are detected correctly. This is because any Java Collection will use these to determine equality and whether an element is contained in the Collection. Note also that the hashCode() should be consistent throughout the lifetime of a persistable object. By that we mean that it should not use some basis before persistence and then use some other basis (such as the object identity) after persistence, for this reason we do not recommend usage of JDOHelper.getObjectId(obj) in the equals/hashCode methods.

141.2 1-N List Unidirectional
We have 2 sample classes Account and Address. These are related in such a way as Account contains a List of objects of type Address, yet each Address knows nothing about the Account objects that it relates to. Like this

There are 2 ways that we can persist this relationship. These are shown below.
141.2.1 Using Join Table

If you define the XML metadata for these classes as follows

```xml
<entity-mappings>
  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      <id name="id">
        <column name="ACCOUNT_ID"/>
      </id>

      <one-to-many name="addresses" target-entity="com.mydomain.Address">
        <order-by>id</order-by>
        <join-table name="ACCOUNT_ADDRESSES">
          <join-column name="ACCOUNT_ID_OID"/>
          <inverse-join-column name="ADDRESS_ID_EID"/>
        </join-table>
      </one-to-many>
    </attributes>
  </entity>

  <entity class="Address">
    <table name="ADDRESS"/>
    <attributes>
      <id name="id">
        <column name="ADDRESS_ID"/>
      </id>

      ...
    </attributes>
  </entity>
</entity-mappings>
```

or alternatively using annotations
public class Account
{
    ...

    @OneToMany
    @OrderBy("id")
    @JoinTable(name="ACCOUNT_ADDRESSES")
    @JoinColumn(name="ACCOUNT_ID_OID")
    @InverseJoinColumn(name="ADDRESS_ID_EID")
    List<Address> addresses
}

public class Address
{
    ...
}

The crucial part is the \texttt{join-table} element on the field element - this signals to JPA to use a join table.

There will be 3 tables, one for Address, one for Account, and the join table. This is identical to the handling for Sets/Collections. Note that we specified \texttt{<order-by>} which defines the order the elements are retrieved in (the "id" is the field in the List element).

![ACCOUNT_ADDRESS_ADDRESSES_ADDRESS](image)

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips
- To specify the name of the table where a class is stored, specify the \texttt{table} element below the \texttt{class} element.
- To specify the names of the columns where the fields of a class are stored, specify the \texttt{column} attribute on the \texttt{basic} element.
- To specify the name of the join table, specify the \texttt{join-table} element below the \texttt{one-to-many} element with the collection.
- To specify the names of the join table columns, use the \texttt{join-column} and \texttt{inverse-join-column} elements below the \texttt{join-table} element.
- The join table will NOT be given a primary key (so that duplicates can be stored).
- If you want to have a surrogate column added to contain the ordering you should specify \texttt{order-column} (@OrderColumn) instead of \texttt{order-by}. This is available from JPA2

141.2.2 Using Foreign-Key

In this relationship, the Account class has a List of Address objects, yet the Address knows nothing about the Account. In this case we don't have a field in the Address to link back to the Account and so
DataNucleus has to use columns in the datastore representation of the Address class. So we define the XML metadata like this

```xml
<entity-mappings>
  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      <id name="id">
        <column name="ACCOUNT_ID"/>
      </id>
      ...
      <one-to-many name="addresses" target-entity="com.mydomain.Address">
        <order-by>city</order-by>
        <join-column name="ACCOUNT_ID"/>
      </one-to-many>
    </attributes>
  </entity>

  <entity class="Address">
    <table name="ADDRESS"/>
    <attributes>
      <id name="id">
        <column name="ADDRESS_ID"/>
      </id>
      ...
    </attributes>
  </entity>
</entity-mappings>
```

or alternatively using annotations

```java
public class Account {
  ...

  @OneToMany
  @OrderBy("city")
  @JoinColumn(name="ACCOUNT_ID")
  List<Address> addresses
}

public class Address {
  ...
}
```

> **Note that you MUST specify the join-column here otherwise it defaults to a join table with JPA!**

Again there will be 2 tables, one for Address, and one for Account. Note that we have no "mapped-by" attribute specified, and also no "join" element. If you wish to specify the names of the columns used in the schema for the foreign key in the Address table you should use the element element within the field of the collection.
In terms of operation within your classes of assigning the objects in the relationship. With DataNucleus and List-based containers you have to take your Account object and add the Address to the Account collection field since the Address knows nothing about the Account.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the `table` element below the `class` element
- To specify the names of the columns where the fields of a class are stored, specify the `column` attribute on the `basic` element.

**Limitations**

- If we are using an "ordered List" and the primary key of the join table contains owner and element then duplicate elements can't be stored in a List. If you want to allow duplicate List entries, then use the "Join Table" variant above.

### 141.3 1-N List Bidirectional

We have 2 sample classes `Account` and `Address`. These are related in such a way as `Account` contains a `List` of objects of type `Address`, and each `Address` has a reference to the `Account` object that it relates to. Like this

There are 2 ways that we can persist this relationship. These are shown below

#### 141.3.1 Using Join Table

If you define the XML metadata for these classes as follows
or alternatively using annotations
The crucial part is the `join` element on the field element - this signals to JPA to use a join table.

This will create 3 tables in the database, one for Address, one for Account, and a join table, as shown below.

The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the `table` element below the `class` element
- To specify the names of the columns where the fields of a class are stored, specify the `column` attribute on the `basic` element.
- To specify the name of the join table, specify the `join-table` element below the `one-to-many` element with the collection.
- To specify the names of the join table columns, use the `join-column` and `inverse-join-column` elements below the `join-table` element.
- The join table will NOT be given a primary key (so that duplicates can be stored).
- When forming the relation please make sure that you set the relation at BOTH sides since DataNucleus would have no way of knowing which end is correct if you only set one end.
- If you want to have a surrogate column added to contain the ordering you should specify `order-column` (@OrderColumn) instead of `order-by`. This is available from JPA2.
141.3.2 Using Foreign-Key

Here we have the 2 classes with both knowing about the relationship with the other.

Please note that an Foreign-Key List will NOT, by default, allow duplicates. This is because it stores the element position in the element table. If you need a List with duplicates we recommend that you use the Join Table List implementation above. If you have an application identity element class then you could in principle add the element position to the primary key to allow duplicates, but this would imply changing your element class identity.

If you define the XML metadata for these classes as follows

```
<entity-mappings>
  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      <id name="id">
        <column name="ACCOUNT_ID"/>
      </id>
      ...
      <one-to-many name="addresses" target-entity="com.mydomain.Address" mapped-by="account">
        <order-by>city ASC</order-by>
        <join-column name="ACCOUNT_ID"/>
      </one-to-many>
    </attributes>
  </entity>
  <entity class="Address">
    <table name="ADDRESS"/>
    <attributes>
      <id name="id">
        <column name="ADDRESS_ID"/>
      </id>
      ...
      <many-to-one name="account"/>
    </attributes>
  </entity>
</entity-mappings>
```

or alternatively using annotations
public class Account
{
    ...

    @OneToMany(mappedBy="account")
    @OrderBy("city")
    @JoinColumn(name="ACCOUNT_ID")
    List<Address> addresses
}

public class Address
{
    ...

    @ManyToOne
    Account account;
    ...
}

The crucial part is the mapped-by attribute of the field on the "1" side of the relationship. This tells the JPA implementation to look for a field called account on the Address class.

This will create 2 tables in the database, one for Address (including an ACCOUNT_ID to link to the ACCOUNT table), and one for Account. Notice the subtle difference to this set-up to that of the Join Table relationship earlier.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the table element below the class element.
- To specify the names of the columns where the fields of a class are stored, specify the column attribute on the basic element.
- When forming the relation please make sure that you set the relation at BOTH sides since DataNucleus would have no way of knowing which end is correct if you only set one end.

Limitation: Since each Address object can have at most one owner (due to the "Foreign Key") this mode of persistence will not allow duplicate values in the Collection. If you want to allow duplicate Collection entries, then use the "Join Table" variant above.

141.4 1-N Collection of non-Entity objects
In JPA1 you cannot have a 1-N List of non-Entity objects. This is available in JPA2. All of the examples above show a 1-N relationship between 2 persistable classes. If you want the element to be primitive or Object types then follow this section. For example, when you have a List of Strings. This will be persisted in the same way as the "Join Table" examples above. A join table is created to hold the collection elements. Let’s take our example. We have an Account that stores a Collection of addresses. These addresses are simply Strings. We define the annotations like this

```java
@Entity
public class Account
{
  ...
  @ElementCollection
  @CollectionTable(name="ACCOUNT_ADDRESSES")
  Collection<String> addresses;
}
```

or using XML metadata

```xml
<entity class="mydomain.Account">
  <attributes>
    ...
    <element-collection name="addresses">
      <collection-table name="ACCOUNT_ADDRESSES"/>
    </element-collection>
  </attributes>
</entity>
```

In the datastore the following is created

The ACCOUNT table is as before, but this time we only have the "join table". Use @Column on the field/method to define the column details of the element in the join table.
142 Maps

142.1 JPA : 1-N Relationships with Maps

You have a 1-N (one to many) when you have one object of a class that has a Map of objects of another class. There are two general ways in which you can represent this in a datastore: **Join Table** (where a join table is used to provide the relationship mapping between the objects), and **Foreign-Key** (where a foreign key is placed in the table of the object contained in the Map).

The various possible relationships are described below.

- Map[Simple, PC] using join table
- Map[Simple, Simple] using join table
- 1-N Unidirectional using Foreign-Key (key stored in the value class)
- 1-N Bidirectional using Foreign-Key (key stored in the value class)

This page is aimed at Map fields and so applies to fields of Java type `java.util.HashMap`, `java.util.Hashtable`, `java.util.LinkedHashMap`, `java.util.Map`, `java.util.SortedMap`, `java.util.TreeMap`, `java.util.Properties`

Please note that RDBMS supports the full range of options on this page, whereas other datastores (ODF, Excel, HBase, MongoDB, etc) persist the Map in a column in the owner object rather than using join-tables or foreign-keys since those concepts are RDBMS-only.

142.2 1-N Map using Join Table

We have a class **Account** that contains a Map. With a Map we store values using keys. As a result we have the following combinations of key and value, bearing in mind whether the key or value is persistable.

142.2.1 Map[Simple, PC]

Here our key is a simple type (in this case a String) and the values are **PersistenceCapable**. Like this

```
Account
- firstName: String
- lastName: String
- addresses: Map<String, Address>

Address
- city: String
- street: String
```

If you define the Meta-Data for these classes as follows
This will create 3 tables in the datastore, one for `Account`, one for `Address` and a join table also containing the key.

You can configure the names of the columns in the join table using the `joinColumns` attributes of the various annotations.

Please note that the column ADPT_PK_IDX is added by DataNucleus when the column type of the key is not valid to be part of a primary key (with the RDBMS being used). If the column type of your key is acceptable for use as part of a primary key then you will not have this "ADPT_PK_IDX" column.

### 142.2.2 Map[Simple, Simple]

Here our keys and values are of simple types (in this case a String). Like this

If you define the Meta-Data for these classes as follows
@Entity
public class Account
{
    @ElementCollection
    @CollectionTable
    Map<String, String> addresses;
    ...
}

This results in just 2 tables. The "join" table contains both the key AND the value.

![Diagram of ACCOUNT and ACCOUNT_ADDRESSES tables]

You can configure the names of the columns in the join table using the `joinColumns` attributes of the various annotations.

Please note that the column ADPT_PK_IDX is added by DataNucleus when the column type of the key is not valid to be part of a primary key (with the RDBMS being used). If the column type of your key is acceptable for use as part of a primary key then you will not have this "ADPT_PK_IDX" column.

### 142.3 1-N Map using Foreign-Key

#### 142.3.1 1-N Foreign-Key Unidirectional (key stored in value)

In this case we have an object with a Map of objects and we're associating the objects using a foreign-key in the table of the value. We're using a field (alias) in the Address class as the key of the map.

![Diagram of Account and Address classes with a 1-N relationship]

In this relationship, the Account class has a Map of Address objects, yet the Address knows nothing about the Account. In this case we don't have a field in the Address to link back to the Account and so DataNucleus has to use columns in the datastore representation of the Address class. So we define the MetaData like this
Again there will be 2 tables, one for Address, and one for Account. If you wish to specify the names of the columns used in the schema for the foreign key in the Address table you should use the join-column element within the field of the map.

In terms of operation within your classes of assigning the objects in the relationship. You have to take your Account object and add the Address to the Account map field since the Address knows nothing about the Account. Also be aware that each Address object can have only one owner, since it has a single foreign key to the Account.
### 142.3.2 1-N Foreign-Key Bidirectional (key stored in value)

In this case we have an object with a Map of objects and we're associating the objects using a foreign-key in the table of the value.

With these classes we want to store a foreign-key in the value table (ADDRESS), and we want to use the "alias" field in the Address class as the key to the map. If you define the Meta-Data for these classes as follows:

```xml
<entity-mappings>
  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      <id name="id">
        <column name="ACCOUNT_ID"/>
      </id>
      ...
      <one-to-many name="addresses" target-entity="com.mydomain.Address" mapped-by="account">
        <map-key name="alias"/>
      </one-to-many>
    </attributes>
  </entity>
  <entity class="Address">
    <table name="ADDRESS"/>
    <attributes>
      <id name="id">
        <column name="ADDRESS_ID"/>
      </id>
      ...
      <basic name="alias">
        <column name="KEY" length="20"/>
      </basic>
      <many-to-one name="account">
        <join-column name="ACCOUNT_ID_OID"/>
      </many-to-one>
    </attributes>
  </entity>
</entity-mappings>
```

This will create 2 tables in the datastore. One for Account, and one for Address. The table for Address will contain the key field as well as an index to the Account record (notated by the `mapped-by` tag).
143 N-to-1 Relations

143.1 JPA : N-1 Relationships

You have a N-to-1 relationship when an object of a class has an associated object of another class (only one associated object) and several of this type of object can be linked to the same associated object. From the other end of the relationship it is effectively a 1-N, but from the point of view of the object in question, it is N-1. You can create the relationship in 2 ways depending on whether the 2 classes know about each other (bidirectional), or whether only the "N" side knows about the other class (unidirectional). These are described below.

143.1.1 Unidirectional

For this case you could have 2 classes, User and Account, as below.

![Diagram showing User and Account classes with a 1:1 relationship]

so the Account class ("N" side) knows about the User class ("1" side), but not vice-versa, and are using a join table. A particular user could be related to several accounts. If you define the Meta-Data for these classes as follows
<entity-mappings>
  <entity class="User">
    <table name="USER"/>
    <attributes>
      <id name="id">
        <column name="USER_ID"/>
      </id>
      ...
    </attributes>
  </entity>

  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      <id name="id">
        <column name="ACCOUNT_ID"/>
      </id>
      ...
      <many-to-one name="user">
        <join-table name="ACCOUNT_USER"/>
      </many-to-one>
    </attributes>
  </entity>
</entity-mappings>

alternatively using annotations

```java
public class Account {
  ...

  @ManyToOne
  @JoinTable(name="ACCOUNT_USER")
  User user;
}
```

This will create 3 tables in the database, one for User (with name USER), one for Account (with name ACCOUNT), and a join table (with name ACCOUNT_USER), as shown below.

Note that in the case of non-RDBMS datastores there is no join-table, simply a "column" in the ACCOUNT "table", storing the "id" of the related object.
143.1.2 Bidirectional

This relationship is described in the guide for 1-N relationships. In particular there are 2 ways to define the relationship for RDBMS: the first uses a Join Table to hold the relationship, whilst the second uses a Foreign Key in the "N" object to hold the relationship. For non-RDBMS datastores each side will have a "column" (or equivalent) in the "table" of the N side storing the "id" of the related (owning) object. Please refer to the 1-N relationships bidirectional relations since they show this exact relationship.
144 M-to-N Relations

144.1 JPA : M-N Relationships

You have a M-to-N (or Many-to-Many) relationship if an object of a class A has associated objects of class B, and class B has associated objects of class A. This relationship may be achieved through Java Collection, Set, List or subclasses of these, although the only one that supports a true M-N is Set.

With DataNucleus this can be set up as described in this section, using what is called a Join Table relationship. Let's take the following example and describe how to model it with the different types of collection classes. We have 2 classes, Product and Supplier as below.

Here the Product class knows about the Supplier class. In addition the Supplier knows about the Product class, however with these relationships are really independent.

The various possible relationships are described below.

- M-N Set relation
- M-N Ordered List relation

144.2 Using Set

If you define the Meta-Data for these classes as follows
or alternatively using annotations

```java
public class Product
{
    ...

    @ManyToMany(mappedBy="products")
    @JoinTable(name="PRODUCTS_SUPPLIERS",
        joinColumns={@JoinColumn(name="PRODUCT_ID")},
        inverseJoinColumns={@InverseJoinColumn(name="SUPPLIER_ID")})
    Collection<Supplier> suppliers
}

public class Supplier
{
    ...

    @ManyToMany
    Collection<Product> products;
    ...
}
Note how we have specified the information only once regarding join table name, and join column names as well as the `<join-table>`. This is the JPA standard way of specification, and results in a single join table. The "mapped-by" ties the two fields together.

See also :-

- M-N Worked Example
- M-N with Attributes Worked Example

### 144.3 Using Ordered Lists

If you define the Meta-Data for these classes as follows

```xml
<entity-mappings>
<entity class="mydomain.Product">
  <table name="PRODUCT"/>
  <attributes>
    <id name="id">
      <column name="PRODUCT_ID"/>
    </id>
    ...
    <many-to-many name="suppliers" mapped-by="products">
      <order-by>name</order-by>
      <join-table name="PRODUCTS_SUPPLIERS">
        <join-column name="PRODUCT_ID"/>
        <inverse-join-column name="SUPPLIER_ID"/>
      </join-table>
    </many-to-many>
  </attributes>
</entity>

<entity class="mydomain.Supplier">
  <table name="SUPPLIER"/>
  <attributes>
    <id name="id">
      <column name="SUPPLIER_ID"/>
    </id>
    ...
    <many-to-many name="products">
      <order-by>name</order-by>
    </many-to-many>
  </attributes>
</entity>
</entity-mappings>
```

or using annotations
There will be 3 tables, one for Product, one for Supplier, and the join table. The difference from the Set example is that we now have <order-by> at both sides of the relation. This has no effect in the datastore schema but when the Lists are retrieved they are ordered using the specified order-by.

Note that you cannot have a many-to-many relation using indexed lists since both sides would need its own index.

144.4 Relationship Behaviour

Please be aware of the following.

- To add an object to an M-N relationship you need to set it at both ends of the relation since the relation is bidirectional and without such information the JPA implementation won’t know which end of the relation is correct.
- If you want to delete an object from one end of a M-N relationship you will have to remove it first from the other objects relationship. If you don’t you will get an error message that the object to be deleted has links to other objects and so cannot be deleted.
145 Managing Relationships

145.1 JPA : Managing Relationships

The power of a Java persistence solution like DataNucleus is demonstrated when persisting relationships between objects. There are many types of relationships.

- **1-1 relationships** - this is where you have an object A relates to a second object B. The relation can be *unidirectional* where A knows about B, but B doesn't know about A. The relation can be *bidirectional* where A knows about B and B knows about A.

- **1-N relationships** - this is where you have an object A that has a collection of other objects of type B. The relation can be *unidirectional* where A knows about the objects B but the Bs don't know about A. The relation can be *bidirectional* where A knows about the objects B and the Bs know about A.

- **N-1 relationships** - this is where you have an object B1 that relates to an object A, and an object B2 that relates to A also etc. The relation can be *unidirectional* where the A doesn't know about the Bs. The relation can be *bidirectional* where the A has a collection of the Bs. [i.e a 1-N relationship but from the point of view of the element]

- **M-N relationships** - this is where you have objects of type A that have a collection of objects of type B and the objects of type B also have a collection of objects of type A. The relation is always *bidirectional* by definition.

145.1.1 Assigning Relationships

When the relation is *unidirectional* you simply set the related field to refer to the other object. For example we have classes A and B and the class A has a field of type B. So we set it like this:

```java
A a = new A();
B b = new B();
a.setB(b); // "a" knows about "b"
```

When the relation is *bidirectional* you **have to set both sides** of the relation. For example, we have classes A and B and the class A has a collection of elements of type B, and B has a field of type A. So we set it like this:

```java
A a = new A();
B b1 = new B();
a.addElement(b1); // "a" knows about "b1"
b1.setA(a); // "b1" knows about "a"
```

So it is really simple, with only 1 real rule. **With a bidirectional relation you must set both sides of the relation.**

145.1.2 Persisting Relationships - Reachability

To persist an object with JPA you call the `EntityManager` method `persist`. The object passed in will be persisted. By default all related objects will not be persisted with that object. You can however change this by specifying the cascade PERSIST property for that field. With this the related object(s) would also be persisted (or updated with any new values if they are already persistent). This process is called *persistence-by-reachability*. With JDO the default is to persist all reachables, whereas with
JPA you have to explicitly set this behaviour. For example we have classes A and B and class A has a field of type B and this field has the cascade property PERSIST set. To persist them we could do

```java
A a = new A();
B b = new B();
a.setB(b); // "a" knows about "b"
em.persist(a);
```

### 145.1.3 Managed Relationships

As we have mentioned above, it is for the user to set both sides of a bidirectional relation. If they dont and object A knows about B, but B doesnt know about A then what is the persistence solution to do? It doesnt know which side of the relation is correct. JPA doesnt define the behaviour currently for this situation. DataNucleus has two ways of handling this situation. If you have the persistence property "datanucleus.manageRelationships" set to true then it will make sure that the other side of the relation is set correctly, correcting obvious omissions, and giving exceptions for obvious errors. If you set that persistence property to false then it will assume that your objects have their bidirectional relationships consistent and will just persist what it finds.

When performing management of relations there are some checks implemented to spot typical errors in user operations e.g add an element to a collection and then remove it (why?!). You can disable these checks using `datanucleus.manageRelationshipsChecks`, set to false.
146 Cascading

146.1 JPA : Cascading Fields

When defining your objects to be persisted and the relationships between them, it is often required to define dependencies between these related objects. When persisting an object should we also persist any related objects? What should happen to a related object when an object is deleted? Can the related object exist in its own right beyond the lifecycle of the other object, or should it be deleted along with the other object? This behaviour can be defined with JPA and with DataNucleus. Let's take an example.

```java
@Entity
public class Owner
{
    @OneToOne
    private DrivingLicense license;

    @OneToMany(mappedBy="owner")
    private Collection cars;

    ...
}

@Entity
public class DrivingLicense
{
    private String serialNumber;

    ...
}

@Entity
public class Car
{
    private String registrationNumber;

    @ManyToOne
    private Owner owner;

    ...
}
```

So we have an Owner of a collection of vintage Car's, and the Owner has a DrivingLicense. We want to define lifecycle dependencies to match the relationships that we have between these objects. So in our example what we are going to do is:

- When an object is persisted/updated its related objects are also persisted/updated.
- When an Owner object is deleted, its DrivingLicense is deleted too (since it can't exist without the person!
- When an Owner object is deleted, the Cars continue to exist (since someone will buy them)
- When a Car object is deleted, the Owner continues to exist (unless he/she dies in the Car, but that will be handled by a different mechanism in our application).
So we update our class to reflect this

```java
@Entity
public class Owner
{
    @ManyToOne(cascade=CascadeType.ALL)
    private DrivingLicense license;

    @OneToMany(mappedBy="owner", cascade={CascadeType.PERSIST, CascadeType.MERGE})
    private Collection cars;

    ...
}

@Entity
public class DrivingLicense
{
    private String serialNumber;

    ...
}

@Entity
public class Car
{
    private String registrationNumber;

    @ManyToOne(cascade={CascadeType.PERSIST, CascadeType.MERGE})
    private Owner owner;

    ...
}
```

So we make use of the `cascade` attribute of the relation annotations. We could express this similarly in XML.
146.1.1 Orphans

When an element is removed from a collection, or when a 1-1 relation is nulled, sometimes it is desirable to delete the other object. JPA2 defines a facility of removing "orphans" by specifying metadata for a 1-1 or 1-N relation. Let's take an example. In the above relation between Owner and DrivingLicense if we set the owner's license field to null, this should mean that the license is deleted. So we could change it to be
@Entity
class Owner
{
    @OneToOne(cascade={CascadeType.PERSIST, CascadeType.MERGE}, orphanRemoval=true)
    private DrivingLicense license;

    ...
}

@Entity
class DrivingLicense
{
    private String serialNumber;

    ...
}

So from now on, if we delete the Owner we delete the DrivingLicense, and if we set the license field of DrivingLicense to null then we also delete the DrivingLicense.
147 MetaData Reference

147.1 JPA : Metadata Overview

JPA requires the persistence of classes to be defined via Metadata. This Metadata can be provided in the following forms

- **XML**: the traditional mechanism, with XML files containing information for each class to be persisted.
- **Annotations**: using JDK1.5+ annotations in the classes to be persisted

We recommend that you use either XML or annotations for the basic persistence information, but always use XML for ORM information. This is because it is liable to change at deployment time and hence is accessible when in XML form whereas in annotations you add an extra compile cycle (and also you may need to deploy to some other datastore at some point, hence needing a different deployment).

147.1.1 Metadata priority

JPA defines the priority order for metadata as being

- JPA XML Metadata
- Annotations

If a class has annotations and JPA XML Metadata then the XML Metadata will take precedence over the annotations (or rather be merged on top of the annotations).

147.1.2 XML Metadata validation

By default any XML Metadata will be validated for accuracy when loading it. Obviously XML is defined by a DTD or XSD schema and so should follow that. You can turn off such validations by setting the persistence property `datanucleus.metadata.validate` to false when creating your PMF. Note that this only turns off the XML strictness validation, and not the checks on inconsistency of specification of relations etc.
148.1 JPA : XML Metadata Reference

JPA XML MetaData allows you to define mapping information but in a separate file (orm.xml) separating persistence mapping from your model. What follows provides a reference guide to MetaData elements. Here is an example header for orm.xml files with XSD specification

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<entity-mappings xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
    http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd" version="2.1">
    ...
</entity-mappings>
```

If using any of the DataNucleus extensions, then the XSD is defined here, in which case you would define your header as :-

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<entity-mappings xmlns="http://www.datanucleus.org/xsd/jpa/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.datanucleus.org/xsd/jpa/orm
    http://www.datanucleus.org/xsd/jpa/orm_2_1.xsd" version="2.1">
    ...
</entity-mappings>
```

- entity-mappings
  - description
  - persistence-unit-metadata
    - xml-mapping-metadata-complete
  - package
  - schema
  - catalog
  - access
  - sequence-generator
  - table-generator
  - named-query
    - query
    - named-native-query
  - query
  - sql-result-set-mapping
    - entity-result
    - field-result
• column-result
• mapped-superclass
  • description
  • id-class
  • exclude-default-listeners
  • exclude-superclass-listeners
  • entity-listeners
    • entity-listener
      • pre-persist
      • post-persist
      • pre-remove
      • post-remove
      • pre-update
      • post-update
      • post-load
    • pre-persist
    • post-persist
    • pre-remove
    • post-remove
    • pre-update
    • post-update
    • post-load
  • attributes
    • Same elements as under <entity>-<attributes>
• entity
  • description
  • table
    • unique-constraint
      • column-name
    • index
  • secondary-table
    • primary-key-join-column
    • primary-key-foreign-key
    • unique-constraint
      • column-name
      • index
    • primary-key-join-column
    • primary-key-foreign-key
  • id-class
• inheritance
• discriminator-value
• discriminator-column
• sequence-generator
• table-generator
  • index
• named-query
  • query
• named-native-query
  • query
• sql-result-set-mapping
  • entity-result
    • field-result
    • column-result
• named-entity-graph
  • named-attribute-node
  • subgraph
    • named-attribute-node
    • subclass-subgraph
  • named-attribute-node
• exclude-default-listeners
• exclude-superclass-listeners
• entity-listeners
  • entity-listener
    • pre-persist
    • post-persist
    • pre-remove
    • post-remove
    • pre-update
    • post-update
    • post-load
• pre-persist
• post-persist
• pre-remove
• post-remove
• pre-update
• post-update
• post-load
• attribute-override
  • column
• association-override
  • join-column
• attributes
  • id
    • column
    • generated-value
    • sequence-generator
    • table-generator
• embedded-id
• basic
  • column
  • lob
  • temporal
  • enumerated
  • convert
• version
  • column
• many-to-one
  • join-column
  • join-table
    • join-column
    • inverse-join-column
    • unique-constraint
      • column-name
• cascade
  • cascade-all
  • cascade-persist
  • cascade-merge
  • cascade-remove
  • cascade-refresh
• element-collection
  • collection-table
    • join-column
    • index
    • foreign-key
- order-by
- order-column
- map-key
- map-key-temporal
- map-key enumer ated
- join-table
  - join-column
  - foreign-key
  - inverse-join-column
  - inverse-foreign-key
  - unique-constraint
    - column-name
- join-column
- one-to-many
  - order-by
  - order-column
  - map-key
  - map-key-temporal
  - map-key enumer ated
  - join-table
    - join-column
    - inverse-join-column
    - unique-constraint
      - column-name
  - join-column
- cascade
  - cascade-all
  - cascade-persist
  - cascade-merge
  - cascade-remove
  - cascade-refresh
- one-to-one
  - join-column
  - foreign-key
  - join-table
    - join-column
    - inverse-join-column
    - unique-constraint
      - column-name
• cascade
  • cascade-all
  • cascade-persist
  • cascade-merge
  • cascade-remove
  • cascade-refresh
• many-to-many
  • order-by
  • order-column
  • map-key
  • map-key-temporal
  • map-key-enumerated
  • join-table
    • join-column
    • inverse-join-column
    • unique-constraint
    • column-name
• cascade
  • cascade-all
  • cascade-persist
  • cascade-merge
  • cascade-remove
  • cascade-refresh
• embedded
• attribute-override
• transient
• embeddable
  • embeddable-attributes
    • basic
    • transient

148.1.1 Metadata for description tag
The <description> element (<entity-mappings>) contains the text describing all classes (and hence entities) defined in this file. It serves no useful purpose other than descriptive.
148.1.2 Metadata for xml-mapping-metadata-complete tag
The `<xml-mapping-metadata-complete>` element (under `<persistence-unit-metadata>`) when specified defines that the classes in this file are fully specified with just their metadata and that any annotations should be ignored.

148.1.3 Metadata for package tag
The `<package>` element (under `<entity-mappings>`) contains the text defining the package into which all classes in this file belong.

148.1.4 Metadata for schema tag
The `<schema>` element (under `<entity-mappings>`) contains the default schema for all classes in this file.

148.1.5 Metadata for catalog tag
The `<catalog>` element (under `<entity-mappings>`) contains the default catalog for all classes in this file.

148.1.6 Metadata for access tag
The `<access>` element (under `<entity-mappings>`) contains the setting for how to access the persistent fields/properties. This can be set to either "FIELD" or "PROPERTY".

148.1.7 Metadata for sequence-generator tag
The `<sequence-generator>` element (under `<entity-mappings>`, or `<entity>` or `<id>`) defines a generator of sequence values, for use elsewhere in this persistence-unit.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the generator (required)</td>
<td></td>
</tr>
<tr>
<td>sequence-name</td>
<td>Name of the sequence</td>
<td></td>
</tr>
<tr>
<td>initial-value</td>
<td>Initial value for the sequence</td>
<td>1</td>
</tr>
<tr>
<td>allocation-size</td>
<td>Number of values that the sequence allocates when needed</td>
<td>50</td>
</tr>
</tbody>
</table>

148.1.8 Metadata for table-generator tag
The `<table-generator>` element (under `<entity-mappings>`, or `<entity>` or `<id>`) defines a generator of sequence values using a datastore table, for use elsewhere in this persistence-unit.
### 148.1.9 Metadata for named-query tag

The `<named-query>` element (under `<entity-mappings>` or under `<entity>`) defines a JPQL query that will be accessible at runtime via the name. The element itself will contain the text of the query. It has the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the query</td>
<td></td>
</tr>
</tbody>
</table>

### 148.1.10 Metadata for named-native-query tag

The `<named-native-query>` element (under `<entity-mappings>` or under `<entity>`) defines an SQL query that will be accessible at runtime via the name. The element itself will contain the text of the query. It has the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the query</td>
<td></td>
</tr>
</tbody>
</table>

### 148.1.11 Metadata for sql-result-set-mapping tag

The `<sql-result-set-mapping>` element (under `<entity-mappings>` or under `<entity>`) defines how the results of the SQL query are output to the user per row of the result set. It will contain sub-elements. It has the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
</table>
### 148.1.12 Metadata for named-entity-graph tag

The `<named-entity-graph>` element (under `<entity>`) defines an entity graph with root as that entity, accessible at runtime via the name. It has the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the entity graph</td>
<td></td>
</tr>
</tbody>
</table>

### 148.1.13 Metadata for named-attribute-node tag

The `<named-attribute-node>` element (under `<named-entity-graph>`) defines a node in the entity graph. It has the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the node (field/property)</td>
<td></td>
</tr>
<tr>
<td>subgraph</td>
<td>Name of a subgraph that maps this attribute fully (optional)</td>
<td></td>
</tr>
</tbody>
</table>

### 148.1.14 Metadata for subgraph/subclass-subgraph tag

The `<subgraph>`/ `<subclass-subgraph>` element (under `<named-entity-graph>`) defines a subgraph in the entity graph. It has the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the subgraph (referenced in the named-attribute-node)</td>
<td></td>
</tr>
<tr>
<td>class</td>
<td>Type of the subgraph attribute</td>
<td></td>
</tr>
</tbody>
</table>

### 148.1.15 Metadata for entity-result tag

The `<entity-result>` element (under `<sql-result-set-mapping>`) defines an entity that is output from an SQL query per row of the result set. It can contain sub-elements of type `<field-result>`. It has the following attributes:
148.1.16 Metadata for field-result tag

The `<field-result>` element (under `<entity-result>`) defines a field of an entity and the column representing it in an SQL query. It has the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the entity field</td>
<td></td>
</tr>
<tr>
<td>column</td>
<td>Name of the SQL column</td>
<td></td>
</tr>
</tbody>
</table>

148.1.17 Metadata for column-result tag

The `<column-result>` element (under `<sql-result-set-mapping>`) defines a column that is output directly from an SQL query per row of the result set. It has the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the SQL column</td>
<td></td>
</tr>
</tbody>
</table>

148.1.18 Metadata for mapped-superclass tag

These are attributes within the `<mapped-superclass>` tag (under `<entity-mappings>`). This is used to define the persistence definition for a class that has no table but is mapped:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>Name of the class (required)</td>
<td></td>
</tr>
<tr>
<td>metadata-complete</td>
<td>Whether the definition of persistence of this class is complete with this MetaData definition. That is, should any annotations be ignored.</td>
<td>true</td>
</tr>
</tbody>
</table>

148.1.19 Metadata for entity tag

These are attributes within the `<entity>` tag (under `<entity-mappings>`). This is used to define the persistence definition for this class.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>Name of the class (required)</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>Name of the entity. Used by JPQL queries</td>
<td></td>
</tr>
<tr>
<td>metadata-complete</td>
<td>Whether the definition of persistence of this class is complete with this MetaData definition. That is, should any annotations be ignored.</td>
<td>true</td>
</tr>
<tr>
<td>cacheable</td>
<td>Whether instances of this class should be cached in the L2 cache. New in JPA2.</td>
<td>true</td>
</tr>
</tbody>
</table>

### 148.1.20 Metadata for description tag

The `<description>` element (under `<entity>`) contains the text describing the class being persisted. It serves no useful purpose other than descriptive.

### 148.1.21 Metadata for table tag

These are attributes within the `<table>` tag (under `<entity>`). This is used to define the table where this class will be persisted.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the table</td>
<td></td>
</tr>
<tr>
<td>catalog</td>
<td>Catalog where the table is stored</td>
<td></td>
</tr>
<tr>
<td>schema</td>
<td>Schema where the table is stored</td>
<td></td>
</tr>
</tbody>
</table>

### 148.1.22 Metadata for secondary-table tag

These are attributes within the `<secondary-table>` tag (under `<entity>`). This is used to define the join of a secondary table back to the primary table where this class will be persisted.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the table</td>
<td></td>
</tr>
<tr>
<td>catalog</td>
<td>Catalog where the table is stored</td>
<td></td>
</tr>
<tr>
<td>schema</td>
<td>Schema where the table is stored</td>
<td></td>
</tr>
</tbody>
</table>
148.1.23 Metadata for join-table tag
These are attributes within the `<join-table>` tag (under `<one-to-one`>, `<one-to-many`>, `<many-to-many>`). This is used to define the join table where a collection/maps relationship will be persisted.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the join table</td>
<td></td>
</tr>
<tr>
<td>catalog</td>
<td>Catalog where the join table is stored</td>
<td></td>
</tr>
<tr>
<td>schema</td>
<td>Schema where the join table is stored</td>
<td></td>
</tr>
<tr>
<td>orphan-removal</td>
<td>Whether to remove orphans when either removing the owner or nulling the relation</td>
<td>false</td>
</tr>
</tbody>
</table>

148.1.24 Metadata for collection-table tag
These are attributes within the `<collection-table>` tag (under `<element-collection>`). This is used to define the join table where a collections relationship will be persisted.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the join table</td>
<td></td>
</tr>
<tr>
<td>catalog</td>
<td>Catalog where the join table is stored</td>
<td></td>
</tr>
<tr>
<td>schema</td>
<td>Schema where the join table is stored</td>
<td></td>
</tr>
</tbody>
</table>

148.1.25 Metadata for unique-constraint tag
This element is specified under the `<table`>, `<secondary-table`> or `<join-table`> tags. This is used to define a unique constraint on the table. No attributes are provided, just sub-element(s) "column-name"

148.1.26 Metadata for column tag
These are attributes within the `<column>` tag (under `<basic>`). This is used to define the column where the data will be stored.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the column</td>
<td></td>
</tr>
<tr>
<td>unique</td>
<td>Whether the column is unique</td>
<td>true</td>
</tr>
<tr>
<td>nullable</td>
<td>Whether the column is nullable</td>
<td>true</td>
</tr>
<tr>
<td>insertable</td>
<td>Whether the column is insertable</td>
<td>true</td>
</tr>
</tbody>
</table>
### updatable
- **Description:** Whether the column is updatable
- **Values:** `true` | `false`

### column-definition
- **Description:** Some vague JPA term that you put anything in and get any unexpected results from

### table
- **Description:** Table for the column

### length
- **Description:** Length for the column (when string type)
- **Values:** `255`

### precision
- **Description:** Precision for the column (when numeric type)
- **Values:** `0`

### scale
- **Description:** Scale for the column (when numeric type)
- **Values:** `0`

### jdbc-type
- **Description:** The JDBC Type to use for this column *(DataNucleus extension)*

### position
- **Description:** The position to use for this column *(first=0) (DataNucleus extension)*

### 148.1.27 Metadata for primary-key-join-column tag
These are attributes within the `<primary-join-key-column>` tag (under `<secondary-table>` or `<entity>`). This is used to define the join of PK columns between secondary and primary tables, or between table of subclass and table of base class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the column</td>
<td></td>
</tr>
<tr>
<td>referenced-column-name</td>
<td>Name of column in primary table</td>
<td></td>
</tr>
</tbody>
</table>

### 148.1.28 Metadata for join-column tag
These are attributes within the `<join-column>` tag (under `<join-table>`). This is used to define the join column.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the column</td>
<td></td>
</tr>
<tr>
<td>referenced-column-name</td>
<td>Name of the column at the other side of the relation that this is a FK to</td>
<td></td>
</tr>
<tr>
<td>unique</td>
<td>Whether the column is unique</td>
<td><code>true</code></td>
</tr>
<tr>
<td>nullable</td>
<td>Whether the column is nullable</td>
<td><code>true</code></td>
</tr>
<tr>
<td>insertable</td>
<td>Whether the column is insertable</td>
<td><code>true</code></td>
</tr>
<tr>
<td>updatable</td>
<td>Whether the column is updatable</td>
<td><code>true</code></td>
</tr>
</tbody>
</table>
148.1.29 Metadata for inverse-join-column tag
These are attributes within the `<inverse-join-column>` tag (under `<join-table>`). This is used to define the join column to the element.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the column</td>
<td></td>
</tr>
<tr>
<td>referenced-column-name</td>
<td>Name of the column at the other side of the relation that this is a FK to</td>
<td></td>
</tr>
<tr>
<td>unique</td>
<td>Whether the column is unique</td>
<td>true</td>
</tr>
<tr>
<td>nullable</td>
<td>Whether the column is nullable</td>
<td>true</td>
</tr>
<tr>
<td>insertable</td>
<td>Whether the column is insertable</td>
<td>true</td>
</tr>
<tr>
<td>updatable</td>
<td>Whether the column is updatable</td>
<td>true</td>
</tr>
<tr>
<td>column-definition</td>
<td>Some vague JPA term that you put anything in and get any unexpected results from. Not supported by DataNucleus.</td>
<td></td>
</tr>
<tr>
<td>table</td>
<td>Table for the column ?</td>
<td></td>
</tr>
</tbody>
</table>

148.1.30 Metadata for id-class tag
These are attributes within the `<id-class>` tag (under `<entity>`). This defines a identity class to be used for this entity.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>Name of the identity class</td>
<td>(required)</td>
</tr>
</tbody>
</table>

148.1.31 Metadata for inheritance tag
These are attributes within the `<inheritance>` tag (under `<entity>`). This defines the inheritance of the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
</table>
### 148.1.32 Metadata for discriminator-value tag

These are attributes within the `<discriminator-value>` tag (under `<entity>`). This defines the value used in a discriminator. The value is contained in the element. Specification of the value will result in a "value-map" discriminator strategy being adopted. If no discriminator-value is present, but discriminator-column is then "class-name" discriminator strategy is used.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the discriminator column</td>
<td>DTYPE</td>
</tr>
<tr>
<td>discriminator-type</td>
<td>Type of data stored in the discriminator column</td>
<td>STRING</td>
</tr>
<tr>
<td>length</td>
<td>Length of the discriminator column</td>
<td></td>
</tr>
</tbody>
</table>

### 148.1.33 Metadata for discriminator-column tag

These are attributes within the `<discriminator-column>` tag (under `<entity>`). This defines the column used for a discriminator.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the field (required)</td>
<td></td>
</tr>
</tbody>
</table>

### 148.1.34 Metadata for id tag

These are attributes within the `<id>` tag (under `<attributes>`). This is used to define the field used to be the identity of the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the field (required)</td>
<td></td>
</tr>
</tbody>
</table>

### 148.1.35 Metadata for generated-value tag

These are attributes within the `<generated-value>` tag (under `<id>`). This is used to define how to generate the value for the identity field.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>strategy</td>
<td>Generation strategy. Please refer to the Identity Generation Guide</td>
<td>auto</td>
</tr>
</tbody>
</table>
148.1.36 Metadata for embedded-id tag

These are attributes within the `<embedded-id>` tag (under `<attributes>`). This is used to define the field used to be the (embedded) identity of the class. **Note that this is not yet fully supported - specify the fields in the class**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the field (required)</td>
<td></td>
</tr>
</tbody>
</table>

148.1.37 Metadata for version tag

These are attributes within the `<version>` tag (under `<attributes>`). This is used to define the field used to be hold the version of the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the field (required)</td>
<td></td>
</tr>
</tbody>
</table>

148.1.38 Metadata for basic tag

These are attributes within the `<basic>` tag (under `<attributes>`). This is used to define the persistence information for the field.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the field (required)</td>
<td></td>
</tr>
<tr>
<td>fetch</td>
<td>Fetch type for this field</td>
<td>LAZY</td>
</tr>
<tr>
<td>optional</td>
<td>Whether this field may be null and may be used in schema generation</td>
<td>true</td>
</tr>
</tbody>
</table>

148.1.39 Metadata for temporal tag

These are attributes within the `<temporal>` tag (under `<basic>`). This is used to define the details of persistence as a temporal type. The contents of the element can be one of DATE, TIME, TIMESTAMP.
148.1.40 Metadata for enumerated tag
These are attributes within the `<enumerated>` tag (under `<basic>`). This is used to define the details of persistence as an enum type. The contents of the element can be one of ORDINAL or STRING to represent whether the enum is persisted as an integer-based or the actual string.

148.1.41 Metadata for one-to-one tag
These are attributes within the `<one-to-one>` tag (under `<attributes>`). This is used to define that the field is part of a 1-1 relation.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the field (required)</td>
<td></td>
</tr>
<tr>
<td>target-entity</td>
<td>Class name of the related entity</td>
<td></td>
</tr>
<tr>
<td>fetch</td>
<td>Whether the field should be fetched immediately</td>
<td>EAGER</td>
</tr>
<tr>
<td>optional</td>
<td>Whether the field can store nulls.</td>
<td>true</td>
</tr>
<tr>
<td>mapped-by</td>
<td>Name of the field that owns the relation (specified on the inverse side)</td>
<td></td>
</tr>
</tbody>
</table>

148.1.42 Metadata for many-to-one tag
These are attributes within the `<many-to-one>` tag (under `<attributes>`). This is used to define that the field is part of a N-1 relation.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the field (required)</td>
<td></td>
</tr>
<tr>
<td>target-entity</td>
<td>Class name of the related entity</td>
<td></td>
</tr>
<tr>
<td>fetch</td>
<td>Whether the field should be fetched immediately</td>
<td>EAGER</td>
</tr>
<tr>
<td>optional</td>
<td>Whether the field can store nulls.</td>
<td>true</td>
</tr>
</tbody>
</table>

148.1.43 Metadata for element-collection tag
These are attributes within the `<element-collection>` tag (under `<attributes>`). This is used to define that the field is part of a 1-N non-PC relation.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the field (required)</td>
<td></td>
</tr>
<tr>
<td>target-class</td>
<td>Class name of the related object</td>
<td></td>
</tr>
</tbody>
</table>
148.1.44 Metadata for one-to-many tag
These are attributes within the `<one-to-many>` tag (under `<attributes>`). This is used to define that the field is part of a 1-N relation.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the field (required)</td>
<td></td>
</tr>
<tr>
<td>target-entity</td>
<td>Class name of the related entity</td>
<td></td>
</tr>
<tr>
<td>fetch</td>
<td>Whether the field should be fetched immediately</td>
<td>EAGER</td>
</tr>
<tr>
<td>mapped-by</td>
<td>Name of the field that owns the relation (specified on the inverse side)</td>
<td></td>
</tr>
<tr>
<td>orphan-removal</td>
<td>Whether to remove orphans when either removing the owner or removing the element</td>
<td>false</td>
</tr>
</tbody>
</table>

148.1.45 Metadata for many-to-many tag
These are attributes within the `<many-to-many>` tag (under `<attributes>`). This is used to define that the field is part of a M-N relation.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the field (required)</td>
<td></td>
</tr>
<tr>
<td>target-entity</td>
<td>Class name of the related entity</td>
<td></td>
</tr>
<tr>
<td>fetch</td>
<td>Whether the field should be fetched immediately</td>
<td>EAGER</td>
</tr>
<tr>
<td>mapped-by</td>
<td>Name of the field on the non-owning side that completes the relation. Specified on the owner side</td>
<td></td>
</tr>
</tbody>
</table>

148.1.46 Metadata for embedded tag
These are attributes within the `<embedded>` tag (under `<attributes>`). This is used to define that the field is part of an embedded relation.
148.1.47 Metadata for order-by tag

This element is specified under `<one-to-many>` or `<many-to-many>`. It is used to define the field(s) of the element class that is used for ordering the elements when they are retrieved from the datastore. It has no attributes and the ordering is specified within the element itself. It should be a comma-separated list of field names (of the element) with optional "ASC" or "DESC" to signify ascending or descending.

148.1.48 Metadata for order-column tag

This element is specified under `<one-to-many>` or `<many-to-many>`. It is used to define that the List will be ordered with the ordering stored in a surrogate column in the other table.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the column</td>
<td>{fieldName}_ORDER</td>
</tr>
<tr>
<td>nullable</td>
<td>Whether the column is nullable</td>
<td>true</td>
</tr>
<tr>
<td>insertable</td>
<td>Whether the column is insertable</td>
<td>true</td>
</tr>
<tr>
<td>updatable</td>
<td>Whether the column is updatable</td>
<td>true</td>
</tr>
<tr>
<td>column-definition</td>
<td>Some vague JPA term that you put anything in and get any unexpected results from</td>
<td></td>
</tr>
<tr>
<td>base</td>
<td>Origin of the ordering (value for the first element)</td>
<td>0</td>
</tr>
</tbody>
</table>

148.1.49 Metadata for map-key tag

These are attributes within the `<map-key>` tag (under `<one-to-many>` or `<many-to-many>`). This is used to define the field of the value class that is the key of a Map.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the field (required)</td>
<td></td>
</tr>
</tbody>
</table>

148.1.50 Metadata for map-key-temporal tag

Within the `<map-key-temporal>` tag (under `<element-collection>`, `<one-to-many>` or `<many-to-many>`) you put the TemporalType value.
148.1.51 Metadata for map-key-enumerated tag
Within the `<map-key-enumerated>` tag (under `<element-collection>`, `<one-to-many>` or `<many-to-many>`) you put the EnumType value.

148.1.52 Metadata for transient tag
These are attributes within the `<transient>` tag (under `<attributes>`). This is used to define that the field is not to be persisted.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the field (required)</td>
<td></td>
</tr>
</tbody>
</table>

148.1.53 Metadata for index tag
These are attributes within the `<index>` element. This is used to define the details of an index when overriding the provider default.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the index</td>
<td></td>
</tr>
<tr>
<td>unique</td>
<td>Whether the index is unique</td>
<td></td>
</tr>
<tr>
<td>column-list</td>
<td>List of columns (including any ASC, DESC specifications for each column)</td>
<td></td>
</tr>
</tbody>
</table>

148.1.54 Metadata for foreign-key tag
These are attributes within the `<foreign-key>` element. This is used to define the details of a foreign-key when overriding the provider default.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the foreign-key</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>Constraint mode</td>
<td></td>
</tr>
<tr>
<td>foreignKeyDefinition</td>
<td>The DDL for the foreign key</td>
<td></td>
</tr>
</tbody>
</table>

148.1.55 Metadata for convert tag
These are attributes within the `<convert>` element, under `<basic>`. This is used to define the use of type conversion on this field.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>converter</td>
<td>Class name of the converter</td>
<td></td>
</tr>
<tr>
<td>attribute-name</td>
<td>Name of the embedded field to convert (optional). <strong>Not yet supported</strong></td>
<td></td>
</tr>
<tr>
<td>disable-conversion</td>
<td>Whether to disable any auto-apply converters for this field</td>
<td>true</td>
</tr>
</tbody>
</table>

148.1.56 Metadata for exclude-default-listeners tag

This element is specified under `<mapped-superclass>` or `<entity>` and is used to denote that any default listeners defined in this file will be ignored.

148.1.57 Metadata for exclude-superclass-listeners tag

This element is specified under `<mapped-superclass>` or `<entity>` and is used to denote that any listeners of superclasses will be ignored.

148.1.58 Metadata for entity-listener tag

These are attributes within the `<entity-listener>` tag (under `<entity-listeners>`). This is used to an EntityListener class and the methods it uses

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>Name of the EntityListener class that receives the callbacks for this Entity</td>
<td></td>
</tr>
</tbody>
</table>

148.1.59 Metadata for pre-persist tag

These are attributes within the `<pre-persist>` tag (under `<entity>`). This is used to define any "PrePersist" method callback.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>method-name</td>
<td>Name of the method (required)</td>
<td></td>
</tr>
</tbody>
</table>

148.1.60 Metadata for post-persist tag

These are attributes within the `<post-persist>` tag (under `<entity>`). This is used to define any "PostPersist" method callback.
**148.1.61 Metadata for pre-remove tag**

These are attributes within the `<pre-remove>` tag (under `<entity>`). This is used to define any "PreRemove" method callback.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>method-name</td>
<td>Name of the method (required)</td>
<td></td>
</tr>
</tbody>
</table>

**148.1.62 Metadata for post-remove tag**

These are attributes within the `<post-remove>` tag (under `<entity>`). This is used to define any "PostRemove" method callback.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>method-name</td>
<td>Name of the method (required)</td>
<td></td>
</tr>
</tbody>
</table>

**148.1.63 Metadata for pre-update tag**

These are attributes within the `<pre-update>` tag (under `<entity>`). This is used to define any "PreUpdate" method callback.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>method-name</td>
<td>Name of the method (required)</td>
<td></td>
</tr>
</tbody>
</table>

**148.1.64 Metadata for post-update tag**

These are attributes within the `<post-update>` tag (under `<entity>`). This is used to define any "PostUpdate" method callback.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>method-name</td>
<td>Name of the method (required)</td>
<td></td>
</tr>
</tbody>
</table>

**148.1.65 Metadata for post-load tag**

These are attributes within the `<post-load>` tag (under `<entity>`). This is used to define any "PostLoad" method callback.
### 148.1.66 Metadata for attribute-override tag

These are attributes within the `<attribute-override>` tag (under `<entity>`). This is used to override the columns for any fields in superclasses.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the field/property (required)</td>
<td></td>
</tr>
</tbody>
</table>

### 148.1.67 Metadata for association-override tag

These are attributes within the `<association-override>` tag (under `<entity>`). This is used to override the columns for any N-1/1-1 fields in superclasses.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the field/property (required)</td>
<td></td>
</tr>
</tbody>
</table>
149 Annotations

149.1 JPA : Annotations
One of the things that JDK 1.5 provides that can be of some use to DataNucleus is annotations. DataNucleus supports both JPA and JDO annotations. In this section we will document some of the more important JPA annotations. When selecting to use annotations please bear in mind the following :-

- You must have the datanucleus-api-jpa jar available in your CLASSPATH.
- You must have the persistence-api jar in your CLASSPATH since this provides the annotations
- Annotations should really only be used for attributes of persistence that you won't be changing at deployment. Things such as table and column names shouldn't really be specified using annotations although it is permitted. Instead it would be better to put such information in an ORM file.
- Annotations can be added in two places - for the class as a whole, or for a field in particular.
- You can annotate fields or getters with field-level information. It doesn't matter which.
- Annotations are prefixed by the @ symbol and can take properties (in brackets after the name, comma-separated)
- JPA doesn't provide for some key JDO concepts and DataNucleus provides its own annotations for these cases.
- You have to import "javax.persistence.XXX" where XXX is the annotation name of a JPA annotation
- You have to import "org.datanucleus.api.jpa.annotations.XXX" where XXX is the annotation name of a DataNucleus value-added annotation

Annotations supported by DataNucleus are shown below. Not all have their documentation written yet. The annotations/attributes coloured in pink are ORM and really should be placed in XML rather than directly in the class using annotations. The annotations coloured in blue are DataNucleus extensions and should be used only where you dont mind losing implementation-independence.

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Class/Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Entity</td>
<td>Class</td>
<td>Specifies that the class is persistent</td>
</tr>
<tr>
<td>@MappedSuperclass</td>
<td>Class</td>
<td>Specifies that this class contains persistent information to be mapped</td>
</tr>
<tr>
<td>@Embeddable</td>
<td>Class</td>
<td>Specifies that the class is persistent embedded in another persistent class</td>
</tr>
<tr>
<td>@PersistenceAware</td>
<td>Class</td>
<td>Specifies that the class is not persistent but needs to be able to access fields of persistent classes</td>
</tr>
<tr>
<td>@IdClass</td>
<td>Class</td>
<td>Defines the primary key class for this class</td>
</tr>
<tr>
<td>@Cacheable</td>
<td>Class</td>
<td>Specifies that instances of this class can be cached in the L2 cache</td>
</tr>
<tr>
<td>@DatastoreIdentity</td>
<td>Class</td>
<td>Defines a class as using datastore-identity (DataNucleus extension).</td>
</tr>
<tr>
<td>Annotation</td>
<td>Class</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>@EntityListeners</td>
<td>Class</td>
<td>Specifies class(es) that are listeners for events from instances of this class</td>
</tr>
<tr>
<td>@NamedQueries</td>
<td>Class</td>
<td>Defines a series of named JPQL queries for use in the current persistence unit</td>
</tr>
<tr>
<td>@NamedQuery</td>
<td>Class</td>
<td>Defines a named JPQL query for use in the current persistence unit</td>
</tr>
<tr>
<td>@NamedNativeQuery</td>
<td>Class</td>
<td>Defines a named SQL query for use in the current persistence unit</td>
</tr>
<tr>
<td>@NamedNativeQueries</td>
<td>Class</td>
<td>Defines a series of named SQL queries for use in the current persistence unit</td>
</tr>
<tr>
<td>@NamedStoredProcedureQuery</td>
<td>Class</td>
<td>Defines a named stored procedure query for use in the current persistence unit</td>
</tr>
<tr>
<td>@NamedStoredProcedureQueries</td>
<td>Class</td>
<td>Defines a series of named stored procedure queries for use in the current persistence unit</td>
</tr>
<tr>
<td>@SqlResultSetMapping</td>
<td>Class</td>
<td>Defines a result mapping for an SQL query for use in the current persistence unit</td>
</tr>
<tr>
<td>@SqlResultSetMappings</td>
<td>Class</td>
<td>Defines a series of mappings for SQL queries for use in the current persistence unit</td>
</tr>
<tr>
<td>@NamedEntityGraph</td>
<td>Class</td>
<td>Defines a named entity graph with root of the class it is specified on</td>
</tr>
<tr>
<td>@NamedEntityGraphs</td>
<td>Class</td>
<td>Defines named entity graphs with root of the class it is specified on</td>
</tr>
<tr>
<td>@Converter</td>
<td>Class</td>
<td>Defines a java type converter for a field type</td>
</tr>
<tr>
<td>@Inheritance</td>
<td>Class</td>
<td>Specifies the inheritance model for persisting this class</td>
</tr>
<tr>
<td>@Table</td>
<td>Class</td>
<td>Defines the table where this class will be stored</td>
</tr>
<tr>
<td>@SecondaryTable</td>
<td>Class</td>
<td>Defines a secondary table where some fields of this class will be stored</td>
</tr>
<tr>
<td>@DiscriminatorColumn</td>
<td>Class</td>
<td>Defines the column where any discriminator will be stored</td>
</tr>
<tr>
<td>@DiscriminatorValue</td>
<td>Class</td>
<td>Defines the value to be used in the discriminator for objects of this class</td>
</tr>
<tr>
<td>@PrimaryKeyJoinColumns</td>
<td>Class</td>
<td>Defines the names of the PK columns when this class has a superclass</td>
</tr>
<tr>
<td>@PrimaryKeyJoinColumn</td>
<td>Class</td>
<td>Defines the name of the PK column when this class has a superclass</td>
</tr>
<tr>
<td>Annotation</td>
<td>Target</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>@AttributeOverride</td>
<td>Class</td>
<td>Defines a field in a superclass that will have its column overridden</td>
</tr>
<tr>
<td>@AttributeOverrides</td>
<td>Class</td>
<td>Defines the field(s) of superclasses that will have their columns overridden</td>
</tr>
<tr>
<td>@AssociationOverride</td>
<td>Class</td>
<td>Defines a N-1/1-1 field in a superclass that will have its column overridden</td>
</tr>
<tr>
<td>@AssociationOverrides</td>
<td>Class</td>
<td>Defines the N-1/1-1 field(s) of superclasses that will have their columns overridden</td>
</tr>
<tr>
<td>@SequenceGenerator</td>
<td>Class/Field/Method</td>
<td>Defines a generator of values using sequences in the datastore for use with persistent entities</td>
</tr>
<tr>
<td>@TableGenerator</td>
<td>Class/Field/Method</td>
<td>Defines a generator of sequences using a table in the datastore for use with persistent entities</td>
</tr>
<tr>
<td>@Embedded</td>
<td>Field/Method</td>
<td>Defines this field as being embedded</td>
</tr>
<tr>
<td>@Id</td>
<td>Field/Method</td>
<td>Defines this field as being (part of) the identity for the class</td>
</tr>
<tr>
<td>@EmbeddedId</td>
<td>Field/Method</td>
<td>Defines this field as being (part of) the identity for the class, and being embedded into this class.</td>
</tr>
<tr>
<td>@Version</td>
<td>Field/Method</td>
<td>Defines this field as storing the version for the class</td>
</tr>
<tr>
<td>@Basic</td>
<td>Field/Method</td>
<td>Defines this field as being persistent</td>
</tr>
<tr>
<td>@Transient</td>
<td>Field/Method</td>
<td>Defines this field as being transient (not persisted)</td>
</tr>
<tr>
<td>@OneToMany</td>
<td>Field/Method</td>
<td>Defines this field as being a 1-N relation with other persistent entities</td>
</tr>
<tr>
<td>@ManyToOne</td>
<td>Field/Method</td>
<td>Defines this field as being a N-1 relation with another persistent entity</td>
</tr>
<tr>
<td>@ManyToMany</td>
<td>Field/Method</td>
<td>Defines this field as being a M-N relation with other persistent entities</td>
</tr>
<tr>
<td>@ManyToOne</td>
<td>Field/Method</td>
<td>Defines this field as being a N-1 relation with another persistent entity</td>
</tr>
<tr>
<td>@ElementCollection</td>
<td>Field/Method</td>
<td>Defines this field as being a 1-N relation of Objects that are not Entities.</td>
</tr>
<tr>
<td>@Index</td>
<td>Field/Method</td>
<td>Specifies an index on this field/property (DataNucleus extension).</td>
</tr>
<tr>
<td>@JdbcType</td>
<td>Field/Method</td>
<td>Specifies the JDBC Type to use on this field/property (DataNucleus extension).</td>
</tr>
<tr>
<td>Annotation</td>
<td>Field/Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>@ColumnPosition</td>
<td>Field/Method</td>
<td>Specifies the column position to use on this field/property (DataNucleus extension).</td>
</tr>
<tr>
<td>@GeneratedValue</td>
<td>Field/Method</td>
<td>Defines that this field has its value generated using a generator</td>
</tr>
<tr>
<td>@MapKey</td>
<td>Field/Method</td>
<td>Defines that this field is the key to a map</td>
</tr>
<tr>
<td>@MapKeyEnumerated</td>
<td>Field/Method</td>
<td>Defines the datastore type for the map key when it is an enum</td>
</tr>
<tr>
<td>@MapKeyTemporal</td>
<td>Field/Method</td>
<td>Defines the datastore type for the map key when it is a temporal type</td>
</tr>
<tr>
<td>@OrderBy</td>
<td>Field/Method</td>
<td>Defines the field(s) used for ordering the elements in this collection</td>
</tr>
<tr>
<td>@OrderColumn</td>
<td>Field/Method</td>
<td>Defines that ordering should be attributed by the implementation using a surrogate column.</td>
</tr>
<tr>
<td>@PrePersist</td>
<td>Field/Method</td>
<td>Defines this method as being a callback for pre-persist events</td>
</tr>
<tr>
<td>@PostPersist</td>
<td>Field/Method</td>
<td>Defines this method as being a callback for post-persist events</td>
</tr>
<tr>
<td>@PreRemove</td>
<td>Field/Method</td>
<td>Defines this method as being a callback for pre-remove events</td>
</tr>
<tr>
<td>@PostRemove</td>
<td>Field/Method</td>
<td>Defines this method as being a callback for post-remove events</td>
</tr>
<tr>
<td>@PreUpdate</td>
<td>Field/Method</td>
<td>Defines this method as being a callback for pre-update events</td>
</tr>
<tr>
<td>@PostUpdate</td>
<td>Field/Method</td>
<td>Defines this method as being a callback for post-update events</td>
</tr>
<tr>
<td>@PostLoad</td>
<td>Field/Method</td>
<td>Defines this method as being a callback for post-load events</td>
</tr>
<tr>
<td>@JoinTable</td>
<td>Field/Method</td>
<td>Defines this field as being stored using a join table</td>
</tr>
<tr>
<td>@CollectionTable</td>
<td>Field/Method</td>
<td>Defines this field as being stored using a join table when containing non-entity objects.</td>
</tr>
<tr>
<td>@Lob</td>
<td>Field/Method</td>
<td>Defines this field as being stored as a large object</td>
</tr>
<tr>
<td>@Temporal</td>
<td>Field/Method</td>
<td>Defines this field as storing temporal data</td>
</tr>
<tr>
<td>@Enumerated</td>
<td>Field/Method</td>
<td>Defines this field as storing enumerated data</td>
</tr>
<tr>
<td>@Convert</td>
<td>Field/Method</td>
<td>Defines a converter for this field/property</td>
</tr>
<tr>
<td>@Column</td>
<td>Field/Method</td>
<td>Defines the column where this field is stored</td>
</tr>
</tbody>
</table>
149.1.1 @Entity
This annotation is used when you want to mark a class as persistent. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name of the entity (used in JPQL to refer to the class)</td>
<td></td>
</tr>
</tbody>
</table>

```java
@Entity
public class MyClass {
    ...
}
```

See the documentation for Class Mapping

149.1.2 @MappedSuperclass
This annotation is used when you want to mark a class as persistent but without a table of its own and being the superclass of the class that has a table, meaning that all of its fields are persisted into the table of its subclass. Specified on the class.

```java
@MappedSuperclass
public class MyClass {
    ...
}
```

See the documentation for Inheritance
149.1.3 @PersistenceAware

This annotation is used when you want to mark a class as knowing about persistence but not persistent itself. That is, it manipulates the fields of a persistent class directly rather than using accessors. **This is a DataNucleus extension.** Specified on the **class**.

```java
@PersistenceAware
public class MyClass {
    ...
}
```

See the documentation for [Class Mapping](#).

149.1.4 @Embeddable

This annotation is used when you want to mark a class as persistent and only storable embedded in another object. Specified on the **class**.

```java
@Embeddable
public class MyClass {
    ...
}
```

149.1.5 @Cacheable

This annotation is used when you want to mark a class so that instance of that class can be cached. Specified on the **class**.

```java
@Cacheable
public class MyClass {
    ...
}
```

See the documentation for [L2 Cache](#).

149.1.6 @Inheritance

This annotation is used to define the inheritance persistence for this class. Specified on the **class**.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
</table>

© 2015, DataNucleus • ALL RIGHTS RESERVED.
strategy | InheritanceType | Inheritance strategy
--- | --- | ---
| | | SINGLE_TABLE
| | | JOINED
| | | TABLE_PER_CLASS

```java
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class MyClass {
    ...
}
```

See the documentation for [Inheritance](#).

### 149.1.7 @Table
This annotation is used to define the table where objects of a class will be stored. Specified on the `class`.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name of the table</td>
<td></td>
</tr>
<tr>
<td>catalog</td>
<td>String</td>
<td>Name of the catalog</td>
<td></td>
</tr>
<tr>
<td>schema</td>
<td>String</td>
<td>Name of the schema</td>
<td></td>
</tr>
<tr>
<td>uniqueConstraints</td>
<td>UniqueConstraint[]</td>
<td>Any unique constraints to apply to the table</td>
<td></td>
</tr>
<tr>
<td>indexes</td>
<td>Index[]</td>
<td>Details of indexes if wanting to override provider default</td>
<td></td>
</tr>
</tbody>
</table>

```java
@Entity
@Table(name="MYTABLE", schema="PUBLIC")
public class MyClass {
    ...
}
```

### 149.1.8 @SecondaryTable
This annotation is used to define a secondary table where some fields of this class are stored in another table. Specified on the `class`.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name of the table</td>
<td></td>
</tr>
<tr>
<td>catalog</td>
<td>String</td>
<td>Name of the catalog</td>
<td></td>
</tr>
</tbody>
</table>
### schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name of the schema</td>
</tr>
<tr>
<td>pkJoinColumns</td>
<td>PrimaryKeyJoinColumns[]</td>
<td>Join columns for the PK of the secondary table back to the primary table</td>
</tr>
<tr>
<td>uniqueConstraints</td>
<td>UniqueConstraint[]</td>
<td>Any unique constraints to apply to the table</td>
</tr>
<tr>
<td>indexes</td>
<td>Index[]</td>
<td>Details of indexes if wanting to override provider default</td>
</tr>
<tr>
<td>foreignKey</td>
<td>ForeignKey</td>
<td>Foreign key details if wanting to override provider default</td>
</tr>
</tbody>
</table>

```java
@Entity
@Table(name="MYTABLE", schema="PUBLIC")
@SecondaryTable(name="MYOTHERTABLE", schema="PUBLIC", columns={@PrimaryKeyJoinColumn(name="MYCLASS_ID")})
public class MyClass
{
    ...
}
```

See the documentation for [Secondary Tables](#secondary-tables)

### 149.1.9 @IdClass

This annotation is used to define a primary-key class for the identity of this class. Specified on the class.

```java
@Entity
@IdClass(org.datanucleus.samples.MyIdentity.class)
public class MyClass
{
    ...
}
```

See the documentation for [Primary Keys](#primary-keys)

### 149.1.10 @DatastoreIdentity

This DataNucleus-extension annotation is used to define that the class uses datastore-identity. Specified on the class.
### 149.1.11 @EntityListeners
This annotation is used to define a class or classes that are listeners for events from instances of this class. Specified on the `class`.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Class[]</td>
<td>Entity listener class(es)</td>
<td></td>
</tr>
</tbody>
</table>

```java
@Entity
@EntityListeners(org.datanucleus.MyListener.class)
public class MyClass
{
  ...
}
```

See the documentation for [Lifecycle Callbacks](#).

### 149.1.12 @NamedQueries
This annotation is used to define a series of named (JPQL) queries that can be used in this persistence unit. Specified on the `class`.

```java
@Entity
@NamedQueries...
public class MyClass
{
  ...
}
```
### 149.1.13 `@NamedQuery`

This annotation is used to define a named (JPQL) query that can be used in this persistence unit. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Symbolic name for the query. The query will be referred to under this name</td>
<td></td>
</tr>
<tr>
<td>query</td>
<td>String</td>
<td>The JPQL query</td>
<td></td>
</tr>
</tbody>
</table>

```java
@Entity
@NamedQueries({
    @NamedQuery(name="AllPeople", query="SELECT p FROM Person p"),
    @NamedQuery(name="PeopleCalledJones", query="SELECT p FROM Person p WHERE p.surname = 'Jones'"))
public class Person {
    ...
}
```

Note that with DataNucleus you can also specify `@NamedQueries` on non-persistable classes
See the documentation for Named Queries

### 149.1.14 `@NamedNativeQueries`

This annotation is used to define a series of named native (SQL) queries that can be used in this persistence unit. Specified on the class.

```java
@Entity
@NamedNativeQueries({
    @NamedNativeQuery(name="AllPeople", query="SELECT p FROM Person p")
public class Person {
    ...
}
```

Note that with DataNucleus you can also specify `@NamedNativeQuery` on non-persistable classes
See the documentation for Named Queries

---

© 2015, DataNucleus • ALL RIGHTS RESERVED.
### Attribute

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>NamedNativeQuery[]</td>
<td>The named native queries</td>
<td></td>
</tr>
</tbody>
</table>

```java
@Entity
@NamedNativeQueries({
    @NamedNativeQuery(name="AllPeople",
        query="SELECT * FROM PERSON WHERE SURNAME = 'Smith'")
    @NamedNativeQuery(name="PeopleCalledJones",
        query="SELECT * FROM PERSON WHERE SURNAME = 'Jones'"))

public class Person
{
    ...
}
```

*Note that with DataNucleus you can also specify @NamedNativeQueries on non-persistable classes*

See the documentation for  [Named Queries](https://www.data-nucleus.org/docs/)

### 149.1.15 @NamedNativeQuery

This annotation is used to define a named (SQL) query that can be used in this persistence unit. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Symbolic name for the query. The query will be referred to under this name</td>
<td></td>
</tr>
<tr>
<td>query</td>
<td>String</td>
<td>The SQL query</td>
<td></td>
</tr>
<tr>
<td>resultClass</td>
<td>Class</td>
<td>Class into which the result rows will be placed</td>
<td>void.class</td>
</tr>
</tbody>
</table>

```java
@Entity
@NamedNativeQuery(name="PeopleCalledSmith", query="SELECT * FROM PERSON WHERE SURNAME = 'Smith'")

public class Person
{
    ...
}
```

*Note that with DataNucleus you can also specify @NamedNativeQuery on non-persistable classes*

See the documentation for  [Named Queries](https://www.data-nucleus.org/docs/)
149.1.16 @NamedStoredProcedureQueries

This annotation is used to define a series of named native stored procedure queries that can be used in this persistence unit. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>NamedStoredProcedureQuery[]</td>
<td>The named stored procedure queries</td>
<td></td>
</tr>
</tbody>
</table>

```java
@Entity
@NamedStoredProcedureQueries({
    @NamedStoredProcedureQuery(name="MyProc", procedureName="MY_PROC_SP1",
    parameters={@StoredProcedureParameter(name="PARAM1", mode=ParameterMode.IN, type=String.class)}),
    @NamedStoredProcedureQuery(name="MyProc2", procedureName="MY_PROC_SP2",
    parameters={@StoredProcedureParameter(name="PARAM1", mode=ParameterMode.IN, type=Long.class)})
})
public class Person {
    ...
}
```

Note that with DataNucleus you can also specify @NamedStoredProcedureQueries on non-persistable classes

See the documentation for Named Stored procedures

149.1.17 @NamedStoredProcedureQuery

This annotation is used to define a named stored procedure query that can be used in this persistence unit. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Symbolic name for the query. The query will be referred to under this name</td>
<td></td>
</tr>
<tr>
<td>procedureName</td>
<td>String</td>
<td>Name of the stored procedure in the datastore</td>
<td></td>
</tr>
<tr>
<td>parameters</td>
<td>StoredProcedureParameter</td>
<td>Any parameter definitions for this stored procedure</td>
<td></td>
</tr>
<tr>
<td>resultClasses</td>
<td>Class[]</td>
<td>Any result class(es) for this stored procedure (one per result set)</td>
<td></td>
</tr>
<tr>
<td>resultSetMappings</td>
<td>Class[]</td>
<td>Any result set mapping(s) for this stored procedure (one per result set)</td>
<td></td>
</tr>
<tr>
<td>hints</td>
<td>QueryHint[]</td>
<td>Any query hints for this stored procedure</td>
<td></td>
</tr>
</tbody>
</table>
Note that with DataNucleus you can also specify `@NamedStoredProcedureQuery` on non-persistable classes
See the documentation for [Named StoredProcedures](#)

### 149.1.18 `@SqlResultSetMappings`

This annotation is used to define a series of result mappings for SQL queries that can be used in this persistence unit. Specified on the `class`.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>SqlResultSetMapping[]</td>
<td>The SQL result mappings</td>
<td></td>
</tr>
</tbody>
</table>

```java
@Entity
@SqlResultSetMappings({
    @SqlResultSetMapping(name="PEOPLE_PLUS_AGE",
        entities={@EntityResult(entityClass=Person.class)}, columns={@ColumnResult(name="AGE")}),
    @SqlResultSetMapping(name="FIRST_LAST_NAMES",
        columns={@ColumnResult(name="FIRSTNAME"), @ColumnResult(name="LASTNAME")})
})
public class Person {
    ...
}
```

### 149.1.19 `@SqlResultSetMapping`

This annotation is used to define a mapping for the results of an SQL query and can be used in this persistence unit. Specified on the `class`.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Symbolic name for the mapping. The mapping will be referenced under this name</td>
<td></td>
</tr>
<tr>
<td>entities</td>
<td>EntityResult[]</td>
<td>Set of entities extracted from the SQL query</td>
<td></td>
</tr>
</tbody>
</table>
### columns

<table>
<thead>
<tr>
<th>columns</th>
<th>ColumnResult[]</th>
<th>Set of columns extracted directly from the SQL query</th>
</tr>
</thead>
</table>

```java
@Entity
@SqlResultSetMapping(name="PEOPLE_PLUS_AGE",
    entities=@EntityResult(entityClass=Person.class), columns={@ColumnResult(name="AGE")})
public class Person
{
    ...
}
```

### 149.1.20 @NamedEntityGraphs

This annotation is used to define a series of named EntityGraphs that can be used in this persistence unit. Specified on the `class`.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>NamedEntityGraph[]</td>
<td>The named EntityGraphs</td>
<td></td>
</tr>
</tbody>
</table>

```java
@Entity
@NamedEntityGraph({
    @NamedEntityGraph(name="PERSON_FULL",
        attributeNodes={@NamedAttributeNode(name="friends"), @NamedAttributeNode(name="parents")}),
    @NamedEntityGraph(name="PERSON_BASIC",
        attributeNodes={@NamedAttributeNode(name="parents")})
})
public class Person
{
    ...
}
```

### 149.1.21 @NamedEntityGraph

This annotation is used to define a named EntityGraph and can be used in this persistence unit. Specified on the `class`.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>name for the Entity Graph.</td>
<td></td>
</tr>
<tr>
<td>attributeNodes</td>
<td>AttributeNode[]</td>
<td>Set of nodes in this EntityGraph</td>
<td></td>
</tr>
</tbody>
</table>
@Entity
@NamedEntityGraph(name="PERSON_FULL",
        attributeNodes={@NamedAttributeNode(name="friends"), @NamedAttributeNode(name="parents")})
public class Person
{
    ...
}

149.1.22 @PrePersist
This annotation is used to define a method that is a callback for pre-persist events. Specified on the method. It has no attributes.

@Entity
public class MyClass
{
    ...
    @PrePersist
    void registerObject()
    {
        ...
    }
}

See the documentation for Lifecycle Callbacks

149.1.23 @PostPersist
This annotation is used to define a method that is a callback for post-persist events. Specified on the method. It has no attributes.

@Entity
public class MyClass
{
    ...
    @PostPersist
    void doSomething()
    {
        ...
    }
}

See the documentation for Lifecycle Callbacks
149.1.24 @PreRemove
This annotation is used to define a method that is a callback for pre-remove events. Specified on the method. It has no attributes.

```java
@Entity
public class MyClass {
   ...
   @PreRemove
   void registerObject()
   {
      ...
   }
}
```

See the documentation for Lifecycle Callbacks

149.1.25 @PostRemove
This annotation is used to define a method that is a callback for post-remove events. Specified on the method. It has no attributes.

```java
@Entity
public class MyClass {
   ...
   @PostRemove
   void doSomething()
   {
      ...
   }
}
```

See the documentation for Lifecycle Callbacks

149.1.26 @PreUpdate
This annotation is used to define a method that is a callback for pre-update events. Specified on the method. It has no attributes.
@Entity
public class MyClass
{
    ...
    @PreUpdate
    void registerObject()
    {
        ...
    }
}

See the documentation for Lifecycle Callbacks

149.1.27 @PostUpdate
This annotation is used to define a method that is a callback for post-update events. Specified on the method. It has no attributes.

@Entity
public class MyClass
{
    ...
    @PostUpdate
    void doSomething()
    {
        ...
    }
}

See the documentation for Lifecycle Callbacks

149.1.28 @PostLoad
This annotation is used to define a method that is a callback for post-load events. Specified on the method. It has no attributes.
See the documentation for Lifecycle Callbacks

**149.1.29 @SequenceGenerator**

This annotation is used to define a generator using sequences in the datastore. It is scoped to the persistence unit. Specified on the class/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name for the generator (required)</td>
<td></td>
</tr>
<tr>
<td>sequenceName</td>
<td>String</td>
<td>Name of the underlying sequence that will be used</td>
<td></td>
</tr>
<tr>
<td>initialValue</td>
<td>int</td>
<td>Initial value for the sequence (optional)</td>
<td>1</td>
</tr>
<tr>
<td>allocationSize</td>
<td>int</td>
<td>Number of values to be allocated each time (optional)</td>
<td>50</td>
</tr>
</tbody>
</table>

```
@Entity
@SequenceGenerator(name="MySeq", sequenceName="SEQ_2")
public class MyClass
{
    ...
}
```

**149.1.30 @TableGenerator**

This annotation is used to define a generator using a table in the datastore for storing the values. It is scoped to the persistence unit. Specified on the class/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
</table>

```
@Entity
@TableGenerator
public class MyClass
{
    ...
}
```
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name for the generator (required)</td>
<td></td>
</tr>
<tr>
<td>table</td>
<td>String</td>
<td>Name of the table to use</td>
<td>SEQUENCE_TABLE</td>
</tr>
<tr>
<td>catalog</td>
<td>String</td>
<td>Catalog of the table to use</td>
<td></td>
</tr>
<tr>
<td>schema</td>
<td>String</td>
<td>Schema of the table to use</td>
<td></td>
</tr>
<tr>
<td>pkColumnName</td>
<td>String</td>
<td>Name of the primary key column for the table</td>
<td>SEQUENCE_NAME</td>
</tr>
<tr>
<td>valueColumnName</td>
<td>String</td>
<td>Name of the value column for the table</td>
<td>NEXT_VAL</td>
</tr>
<tr>
<td>pkColumnValue</td>
<td>String</td>
<td>Value to store in the PK column for the row used by this generator</td>
<td>(name of the class)</td>
</tr>
<tr>
<td>initialValue</td>
<td>int</td>
<td>Initial value for the table row (optional)</td>
<td>0</td>
</tr>
<tr>
<td>allocationSize</td>
<td>int</td>
<td>Number of values to be allocated each time (optional)</td>
<td>50</td>
</tr>
<tr>
<td>indexes</td>
<td>Index[]</td>
<td>Index(es) if wanting to override the provider default</td>
<td></td>
</tr>
</tbody>
</table>

```java
@Entity
@TableGenerator(name="MySeq", table="MYAPP_IDENTITIES", pkColumnValue="MyClass")
public class MyClass
{
    ...
}
```

### 149.1.31 @DiscriminatorColumn

This annotation is used to define the discriminator column for a class. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name of the discriminator column</td>
<td>DTYPE</td>
</tr>
<tr>
<td>discriminatorType</td>
<td>DiscriminatorType</td>
<td>Type of the discriminator column</td>
<td>STRING</td>
</tr>
<tr>
<td>length</td>
<td>String</td>
<td>Length of the discriminator column</td>
<td>31</td>
</tr>
</tbody>
</table>
149 Annotations

@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="OBJECT_TYPE", discriminatorType=DiscriminatorType.STRING)
public class MyClass
{
  ...
}

See the documentation for Inheritance

149.1.32 @DiscriminatorValue

This annotation is used to define the value to be stored in the discriminator column for a class (when used). Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>String</td>
<td>Value for the discriminator column</td>
<td></td>
</tr>
</tbody>
</table>

@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="OBJECT_TYPE", discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("MyClass")
public class MyClass
{
  ...
}

See the documentation for Inheritance

149.1.33 @PrimaryKeyJoinColumn

This annotation is used to define the names of the primary key columns when this class has a superclass. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>PrimaryKeyJoinColumn[]</td>
<td>Array of column definitions for the primary key</td>
<td></td>
</tr>
<tr>
<td>foreignKey</td>
<td>ForeignKey</td>
<td>Foreign key details if wanting to override provider default</td>
<td></td>
</tr>
</tbody>
</table>
149.1.34 @PrimaryKeyJoinColumn

This annotation is used to define the name of the primary key column when this class has a superclass. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name of the column</td>
<td></td>
</tr>
<tr>
<td>referencedColumnName</td>
<td>String</td>
<td>Name of the associated PK column in the superclass</td>
<td></td>
</tr>
<tr>
<td>columnDefinition</td>
<td>String</td>
<td>DDL to use for the column (everything except the column name). This must include the SQL type of the column</td>
<td></td>
</tr>
<tr>
<td>foreignKey</td>
<td>ForeignKey</td>
<td>Foreign key details if wanting to override provider default</td>
<td></td>
</tr>
</tbody>
</table>

149.1.35 @AttributeOverride

This annotation is used to define a field of a superclass that has its column overridden. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name of the field</td>
<td></td>
</tr>
</tbody>
</table>
149.1.36 **@AttributeOverrides**

This annotation is used to define fields of a superclass that have their columns overridden. Specified on the `class`.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>value</code></td>
<td><code>AttributeOverride[]</code></td>
<td>The overrides</td>
<td></td>
</tr>
</tbody>
</table>

```java
@Entity
@AttributeOverride(name="attr", column=@Column(name="NEW_NAME"))
public class MyClass extends MySuperClass
{
    ... 
}
```

149.1.37 **@AssociationOverride**

This annotation is used to define a 1-1/N-1 field of a superclass that has its column overridden. Specified on the `class`.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>name</code></td>
<td><code>String</code></td>
<td>Name of the field</td>
<td></td>
</tr>
<tr>
<td><code>joinColumn</code></td>
<td><code>JoinColumn</code></td>
<td>Column information for the FK column</td>
<td></td>
</tr>
</tbody>
</table>

```java
@Entity
@AssociationOverride(name="friend", joinColumn=@JoinColumn(name="FRIEND_ID"))
public class Employee extends Person
{
    ... 
}
```
149.1.38 @AssociationOverrides

This annotation is used to define 1-1/N-1 fields of a superclass that have their columns overridden. Specified on the class.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>AssociationOverride[]</td>
<td>The overrides</td>
<td></td>
</tr>
</tbody>
</table>

```java
@Entity
@AssociationOverrides({
    @AssociationOverride(name="friend", joinColumn=@JoinColumn(name="FRIEND_ID")),
    @AssociationOverride(name="teacher", joinColumn=@JoinColumn(name="TEACHER_ID"))
})
public class Employee extends Person {
    ...
}
```

149.1.39 @Id

This annotation is used to define a field to use for the identity of the class. Specified on the field/method.

```java
@Entity
public class MyClass {
    @Id
    long id;
    ...
}
```

149.1.40 @Embedded

This annotation is used to define a field as being embedded. Specified on the field/method.

```java
@Entity
public class MyClass {
    @Embedded
    Object myField;
    ...
}
```
149.1.41 @EmbeddedId
This annotation is used to define a field to use for the identity of the class when embedded. Specified on the field/method.

```java
@Entity
public class MyClass {
    @EmbeddedId
    MyPrimaryKey pk;
    ...
}
```

149.1.42 @Version
This annotation is used to define a field as holding the version for the class. Specified on the field/method.

```java
@Entity
public class MyClass {
    @Id
    long id;
    @Version
    int ver;
    ...
}
```

149.1.43 @Basic
This annotation is used to define a field of the class as persistent. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch</td>
<td>FetchType</td>
<td>Type of fetching for this field</td>
<td>LAZY</td>
</tr>
<tr>
<td>optional</td>
<td>boolean</td>
<td>Whether this field having a value is optional (can it have nulls)</td>
<td>true</td>
</tr>
</tbody>
</table>
public class Person {
  @Id
  long id;

  @Basic(optional=false)
  String forename;
  ...
}

See the documentation for Fields/Properties

@Transient
This annotation is used to define a field of the class as not persistent. Specified on the field/method.

public class Person {
  @Id
  long id;

  @Transient
  String personalInformation;
  ...
}

See the documentation for Fields/Properties

@JoinTable
This annotation is used to define that a collection/map is stored using a join table. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name of the table</td>
<td></td>
</tr>
<tr>
<td>catalog</td>
<td>String</td>
<td>Name of the catalog</td>
<td></td>
</tr>
<tr>
<td>schema</td>
<td>String</td>
<td>Name of the schema</td>
<td></td>
</tr>
<tr>
<td>joinColumns</td>
<td>JoinColumn[]</td>
<td>Columns back to the owning object (with the collection/map)</td>
<td></td>
</tr>
<tr>
<td>inverseJoinColumns</td>
<td>JoinColumn[]</td>
<td>Columns to the element object (stored in the collection/map)</td>
<td></td>
</tr>
</tbody>
</table>
149.1.46 @CollectionTable

This annotation is used to define that a collection/map of non-entities is stored using a join table. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name of the table</td>
<td></td>
</tr>
<tr>
<td>catalog</td>
<td>String</td>
<td>Name of the catalog</td>
<td></td>
</tr>
<tr>
<td>schema</td>
<td>String</td>
<td>Name of the schema</td>
<td></td>
</tr>
<tr>
<td>joinColumns</td>
<td>JoinColumn[]</td>
<td>Columns back to the owning object (with the collection/map)</td>
<td></td>
</tr>
<tr>
<td>uniqueConstraints</td>
<td>UniqueConstraint[]</td>
<td>Any unique constraints to apply to the table</td>
<td></td>
</tr>
<tr>
<td>indexes</td>
<td>Index[]</td>
<td>Details of indexes if wanting to override provider default</td>
<td></td>
</tr>
<tr>
<td>foreignKey</td>
<td>ForeignKey</td>
<td>Details of foreign key if wanting to override provider default</td>
<td></td>
</tr>
<tr>
<td>inverseForeignKey</td>
<td>ForeignKey</td>
<td>Details of foreign key if wanting to override provider default</td>
<td></td>
</tr>
</tbody>
</table>

```java
@Entity
public class Person {
    @OneToMany
    @JoinTable(name="PEOPLES_FRIENDS")
    Collection friends;
    ...
}
```
149 Annotations

@Entity
public class Person
{
    @ElementCollection
    @CollectionTable(name="PEOPLES_FRIENDS")
    Collection<String> values;
    ...
}

149.1.47 @Lob
This annotation is used to define that a field will be stored using a large object in the datastore. Specified on the field/method.

@Entity
public class Person
{
    @Lob
    byte[] photo;
    ...
}

149.1.48 @Temporal
This annotation is used to define that a field is stored as a temporal type. It specifies the JDBC type to use for storage of this type, so whether it stores the date, the time, or both. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>TemporalType</td>
<td>Type for storage</td>
<td>DATE</td>
</tr>
</tbody>
</table>

@Entity
public class Person
{
    @Temporal(TemporalType.TIMESTAMP)
    java.util.Date dateOfBirth;
    ...
}
149.1.49 @Enumerated

This annotation is used to define that a field is stored enumerated (not that it wasn't obvious from the type!). Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>EnumType</td>
<td>Type for storage</td>
<td>ORDINAL</td>
</tr>
</tbody>
</table>

```java
enum Gender {MALE, FEMALE};

@Entity
public class Person {
    @Enumerated
    Gender gender;
    ...
}
```

149.1.50 @OneToOne

This annotation is used to define that a field represents a 1-1 relation. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>targetEntity</td>
<td>Class</td>
<td>Class at the other side of the relation</td>
<td></td>
</tr>
<tr>
<td>fetch</td>
<td>FetchType</td>
<td>Whether the field should be fetched immediately</td>
<td>EAGER</td>
</tr>
<tr>
<td>optional</td>
<td>boolean</td>
<td>Whether the field can store nulls.</td>
<td>true</td>
</tr>
<tr>
<td>mappedBy</td>
<td>String</td>
<td>Name of the field that owns the relation (specified on the inverse side)</td>
<td></td>
</tr>
<tr>
<td>cascade</td>
<td>CascadeType[]</td>
<td>Whether persist, update, delete, refresh operations are cascaded</td>
<td></td>
</tr>
<tr>
<td>orphanRemoval</td>
<td>boolean</td>
<td>Whether to remove orphans when either removing this side of the relation or when nulling the relation</td>
<td>true</td>
</tr>
</tbody>
</table>
@Entity
public class Person
{
    @OneToOne
    Person bestFriend;
    ...
}

See the documentation for 1-1 Relations

149.1.51 @OneToMany

This annotation is used to define that a field represents a 1-N relation. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>targetEntity</td>
<td>Class</td>
<td>Class at the other side of the relation</td>
<td></td>
</tr>
<tr>
<td>fetch</td>
<td>FetchType</td>
<td>Whether the field should be fetched immediately</td>
<td>EAGER</td>
</tr>
<tr>
<td>mappedBy</td>
<td>String</td>
<td>Name of the field that owns the relation (specified on the inverse side)</td>
<td></td>
</tr>
<tr>
<td>cascade</td>
<td>CascadeType[]</td>
<td>Whether persist, update, delete, refresh operations are cascaded</td>
<td></td>
</tr>
<tr>
<td>orphanRemoval</td>
<td>boolean</td>
<td>Whether to remove orphans when either removing this side of the relation or when nulling the relation removing an element</td>
<td>true</td>
</tr>
</tbody>
</table>

@Entity
public class Person
{
    @OneToMany
    Collection<Person> friends;
    ...
}

See the documentation for 1-N Relations

149.1.52 @ManyToMany

This annotation is used to define that a field represents a M-N relation. Specified on the field/method.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>(targetEntity)</td>
<td>Class</td>
<td>Class at the other side of the relation</td>
<td></td>
</tr>
<tr>
<td>fetch</td>
<td>FetchType</td>
<td>Whether the field should be fetched immediately</td>
<td>EAGER</td>
</tr>
<tr>
<td>mappedBy</td>
<td>String</td>
<td>Name of the field on the non-owning side that completes the relation. Specified on the owner side.</td>
<td></td>
</tr>
<tr>
<td>cascade</td>
<td>CascadeType[]</td>
<td>Whether persist, update, delete, refresh operations are cascaded</td>
<td></td>
</tr>
</tbody>
</table>

@Entity
public class Customer
{
    @ManyToMany(mappedBy="customers")
    Collection<Supplier> suppliers;
    ...
}

@Entity
public class Supplier
{
    @ManyToMany
    Collection<Customer> customers;
    ...
}

See the documentation for M-N Relations

149.1.53 @ManyToOne
This annotation is used to define that a field represents a N-1 relation. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>(targetEntity)</td>
<td>Class</td>
<td>Class at the other side of the relation</td>
<td></td>
</tr>
<tr>
<td>fetch</td>
<td>FetchType</td>
<td>Whether the field should be fetched immediately</td>
<td>EAGER</td>
</tr>
<tr>
<td>optional</td>
<td>boolean</td>
<td>Whether the field can store nulls.</td>
<td>true</td>
</tr>
<tr>
<td>cascade</td>
<td>CascadeType[]</td>
<td>Whether persist, update, delete, refresh operations are cascaded</td>
<td></td>
</tr>
</tbody>
</table>
@Entity
public class House
{
    @OneToMany(mappedBy="house")
    Collection<Window> windows;
    ...  
}

@Entity
public class Window
{
    @ManyToOne
    House house;
    ...  
}

See the documentation for N-1 Relations

149.1.54 @ElementCollection
This annotation is used to define that a field represents a 1-N relation to non-entity objects. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>targetClass</td>
<td>Class</td>
<td>Class at the other side of the relation</td>
<td></td>
</tr>
<tr>
<td>fetch</td>
<td>FetchType</td>
<td>Whether the field should be fetched immediately</td>
<td>EAGER</td>
</tr>
</tbody>
</table>

@ElementCollection
public class Person
{
    @ElementCollection
    Collection<String> values;
    ...  
}

149.1.55 @Index (field/method - extension)
This DataNucleus-extension annotation is used to define an index for this field/property. Specified on the field/property.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name of the index</td>
<td></td>
</tr>
</tbody>
</table>
unique | boolean | Whether the index is unique | false
--- | --- | --- | ---

```java
@Entity
public class MyClass {
    @Index(name="ENABLED_IDX")
    boolean enabled;
    ...
}
```

### 149.1.56 @JdbcType
This DataNucleus-extension annotation is used to define the jdbc-type to use for this field/property. Specified on the field/property.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>String</td>
<td>JDBC Type (VARCHAR, INTEGER, BLOB, etc)</td>
<td></td>
</tr>
</tbody>
</table>

```java
@Entity
public class MyClass {
    @JdbcType("CHAR")
    boolean enabled;
    ...
}
```

### 149.1.57 @ColumnPosition
This DataNucleus-extension annotation is used to define the column position to use for this field/property. Specified on the field/property.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Integer</td>
<td>position of the column (first is &quot;0&quot;, increasing)</td>
<td></td>
</tr>
</tbody>
</table>
@Entity
public class MyClass
{
    @ColumnPosition(0)
    boolean enabled;
    ...
}

149.1.58 @GeneratedValue
This annotation is used to define the generation of a value for a (PK) field. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>strategy</td>
<td>GenerationType</td>
<td>Strategy to use when generating the values for this field. Has possible values of GenerationType TABLE, SEQUENCE, IDENTITY, AUTO.</td>
<td>GenerationType.AUTO</td>
</tr>
<tr>
<td>generator</td>
<td>String</td>
<td>Name of the generator to use. See @TableGenerator and @SequenceGenerator</td>
<td></td>
</tr>
</tbody>
</table>

@Entity
public class Person
{
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE)
    long id;
    ...
}

149.1.59 @MapKey
This annotation is used to define the field in the value class that represents the key in a Map. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name of the field in the value class to use for the key. If no value is supplied and the field is a Map then it is assumed that the key will be the primary key of the value class. DataNucleus only supports this null value treatment if the primary key of the value has a single field.</td>
<td></td>
</tr>
</tbody>
</table>

```java
@Entity
public class Person {
    @OneToMany
    @MapKey(name="nickname")
    Map<String, Person> friends;
    ...
}
```

**149.1.60 @MapKeyTemporal**

This annotation is used to define the datastore type used for the key of a map when it is a temporal type. Specified on the field/method.

```java
@Entity
public class Person {
    @ElementCollection
    @MapKeyTemporal(TemporalType.DATE)
    Map<Date, String> dateMap;
    ...
}
```

**149.1.61 @MapKeyEnumerated**

This annotation is used to define the datastore type used for the key of a map when it is an enum. Specified on the field/method.
```java
@Entity
public class Person {
    @ElementCollection
    @MapKeyEnumerated(EnumType.STRING)
    Map<MyEnum, String> dateMap;
    ...}
```

### 149.1.62 @OrderBy

This annotation is used to define a field in the element class that is used for ordering the elements of the List when it is retrieved. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>String</td>
<td>Name of the field(s) in the element class to use for ordering the elements of the List when retrieving them from the datastore. This is used by JPA &quot;ordered lists&quot; as opposed to JDO &quot;indexed lists&quot; (which always return the elements in the same order as they were persisted. The value will be a comma separated list of fields and optionally have ASC/DESC to signify ascending or descending</td>
<td></td>
</tr>
</tbody>
</table>

```java
@Entity
public class Person {
    @OneToMany
    @OrderBy(value="nickname")
    List<Person> friends;
    ...}
```

### 149.1.63 @OrderColumn

This annotation is used to define that the JPA implementation will handle the ordering of the List elements using a surrogate column. Specified on the field/method.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name of the column to use.</td>
<td>{fieldName}_ORDER</td>
</tr>
<tr>
<td>nullable</td>
<td>boolean</td>
<td>Whether the column is nullable</td>
<td>true</td>
</tr>
<tr>
<td>insertable</td>
<td>boolean</td>
<td>Whether the column is insertable</td>
<td>true</td>
</tr>
<tr>
<td>updatable</td>
<td>boolean</td>
<td>Whether the column is updatable</td>
<td>true</td>
</tr>
<tr>
<td>base</td>
<td>int</td>
<td>Base for ordering (not currently supported)</td>
<td>0</td>
</tr>
</tbody>
</table>

```java
@Entity
public class Person {
    @OneToMany
    @OrderColumn
    List<Person> friends;
    ...
}
```

### 149.1.64 @Convert

This annotation is used to define a converter for the field/property. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>converter</td>
<td>Class</td>
<td>Converter class</td>
<td></td>
</tr>
<tr>
<td>attributeName</td>
<td>String</td>
<td>Name of the embedded field to be converted (NOT YET SUPPORTED)</td>
<td></td>
</tr>
<tr>
<td>disableConversion</td>
<td>boolean</td>
<td>Whether we should disable any use of @Converter set to auto-apply</td>
<td></td>
</tr>
</tbody>
</table>

```java
@Entity
public class Person {
    @Basic
    @Convert(converter=MyURLConverter.class)
    URL website;
    ...
}
```
### 149.1.65 @Converter

This annotation is used to mark a class as being an attribute converter. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoApply</td>
<td>boolean</td>
<td>Whether this converter should always be used when storing this java type</td>
<td>false</td>
</tr>
</tbody>
</table>

```java
@Converter
public class MyConverter {
    ...
}
```

### 149.1.66 @Column

This annotation is used to define the column where a field is stored. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name for the column</td>
<td>true</td>
</tr>
<tr>
<td>unique</td>
<td>boolean</td>
<td>Whether the field is unique</td>
<td>true</td>
</tr>
<tr>
<td>nullable</td>
<td>boolean</td>
<td>Whether the field is nullable</td>
<td>true</td>
</tr>
<tr>
<td>insertable</td>
<td>boolean</td>
<td>Whether the field is insertable</td>
<td>true</td>
</tr>
<tr>
<td>updatable</td>
<td>boolean</td>
<td>Whether the field is updatable</td>
<td>true</td>
</tr>
<tr>
<td>table</td>
<td>String</td>
<td>Name of the table</td>
<td></td>
</tr>
<tr>
<td>length</td>
<td>int</td>
<td>Length for the column</td>
<td>255</td>
</tr>
<tr>
<td>precision</td>
<td>int</td>
<td>Decimal precision for the column</td>
<td>0</td>
</tr>
<tr>
<td>scale</td>
<td>int</td>
<td>Decimal scale for the column</td>
<td>0</td>
</tr>
<tr>
<td>columnDefinition</td>
<td>String</td>
<td>DDL to use for the column (everything except the column name). This must include the SQL type of the column</td>
<td></td>
</tr>
</tbody>
</table>
@Entity
public class Person {
    @Basic
    @Column(name="SURNAME", length=100, nullable=false)
    String surname;
    ...
}

149.1.67 @JoinColumn
This annotation is used to define the FK column for joining to another table. This is part of a 1-1, 1-N, or N-1 relation. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name for the column</td>
<td></td>
</tr>
<tr>
<td>referencedColumnName</td>
<td>String</td>
<td>Name of the column in the other table that this is the FK for</td>
<td></td>
</tr>
<tr>
<td>unique</td>
<td>boolean</td>
<td>Whether the field is unique</td>
<td>true</td>
</tr>
<tr>
<td>nullable</td>
<td>boolean</td>
<td>Whether the field is nullable</td>
<td>true</td>
</tr>
<tr>
<td>insertable</td>
<td>boolean</td>
<td>Whether the field is insertable</td>
<td>true</td>
</tr>
<tr>
<td>updatable</td>
<td>boolean</td>
<td>Whether the field is updatable</td>
<td>true</td>
</tr>
<tr>
<td>columnDefinition</td>
<td>String</td>
<td>DDL to use for the column (everything except the column name). This must include the SQL type of the column</td>
<td></td>
</tr>
<tr>
<td>foreignKey</td>
<td>ForeignKey</td>
<td>Foreign key details if wanting to override provider default</td>
<td></td>
</tr>
</tbody>
</table>

@Entity
public class Person {
    @OneToOne
    @JoinColumn(name="PET_ID", nullable=true)
    Animal pet;
    ...
}
149.1.68 **@JoinColumn**

This annotation is used to define the FK columns for joining to another table. This is part of a 1-1, 1-N, or N-1 relation. Specified on the field/method.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>JoinColumn[]</td>
<td>Details of the columns</td>
<td></td>
</tr>
<tr>
<td>foreignKey</td>
<td>ForeignKey</td>
<td>Foreign key details if wanting to override provider default</td>
<td></td>
</tr>
</tbody>
</table>

```java
@Entity
public class Person
{
    @OneToOne
    @JoinColumn(name="PET1_ID")
    @JoinColumn(name="PET2_ID")
    Animal pet; // composite PK
    ...
}
```

149.1.69 **@UniqueConstraint**

This annotation is used to define a unique constraint to apply to a table. It is specified as part of @Table, @JoinTable or @SecondaryTable.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>columnNames</td>
<td>String[]</td>
<td>Names of the column(s)</td>
<td></td>
</tr>
</tbody>
</table>

```java
@Entity
@Table(name="PERSON", uniqueConstraints={@UniqueConstraint(columnNames={"firstName","lastName"})})
public class Person
{
    @Basic
    String firstName;

    @Basic
    String lastName;
    ...
}
```

See the documentation for **Schema Constraints**
149.1.70 @Index
This annotation is used to define the details for an Index. It is specified as part of @Table, @JoinTable, @CollectionTable or @SecondaryTable.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name of the index</td>
<td></td>
</tr>
<tr>
<td>columnList</td>
<td>String</td>
<td>Columns to be included in this index of the form</td>
<td></td>
</tr>
<tr>
<td>unique</td>
<td>boolean</td>
<td>Whether the index is unique</td>
<td>false</td>
</tr>
</tbody>
</table>

See the documentation for Schema Constraints

149.1.71 @ForeignKey
This annotation is used to define the details for a ForeignKey. It is specified as part of @JoinColumn, @JoinTable, @CollectionTable or @SecondaryTable.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name of the foreign key</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>ConstraintMode</td>
<td>Constraint mode</td>
<td>ConstraintMode.CONSTRAINT</td>
</tr>
<tr>
<td>foreignKeyDefinition</td>
<td>String</td>
<td>DDL for the FOREIGN KEY statement of the form FOREIGN</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>KEY ( colExpr1 [, colExpr2]... ) REFERENCES</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>tblIdentifier [ ( otherColExpr1 [, otherColExpr2]... ) ]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ ON UPDATE updateAction ] [ ON DELETE deleteAction ]</td>
<td></td>
</tr>
</tbody>
</table>

See the documentation for Schema Constraints

149.1.72 @Extensions
DataNucleus Extension Annotation used to define a set of extensions specific to DataNucleus. Specified on the class or field.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Extension[]</td>
<td>Array of extensions - see @Extension annotation</td>
<td></td>
</tr>
</tbody>
</table>
@Entity
@Extensions([@Extension(key="firstExtension", value="myValue"),
               @Extension(key="secondExtension", value="myValue")])
public class Person
{
    ...
}

149.1.73 @Extension

DataNucleus Extension Annotation used to define an extension specific to DataNucleus. Specified on the class or field.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>vendorName</td>
<td>String</td>
<td>Name of the vendor</td>
<td>datanucleus</td>
</tr>
<tr>
<td>key</td>
<td>String</td>
<td>Key for the extension</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>String</td>
<td>Value of the extension</td>
<td></td>
</tr>
</tbody>
</table>

@Entity
@Extension(key="RunFast", value="true")
public class Person
{
    ...
}
150 Schema Mapping

150.1 JPA : Schema Mapping

You saw in our basic class mapping guide how you define a classes basic persistence, notating which fields are persisted. The next step is to define how it maps to the schema of the datastore (in this case RDBMS). The simplest way of mapping is to map each class to its own table. This is the default model in JDO persistence (with the exception of inheritance). If you don't specify the table and column names, then DataNucleus will generate table and column names for you. You should specify your table and column names if you have an existing schema. Failure to do so will mean that DataNucleus uses its own names and these will almost certainly not match what you have in the datastore.

150.1.1 Tables and Column names

The main thing that developers want to do when they set up the persistence of their data is to control the names of the tables and columns used for storing the classes and fields. This is an essential step when mapping to an existing schema, because it is necessary to map the classes onto the existing database entities. Let's take an example

```java
public class Hotel {
    private String name;
    private String address;
    private String telephoneNumber;
    private int numberOfRooms;
    ...
}
```

In our case we want to map this class to a table called ESTABLISHMENT, and has columns NAME, DIRECTION, PHONE and NUMBER_OF_ROOMS (amongst other things). So we define our MetaData like this

```xml
<entity class="Hotel">
    <table name="ESTABLISHMENT"/>
    <attributes>
        <basic name="name">
            <column name="NAME"/>
        </basic>
        <basic name="address">
            <column name="DIRECTION"/>
        </basic>
        <basic name="telephoneNumber">
            <column name="PHONE"/>
        </basic>
        <basic name="numberOfRooms">
            <column name="NUMBER_OF_ROOMS"/>
        </basic>
    </attributes>
</entity>
```
So we have defined the table and the column names. It should be mentioned that if you don't specify the table and column names then DataNucleus will generate names for the datastore identifiers consistent with the JPA specification. The table name will be based on the class name, and the column names will be based on the field names and the role of the field (if part of a relationship).

See also :-

- Identifier Guide - defining the identifiers to use for table/column names
- MetaData reference for <column> element

150.1.2 Column nullability and default values

So we've seen how to specify the basic structure of a table, naming the table and its columns, and how to control the types of the columns. We can extend this further to control whether the columns are allowed to contain nulls. Let's take a related class for our hotel. Here we have a class to model the payments made to the hotel.

```java
public class Payment
{
    Customer customer;
    String bankTransferReference;
    String currency;
    double amount;
}
```

In this class we can model payments from a customer of an amount. Where the customer pays by bank transfer we can save the reference number. Since the bank transfer reference is optional we want that column to be nullable. So let's specify the MetaData for the class.

```xml
<entity class="Payment">
    <attributes>
        <one-to-one name="customer">
            <primary-key-join-column name="CUSTOMER_ID"/>
        </one-to-one>
        <basic name="bankTransferReference">
            <column name="TRANSFER_REF" nullable="true"/>
        </basic>
        <basic name="currency">
            <column name="CURRENCY" default-value="GBP"/>
        </basic>
        <basic name="amount">
            <column name="AMOUNT"/>
        </basic>
    </attributes>
</entity>
```

So we make use of the `nullable` attribute. The table, when created by DataNucleus, will then provide the nullability that we require. Unfortunately with JPA there is no way to specify a default value for a field when it hasnt been set (unlike JDO where you can do that).

See also :-

- MetaData reference for <column> element
150.1.3 Column types

DataNucleus will provide a default type for any columns that it creates, but it will allow users to override this default. The default that DataNucleus chooses is always based on the Java type for the field being mapped. For example a Java field of type "int" will be mapped to a column type of INTEGER in RDBMS datastores. Similarly String will be mapped to VARCHAR. JPA does NOT allow detailed control over the JDBC type as such, with the exception of distinguishing BLOB/CLOB/TIME/TIMESTAMP types. Fortunately DataNucleus (from v3.0.2) provides an extension to overcome this flaw in the JPA spec. Here we make use of a DataNucleus annotation @JdbcType

```java
public class Payment
{
   @JdbcType("CHAR")
   String currency;
   ...
}
```

So we defined the JDBC type that this field will use (rather than the default of VARCHAR).

JPA does allow permit control over the length/precision/scale of columns. So we define this as follows

```xml
<entity name="Payment">
   <attributes>
      <one-to-one name="customer">
         <primary-key-join-column name="CUSTOMER_ID"/>
      </one-to-one>
      <basic name="bankTransferReference">
         <column name="TRANSFER_REF" nullable="true" length="255"/>
      </basic>
      <basic name="currency">
         <column name="CURRENCY" default-value="GBP" length="3"/>
      </basic>
      <basic name="amount">
         <column name="AMOUNT" precision="10" scale="2"/>
      </basic>
   </attributes>
</entity>
```

So we have defined TRANSFER_REF to use VARCHAR(255) column type, CURRENCY to use (VAR)CHAR(3) column type, and AMOUNT to use DECIMAL(10,2) column type.

See also :-

- Types Guide - defining mapping of Java types
- RDBMS Types Guide - defining mapping of Java types to JDBC/SQL types
- MetaData reference for <column> element
151 Multitenancy

151.1 JPA : Multitenancy

On occasion you need to share a data model with other user-groups or other applications and where the model is persisted to the same structure of datastore. There are three ways of handling this with DataNucleus.

- **Separate Database per Tenant** - have a different database per user-group/application.
- **Separate Schema per Tenant** - as the first option, except use different schemas.
- **Same Database/Schema but with a Discriminator** - this is described below.

151.1.1 Multitenancy via Discriminator

If you specify the persistence property `datanucleus.tenantId` as an identifier for your user-group/application then DataNucleus will know that it needs to provide a tenancy discriminator to all primary tables of persisted classes. This discriminator is then used to separate the data of the different user-groups.

By default this will add a column **TENANT_ID** to each primary table, of String-based type. You can control this by specifying extension metadata for each persistable class.

```xml
<class name="MyClass">
  <extension vendor-name="datanucleus" key="multitenancy-column-name" value="TENANT"/>
  <extension vendor-name="datanucleus" key="multitenancy-column-length" value="24"/>
  ...
</class>
```

In all subsequent use of DataNucleus, any "insert" to the primary "table"(s) will also include the TENANT column value. Additionally any query will apply a WHERE clause restricting to a particular value of TENANT column.

If you want to disable multitenancy on a class, just specify the following metadata.

```xml
<class name="MyClass">
  <extension vendor-name="datanucleus" key="multitenancy-disable" value="true"/>
  ...
</class>
```
152 Datastore Identifiers

152.1 JPA : Datastore Identifiers

A datastore identifier is a simple name of a database object, such as a column, table, index, or view, and is composed of a sequence of letters, digits, and underscores ( _ ) that represents it’s name. DataNucleus allows users to specify the names of tables, columns, indexes etc but if the user doesn't specify these DataNucleus will generate names. Generation of identifier names is controlled by an IdentifierFactory, and DataNucleus provides a default implementation. You can provide your own IdentifierFactory plugin to give your own preferred naming if so desired. You set the IdentifierFactory by setting the persistence property datanucleus.identifierFactory. Set it to the symbolic name of the factory you want to use. JPA defines what datastore identifiers should default to when not specified. DataNucleus provides a factory that meets this requirement.

- jpa IdentifierFactory (default for JPA persistence)

In describing the different possible naming conventions available out of the box with DataNucleus we’ll use the following example

```java
class MyClass
{
    String myField1;
    Collection<MyElement> elements1; // Using join table
    Collection<MyElement> elements2; // Using foreign-key
}
class MyElement
{
    String myElementField;
    MyClass myClass2;
}
```

152.1.1 IdentifierFactory 'jpa'

The IdentifierFactory "jpa" aims at providing a naming policy consistent with the JPA specification. Using the same example above, the rules in this IdentifierFactory mean that, assuming that the user doesn't specify any <column> elements :-

- MyClass will be persisted into a table named MYCLASS
- When using datastore identity MYCLASS will have a column called MYCLASS_ID
- MyClass.myField1 will be persisted into a column called MYFIELD1
- MyElement will be persisted into a table named MELEMENT
- MyClass.elements1 will be persisted into a join table called MYCLASS_MYELEMENT
- MYCLASS_ELEMENTS1 will have columns called MYCLASS_MYCLASS_ID (FK to owner table) and ELEMENTS1_ELEMENT_ID (FK to element table)
- MyClass.elements2 will be persisted into a column ELEMENTS2_MYCLASS_ID (FK to owner) table
- Any discriminator column will be called DTYPE
- Any index column in a List for field MyClass.myField1 will be called MYFIELD1_ORDER
- Any adapter column added to a join table to form part of the primary key will be called IDX
• Any version column for a table will be called VERSION

152.1.2 IdentifierFactory - Controlling the Case

The underlying datastore will define what case of identifiers are accepted. By default, DataNucleus will capitalise names (assuming that the datastore supports it). You can however influence the case used for identifiers. This is specifiable with the persistence property `datanucleus.identifier.case`, having the following values

• UpperCase: identifiers are in upper case
• LowerCase: identifiers are in lower case
• PreserveCase: No case changes are made to the name of the identifier provided by the user (class name or jdo metadata).

Please be aware that some datastores only support UPPERCASE or lowercase identifiers and so setting this parameter may have no effect if your database doesn’t support that option. Please note also that this case control only applies to DataNucleus-generated identifiers. If you provide your own identifiers for things like schema/catalog etc then you need to specify those using the case you wish to use in the datastore (including quoting as necessary)
153 Secondary Tables

153.1 JPA : Secondary Tables

The standard JPA persistence strategy is to persist an object of a class into its own table. In some situations you may wish to map the class to a primary table as well as one or more secondary tables. For example when you have a Java class that could have been split up into 2 separate classes yet, for whatever reason, has been written as a single class, however you have a legacy datastore and you need to map objects of this class into 2 tables. JPA allows persistence of fields of a class into secondary tables.

The process for managing this situation is best demonstrated with an example. Let's suppose we have a class that represents a Printer. The Printer class contains within it various attributes of the toner cartridge. So we have

```java
package com.mydomain.samples.secondarytable;

public class Printer
{
    long id;
    String make;
    String model;
    String tonerModel;
    int tonerLifetime;

    /**
     * Constructor.
     * @param make Make of printer (e.g Hewlett-Packard)
     * @param model Model of Printer (e.g LaserJet 1200L)
     * @param tonerModel Model of toner cartridge
     * @param tonerLifetime lifetime of toner (number of prints)
     */
    public Printer(String make, String model, String tonerModel, int tonerLifetime)
    {
        this.make = make;
        this.model = model;
        this.tonerModel = tonerModel;
        this.tonerLifetime = tonerLifetime;
    }
}
```

Now we have a database schema that has 2 tables (PRINTER and PRINTER_TONER) in which to store objects of this class. So we need to tell DataNucleus to perform this mapping. So we define the MetaData for the Printer class like this
So here we have defined that objects of the `Printer` class will be stored in the primary table `PRINTER`. In addition we have defined that some fields are stored in the table `PRINTER_TONER`.

- We declare the "secondary-table"(s) that we will be using at the start of the definition.
- We define `tonerModel` and `tonerLifetime` to use columns in the table `PRINTER_TONER`. This uses the "table" attribute of `<column>`
- Whilst defining the secondary table(s) we will be using, we also define the join column to be called "PRINTER_REFID".

This results in the following database tables :-

![Database Tables Diagram]

So we now have our primary and secondary database tables. The primary key of the `PRINTER_TONER` table serves as a foreign key to the primary class. Whenever we persist a `Printer` object a row will be inserted into both of these tables.

See also :-

- MetaData reference for `<secondary-table>` element
- MetaData reference for `<column>` element
- Annotations reference for `@SecondaryTable`
- Annotations reference for `@Column`
154 Constraints

154.1 JPA : Constraints
A datastore often provides ways of constraining the storage of data to maintain relationships and improve performance. These are known as constraints and they come in various forms. These are:-

- **Indexes** - these are used to mark fields that are referenced often as indexes so that when they are used the performance is optimised.
- **Unique constraints** - these are placed on fields that should have a unique value. That is, only one object will have a particular value.
- **Foreign-Keys** - these are used to interrelate objects, and allow the datastore to keep the integrity of the data in the datastore.
- **Primary-Keys** - allow the PK to be set, and also to have a name.

154.1.1 Indexes

Applicable to RDBMS, NeoDatis, MongoDB

Many datastores provide the ability to have indexes defined to give performance benefits. With RDBMS the indexes are specified on the table and the indexes to the rows are stored separately. In the same way an ODBMS typically allows indexes to be specified on the fields of the class, and these are managed by the datastore. JPA 2.1 allows you to define the indexes on a table-by-table basis by metadata as in the following example (note that you cannot specify indexes on a field basis like in JDO)

```java
import javax.persistence.Index;

@Entity
@Table(indexes={@Index(name="SOME_VAL_IDX", columnList="SOME_VALUE")})
public class MyClass
{
    @Column(name="SOME_VALUE")
    long someValue;
    ...
}
```

The JPA @Index annotation is only applicable at a class level. DataNucleus provides its own @Index annotation that you can specify on a field/method to signify that the column(s) for this field/method will be indexed. Like this
154.1.2 Unique constraints

Applicable to RDBMS, NeoDatis, MongoDB

Some datastores provide the ability to have unique constraints defined on tables to give extra control over data integrity. JPA1 provides a mechanism for defining such unique constraints. Let's take an example class, and show how to specify this

```java
public class Person
{
    String forename;
    String surname;
    String nickname;
    ...
}
```

and here we want to impose uniqueness on the "nickname" field, so there is only one Person known as "DataNucleus Guru" for example!

```
<entity class="Person">
    <table name="PEOPLE"/>
    <attributes>
        ...
        <basic name="nickname">
            <column name="SURNAME" unique="true"/>
        </basic>
        ...
    </attributes>
</entity>
```

The second use of unique constraints is where we want to impose uniqueness across composite columns. So we reuse the class above, and this time we want to impose a constraint that there is only one Person with a particular "forename+surname".
In the same way we can also impose unique constraints on `<join-table>` and `<secondary-table>`

See also :-

- MetaData reference for `<column>` element
- MetaData reference for `<unique-constraint>` element
- Annotations reference for `@Column`
- Annotations reference for `@UniqueConstraint`

### 154.1.3 Foreign Keys

**Applicable to RDBMS**

When objects have relationships with one object containing, for example, a Collection of another object, it is common to store a foreign key in the datastore representation to link the two associated tables. Moreover, it is common to define behaviour about what happens to the dependent object when the owning object is deleted. Should the deletion of the owner cause the deletion of the dependent object maybe ? JPA 2.1 adds support for defining the foreign key for relation fields as per the following example

```java
public class MyClass
{
    ...

    @OneToOne
    @JoinColumn(name="OTHER_ID", foreignKey=@ForeignKey(name="OTHER_FK",
            foreignKeyDefinition="FOREIGN KEY (OTHER_ID) REFERENCES MY_OTHER_TBL (MY_OTHER_ID) ")")
    MyOtherClass other;
}
```
Note that when you don't specify any foreign key the JPA provider is free to add the foreign keys that it considers are necessary.

154.1.4 Primary Keys

Applicable to RDBMS

In RDBMS datastores, it is accepted as good practice to have a primary key on all tables. You specify in other parts of the MetaData which fields are part of the primary key (if using application identity). Unfortunately JPA1 doesn't allow specification of the name of the primary key constraint, nor of whether join tables are given a primary key constraint at all.
155 Enhancer

155.1 DataNucleus Enhancer

As is described in the Class Enhancement guide below, DataNucleus utilises the common technique of byte-code manipulation to make your normal Java classes "persistable". The mechanism provided by DataNucleus is to use an "enhancer" process to perform this manipulation before you use your classes at runtime. The process is very quick and easy.

How to use the DataNucleus Enhancer depends on what environment you are using. Below are some typical examples.

- Post-compilation
  - Using Maven via the DataNucleus Maven plugin
  - Using Ant
  - Manual invocation at the command line
  - Using the Eclipse DataNucleus plugin
- At runtime
  - Runtime Enhancement
  - Programmatically via an API

155.1.1 Maven

Maven operates from a series of plugins. There is a DataNucleus plugin for Maven that allows enhancement of classes. Go to the Download section of the website and download this. Once you have the Maven plugin, you then need to set any properties for the plugin in your pom.xml file. Some properties that you may need to change are below

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>persistenceUnitName</td>
<td></td>
<td>Name of the persistence-unit to enhance. Mandatory</td>
</tr>
<tr>
<td>log4jConfiguration</td>
<td></td>
<td>Config file location for Log4J (if using it)</td>
</tr>
<tr>
<td>jdkLogConfiguration</td>
<td></td>
<td>Config file location for JDK1.4 logging (if using it)</td>
</tr>
<tr>
<td>api</td>
<td>JDO</td>
<td>API to enhance to (JDO, JPA). Mandatory: Set this to JPA</td>
</tr>
<tr>
<td>verbose</td>
<td>false</td>
<td>Verbose output?</td>
</tr>
<tr>
<td>quiet</td>
<td>false</td>
<td>No output?</td>
</tr>
<tr>
<td>targetDirectory</td>
<td></td>
<td>Where the enhanced classes are written (default is to overwrite them)</td>
</tr>
<tr>
<td>fork</td>
<td>true</td>
<td>Whether to fork the enhancer process (e.g if you get a command line too long with Windows).</td>
</tr>
</tbody>
</table>

©2015, DataNucleus • ALL RIGHTS RESERVED.
generatePK  true  Generate a PK class (of name 
{MyClass}_PK) for cases where there are multiple PK fields yet no PK class is defined.
generateConstructor  true  Generate a default constructor if not defined for the class being enhanced.
detachListener  false  Whether to enhance classes to make use of a detach listener for attempts to access an undetached field.

You will need to add (javax.jdo) jdo-api and (org.datanucleus) datanucleus-api-jpa into the CLASSPATH (of the plugin, or your project) for the enhancer to operate. Similarly persistence-api (but then you almost certainly will have that in your project CLASSPATH anyway).

You then run the Maven DataNucleus plugin, as follows

```
mvn datanucleus:enhance
```

This will enhance all classes for the specified persistence-unit. If you want to check the current status of enhancement you can also type

```
mvn datanucleus:enhance-check
```

Or alternatively, you could add the following to your POM

```
<build>
  ...
  <plugins>
    <plugin>
      <groupId>org.datanucleus</groupId>
      <artifactId>datanucleus-maven-plugin</artifactId>
      <version>3.3.0-release</version>
      <configuration>
        <api>JPA</api>
        <persistenceUnitName>MyUnit</persistenceUnitName>
        <log4jConfiguration>${basedir}/log4j.properties</log4jConfiguration>
        <verbose>true</verbose>
      </configuration>
      <executions>
        <execution>
          <phase>process-classes</phase>
          <goals>
            <goal>enhance</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
  ...
</build>
```
So you then get auto-enhancement after each compile. Please refer to the Maven JPA guide for more details.

155.1.2 Ant

Ant provides a powerful framework for performing tasks. DataNucleus provides an Ant task to enhance classes. DataNucleus provides an Enhancer in datanucleus-core.jar. You need to make sure that the datanucleus-core.jar, datanucleus-api-jpa.jar, log4j.jar (optional), persistence-api.jar and jdo-api.jar are in your CLASSPATH. In the DataNucleus Enhancer Ant task, the following parameters are available:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>destination</td>
<td>Optional. Defining a directory where enhanced classes will be written. If omitted, the original classes are updated.</td>
<td></td>
</tr>
<tr>
<td>api</td>
<td>Defines the API to be used when enhancing</td>
<td>Set this to JPA</td>
</tr>
<tr>
<td>persistenceUnit</td>
<td>Defines the &quot;persistence-unit&quot; to enhance.</td>
<td></td>
</tr>
<tr>
<td>checkonly</td>
<td>Whether to just check the classes for enhancement status. Will respond for each class with &quot;ENHANCED&quot; or &quot;NOT ENHANCED&quot;. This will disable the enhancement process and just perform these checks.</td>
<td>true, false</td>
</tr>
<tr>
<td>verbose</td>
<td>Whether to have verbose output.</td>
<td>true, false</td>
</tr>
<tr>
<td>quiet</td>
<td>Whether to have no output.</td>
<td>true, false</td>
</tr>
<tr>
<td>generatePK</td>
<td>Whether to generate PK classes as required.</td>
<td>true, false</td>
</tr>
<tr>
<td>generateConstructor</td>
<td>Whether to generate a default constructor as required.</td>
<td>true, false</td>
</tr>
<tr>
<td>if</td>
<td>Optional. The name of a property that must be set in order to the Enhancer Ant Task to execute.</td>
<td></td>
</tr>
</tbody>
</table>

The enhancer task extends the Apache Ant Java task, thus all parameters available to the Java task are also available to the enhancer task.

So you could define something like the following, setting up the parameter enhancer.classpath, and log4j.config.file to suit your situation.
155.1.3 Manually

DataNucleus provides an Enhancer in datanucleus-core.jar. If you are building your application manually and want to enhance your classes you follow the instructions in this section. You invoke the enhancer as follows

```java
java -cp classpath org.datanucleus.enhancer.DataNucleusEnhancer [options]
```

where options can be

- `-pu (persistence-unit-name)`: Name of a "persistence-unit" to enhance the classes for
- `-d (target-dir-name)`: Write the enhanced classes to the specified directory
- `-api (api-name)`: Name of the API we are enhancing for (JDO, JPA). Set this to JPA
- `-checkonly`: Just check the classes for enhancement status
- `-v`: verbose output
- `-q`: quiet mode (no output, overrides verbose flag too)
- `-generatePK {flag}`: generate any PK classes where needed
  `{{flag} should be true or false - default=true)
- `-generateConstructor {flag}`: generate default constructor where needed
  `{{flag} should be true or false - default=true)

where "mapping-files" and "class-files" are provided when not enhancing a persistence-unit, and give the paths to the mapping files and class-files that define the classes being enhanced.

where classpath must contain the following
- datanucleus-core.jar
- datanucleus-api-jpa.jar
- persistence-api.jar
- jdo-api.jar
- log4j.jar (optional)
- your classes
- your meta-data files

The input to the enhancer should be the name of the "persistence-unit" to enhance. To give an example of how you would invoke the enhancer
Linux/Unix:

```
java -cp target/classes:lib/datanucleus-core.jar:lib/jdo-api.jar:
   lib/datanucleus-api-jpa.jar:lib/persistence-api.jar:lib/log4j.jar
   -Dlog4j.configuration=file:log4j.properties
org.datanucleus.enhancer.DataNucleusEnhancer
-api JPA -pu MyUnit
```

Windows:

```
java -cp target\classes;lib\datanucleus-core.jar;lib\jdo-api.jar;
   lib\datanucleus-api-jpa.jar;lib\persistence-api.jar;lib\log4j.jar
   -Dlog4j.configuration=file:log4j.properties
org.datanucleus.enhancer.DataNucleusEnhancer
-api JPA -pu MyUnit
```

[should all be on same line. Shown like this for clarity]

So you pass in the persistence-unit name as the final argument(s) in the list, and include the respective JAR’s in the classpath (-cp). The enhancer responds as follows

```
DataNucleus Enhancer (version 3.2.0.m2) for API "JPA" using JRE "1.6"

DataNucleus Enhancer : Classpath
>> /home/andy/work/myproject/target/classes
>> /home/andy/work/myproject/lib/log4j.jar
>> /home/andy/work/myproject/lib/jdo-api.jar
>> /home/andy/work/myproject/lib/persistence-api.jar
>> /home/andy/work/myproject/lib/datanucleus-core.jar
>> /home/andy/work/myproject/lib/datanucleus-api-jpa.jar

ENHANCED (PersistenceCapable): org.mydomain.mypackage1.Pack
ENHANCED (PersistenceCapable): org.mydomain.mypackage1.Card
DataNucleus Enhancer completed with success for 2 classes. Timings : input=422 ms, enhance=490 ms, total=912 ms.
... Consult the log for full details
```

If you have errors here relating to "Log4J" then you must fix these first. If you receive no output about which class was ENHANCED then you should look in the DataNucleus enhancer log for errors. The enhancer performs much error checking on the validity of the passed MetaData and the majority of errors are caught at this point. You can also use the DataNucleus Enhancer to check whether classes are enhanced. To invoke the enhancer in this mode you specify the **checkonly** flag. This will return a list of the classes, stating whether each class is enhanced for persistence under JDO or not. The classes need to be in the CLASSPATH (Please note that a CLASSPATH should contain a set of JAR’s, and a set of directories. It should NOT explicitly include class files, and should NOT include parts of the package names. If in doubt please consult a Java book).

### 155.1.4 Runtime Enhancement

Enhancement of persistent classes at runtime is possible when using JRE 1.5 or superior versions. Runtime Enhancement requires the `datanucleus-core` jar be in the CLASSPATH but then you’d have that if using DataNucleus.
When operating in a JavaEE environment (JBoss, WebSphere, etc) instead set the persistence property `datanucleus.jpa.addClassTransformer` to `true`. Note that this is only for a real JavaEE server that implements the JavaEE parts of the JPA spec.

To enable runtime enhancement in other environments, the `javaagent` option must be set in the java command line. For example,

```
java -javaagent:datanucleus-core.jar=-api=JPA Main
```

The statement above will mean that all classes, when being loaded, will be processed by the ClassFileTransformer (except class in packages "java.*", "javax.*", "org.datanucleus.*"). This means that it can be slow since the MetaData search algorithm will be utilised for each. To speed this up you can specify an argument to that command specifying the names of package(s) that should be processed (and all others will be ignored). Like this

```
java -javaagent:datanucleus-core.jar=-api=JPA,mydomain.mypackage1,mydomain.mypackage2 Main
```

so in this case only classes being loaded that are in `mydomain.mypackage1` and `mydomain.mypackage2` will be attempted to be enhanced.

Please take care over the following when using runtime enhancement

- When you have a class with a field of another persistable type make sure that you mark that field as "persistent" (@Persistent, or in XML) since with runtime enhancement at that point the related class is likely not yet enhanced so will likely not be marked as persistent otherwise. **Be explicit**
- If the agent jar is not found make sure it is specified with an absolute path.

### 155.1.5 Programmatic API

You could alternatively programmatically enhance classes from within your application. This is done as follows.

```java
import org.datanucleus.enhancer.DataNucleusEnhancer;

DataNucleusEnhancer enhancer = new DataNucleusEnhancer("JPA", null);
enhancer.setVerbose(true);
enhancer.addPersistenceUnit("MyPersistenceUnit");
enhancer.enhance();
```

This will look in META-INF/persistence.xml and enhance all classes defined by that unit. **Please note that you will need to load the enhanced version of the class into a different ClassLoader after performing this operation to use them.** See this guide

### 155.2 Class enhancement

DataNucleus requires that all classes that are persisted implement `PersistenceCapable`, an interface defined by JDO. **Why should we do this, Hibernate/TopLink dont need it ?**. Well that's a simple question really

- DataNucleus uses this `PersistenceCapable` interface, and adds it using bytecode enhancement techniques so that you never need to actually change your classes. This means that you get **transparent persistence**, and your classes always remain your classes. ORM tools that use a mix of reflection and/or proxies are not totally transparent.
• DataNucleus' use of PersistenceCapable provides transparent change tracking. When any change is made to an object the change creates a notification to DataNucleus allowing it to be optimally persisted. ORM tools that don't have access to such change tracking have to use reflection to detect changes. The performance of this process will break down as soon as you read a large number of objects, but modify just a handful, with these tools having to compare all object states for modification at transaction commit time.

Why not also read this comparison of bytecode enhancement, and proxies. It gives a clear enough comparison of the relative benefits.

Note that when using JPA, DataNucleus also requires bytecode enhancement. We make use of the very same JDO bytecode enhancement contract since it is a defined standard, so easier to adopt that for JPA than adding our own custom enhancement that does the same as JDO.

In the JDO bytecode enhancement contract there are 3 categories of classes. These are PersistenceCapable, PersistenceAware and normal classes. The Meta-Data defines which classes fit into these categories. To give an example, we have 3 classes. The class A is to be persisted in the datastore. The class B directly updates the fields of class A but doesn't need persisting. The class C is not involved in the persistence process. We would define these classes as follows

```java
@Entity
public class A {
    String myField;
    ...
}

@org.datanucleus.api.jpa.annotations.PersistenceAware
public class B {
    ...
}
```

So our MetaData is mainly for those classes that are PersistenceCapable and are to be persisted to the datastore (we don't really need the persistence-modifier for these classes since this is the default). For PersistenceAware classes we simply notate that the class knows about persistence. We don't define MetaData for any class that has no knowledge of persistence.

The JDO contract requires that all classes to be persisted must implement the PersistenceCapable interface

```javadoc```, and for JPA they also implement the Detachable interface

```javadoc```

Users could manually do this themselves but this would impose work on them. DataNucleus provides a byte-code enhancer that converts the users normal classes to implement this interface (this can be found in the datanucleus-core.jar). The DataNucleus enhancer fully implements JDO bytecode enhancement contract. The example below doesn't show all PersistenceCapable and Detachable methods, but demonstrates that all added methods and fields are prefixed with "jdo" to distinguish them from the users own methods and fields. Also each persistent field of the class will be given a jdoGetXXX, jdoSetXXX method so that accesses of these fields are intercepted so that DataNucleus can manage their "dirty" state. The MetaData defines which classes are required to be persisted, and also defines which aspects of persistence each class requires.
The main thing to know is that the detached state (object id of the datastore object, the version of the datastore object when it was detached, and which fields were detached is stored in "jdoDetachedState"). Please see the JDO spec for more details.

**If the MetaData is changed in any way during development, the classes should always be recompiled and re-enhanced afterwards.**

### 155.2.1 Byte-Code Enhancement Myths

Some groups (e.g. Hibernate) perpetuated arguments against "byte-code enhancement" saying that it was somehow 'evil'. The most common were:-

- *Slows down the code-test cycle*. This is erroneous since you only need to enhance just before test and the provided plugins for Ant, Eclipse and Maven all do the enhancement job automatically and rapidly.

- *Is less "lazy" than the proxy approach since you have to load the object as soon as you get a pointer to it*. In a 1-1 relation you have to load the object then since you would cause issues with null pointers otherwise. With 1-N relations you load the elements of the collection/map only when you access them and not the collection/map. Hardly an issue then is it!

- *Fail to detect changes to public fields unless you enhance your client code*. Firstly very few people will be writing code with public fields since it is bad practice in an OO design, and secondly, this is why we have "PersistenceAware" classes.

So as you can see, there are no valid reasons against byte-code enhancement, and the pluses are that runtime detection of dirty events on objects is much quicker, hence your persistence layer operates faster without any need for iterative reflection-based checks. The fact is that Hibernate itself also now has a mode whereby you can do bytecode enhancement although not the default mode of Hibernate. So maybe it wasn't so evil after all?
155.2.2 Decompilation

Many people will wonder what actually happens to a class upon bytecode enhancement. In simple terms the necessary methods and fields are added so as to implement PersistenceCapable. If you want to check this, just use a Java decompiler such as JD. It has a nice GUI allowing you to just select your class to decompile and shows you the source.
156 Datastore Schema

156.1 JPA : Datastore Schema

Some datastores have a well-defined structure and when persisting/retrieving from these datastores you have to have this *schema* in place. DataNucleus provides various controls for creation of any necessary schema components. This creation can be performed as follows

- At runtime, **as a one-off generate-schema step**. This is the recommended option since it is standard in JPA2.1
- One off task before running your application using *SchemaTool*
- At runtime, **auto-generating tables as it requires them**

The thing to remember when using DataNucleus is that **the schema is under your control**. DataNucleus does not impose anything on you as such, and you have the power to turn on/off all schema components. Some Java persistence tools add various types of information to the tables for persisted classes, such as special columns, or meta information. DataNucleus is very unobtrusive as far as the datastore schema is concerned. It minimises the addition of any implementation artifacts to the datastore, and adds *nothing* (other than any datastore identities, and version columns where requested) to any schema tables.

### 156.1.1 Schema Generation for persistence-unit

DataNucleus allows you to generate the schema for your *persistence-unit* when creating a EMF. You enable this by specifying the persistence property `javax.persistence.schema-generation-action` to create (other values drop which deletes the schema, and drop-and-create which deletes and recreates the schema). When you create your EMF it will generate the schema before it returns your EMF.

You have some additional control over whether to actually create the schema, or whether to just output DDL for the schema (which you could apply yourself) - see the persistence property `javax.persistence.schema-generation-target` which can be set to *scripts* to just create the DDL.

For RDBMS there are a few extensions here that are also worthy of mention, in that you can define scripts that are run during this schema generation phase. These are controlled by the following persistence properties

- `javax.persistence.schema-generation.create-script-source` - set this to an SQL script of your own that will create some tables (prior to any schema generation from the persistable objects)
- `javax.persistence.schema-generation.drop-script-source` - set this to an SQL script of your own that will drop some tables (prior to any schema generation from the persistable objects)
- `javax.persistence.sql.load-script-source` - set this to an SQL script of your own that will insert any data that you require to be available when your PMF is initialised

### 156.1.2 Schema Auto-Generation at runtime

*Extension*

If you want to create the schema ("tables"+"columns"+"constraints") during the persistence process, the property `datanucleus.autoCreateSchema` provides a way of telling DataNucleus to do this. It's a shortcut to setting the other 3 properties to true. Thereafter, during calls to DataNucleus to persist classes or performs queries of persisted data, whenever it encounters a new class to persist that it has no information about, it will use the MetaData to check the datastore for presence of the "table", and
if it doesn’t exist, will create it. In addition it will validate the correctness of the table (compared to the MetaData for the class), and any other constraints that it requires (to manage any relationships). If any constraints are missing it will create them.

• If you wanted to only create the “tables” required, and none of the “constraints” the property `datanucleus.autoCreateTables` provides this, simply performing the tables part of the above.
• If you want to create any missing "columns" that are required, the property `datanucleus.autoCreateColumns` provides this, validating and adding any missing columns.
• If you wanted to only create the "constraints" required, and none of the "tables" the property `datanucleus.autoCreateConstraints` provides this, simply performing the "constraints" part of the above.
• If you want to keep your schema fixed (i.e don't allow any modifications at runtime) then the property `datanucleus.fixedDatastore` should be set to `true` and this has the effect of setting the "table"+"columns"+"constraints" autoCreate values to `false`

156.1.3 Schema Generation : Validation

DataNucleus can check any existing schema against what is implied by the MetaData.

The property `datanucleus.validateTables` provides a way of telling DataNucleus to validate any tables that it needs against their current definition in the datastore. If the user already has a schema, and want to make sure that their tables match what DataNucleus requires (from the MetaData definition) they would set this property to `true`. This can be useful for example where you are trying to map to an existing schema and want to verify that you’ve got the correct MetaData definition.

The property `datanucleus.validateColumns` provides a way of telling DataNucleus to validate any columns of the tables that it needs against their current definition in the datastore. If the user already has a schema, and want to make sure that their tables match what DataNucleus requires (from the MetaData definition) they would set this property to `true`. This will validate the precise column types and widths etc, including defaultability/nullability settings. **Please be aware that many JDBC drivers contain bugs that return incorrect column detail information and so having this turned off is sometimes the only option (dependent on the JDBC driver quality).**

The property `datanucleus.validateConstraints` provides a way of telling DataNucleus to validate any constraints (primary keys, foreign keys, indexes) that it needs against their current definition in the datastore. If the user already has a schema, and want to make sure that their table constraints match what DataNucleus requires (from the MetaData definition) they would set this property to `true`.

156.1.4 Schema Generation : Naming Issues

Some datastores allow access to multiple "schemas" (such as with most RDBMS). DataNucleus will, by default, use the "default" database schema for the Connection URL and user supplied. This may cause issues where the user has been set up and in some databases (e.g Oracle) you want to write to a different schema (which that user has access to). To achieve this in DataNucleus you would set the persistence properties

```
datanucleus.mapping.Catalog={the_catalog_name}
datanucleus.mapping.Schema={the_schema_name}
```

This will mean that all RDBMS DDL and SQL statements will prefix table names with the necessary catalog and schema names (specify which ones your datastore supports).
156.1.5 Schema Generation: Column Ordering

By default all tables are generated with columns in alphabetical order, starting with root class fields followed by subclass fields (if present in the same table) etc. This is not part of JPA but DataNucleus allows an extension to specify the relative position, such as

```java
@ColumnPosition(3)
```

Note that the values of the position start at 0, and should be specified completely for all columns of all fields.

156.1.6 Schema: Read-Only

If your datastore is read-only (you can't add/update/delete any data in it), obviously you could just configure your application to not perform these operations. An alternative is to set the EMF as read-only, by setting the persistence property `datanucleus.ReadOnlyDatastore` to `true`.

From now on, whenever you perform a persistence operation that implies a change in datastore data, the operation will throw a `PersistenceException`.

DataNucleus provides an additional control over the behaviour when an attempt is made to change a read-only datastore. The default behaviour is to throw an exception. You can change this using the persistence property `datanucleus.readOnlyDatastoreAction` with values of "EXCEPTION" (default), and "IGNORE". "IGNORE" has the effect of simply ignoring all attempted updates to read-only objects.

You can take this read-only control further and specify it just on specific classes. Like this

```java
@Extension(vendorName="datanucleus", key="read-only", value="true")
public class MyClass {...}
```

156.2 SchemaTool

DataNucleusSchemaTool currently works with RDBMS, HBase, Excel, OOXML, ODF, MongoDB datastores and is very simple to operate. It has the following modes of operation:

- **create** - create all database tables required for the classes defined by the input data.
- **delete** - delete all database tables required for the classes defined by the input data.
- **deletecreate** - delete all database tables required for the classes defined by the input data, then create the tables.
- **validate** - validate all database tables required for the classes defined by the input data.
- **dbinfo** - provide detailed information about the database, it's limits and datatypes support. Only for RDBMS currently.
- **schemainfo** - provide detailed information about the database schema. Only for RDBMS currently.

In addition for RDBMS, the **create/delete** modes can be used by adding "-ddlFile {filename}" and this will then not create/delete the schema, but instead output the DDL for the tables/constraints into the specified file.
For the create, delete and validate modes DataNucleus SchemaTool accepts either of the following types of input:

- A set of MetaData and class files. The MetaData files define the persistence of the classes they contain. The class files are provided when the classes have annotations.
- The name of a persistence-unit. The persistence-unit name defines all classes, metadata files, and jars that make up that unit. Consequently, running DataNucleus SchemaTool with a persistence unit name will create the schema for all classes that are part of that unit.

Here we provide many different ways to invoke DataNucleus SchemaTool:

- Invoke it using Maven, with the DataNucleus Maven plugin
- Invoke it using Ant, using the provided DataNucleus SchemaTool Ant task
- Invoke it manually from the command line
- Invoke it using the DataNucleus Eclipse plugin
- Invoke it programmatically from within an application

### 156.2.1 Maven

If you are using Maven to build your system, you will need the DataNucleus Maven plugin. This provides 5 goals representing the different modes of DataNucleus SchemaTool. You can use the goals datanucleus:schema-create, datanucleus:schema-delete, datanucleus:schema-validate depending on whether you want to create, delete or validate the database tables. To use the DataNucleus Maven plugin you will may need to set properties for the plugin (in your pom.xml). For example

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>api</td>
<td>JDO</td>
<td>API for the metadata being used (JDO, JPA). Set this to JPA normally</td>
</tr>
<tr>
<td>persistenceUnitName</td>
<td></td>
<td>Name of the persistence-unit to generate the schema for (defines the classes and the properties defining the datastore). Mandatory</td>
</tr>
<tr>
<td>log4jConfiguration</td>
<td></td>
<td>Config file location for Log4J (if using it)</td>
</tr>
<tr>
<td>jdkLogConfiguration</td>
<td></td>
<td>Config file location for JDK1.4 logging (if using it)</td>
</tr>
<tr>
<td>verbose</td>
<td>false</td>
<td>Verbose output?</td>
</tr>
<tr>
<td>fork</td>
<td>true</td>
<td>Whether to fork the SchemaTool process. Note that if you don’t fork the process, DataNucleus will likely struggle to determine class names from the input filenames, so you need to use a persistence.xml file defining the class names directly.</td>
</tr>
<tr>
<td>ddlFile</td>
<td></td>
<td>Name of an output file to dump any DDL to (for RDBMS)</td>
</tr>
<tr>
<td>completeDdl</td>
<td>false</td>
<td>Whether to generate DDL including things that already exist? (for RDBMS)</td>
</tr>
</tbody>
</table>
So to give an example, I add the following to my pom.xml

```xml
<build>
  ...
  <plugins>
    <plugin>
      <groupId>org.datanucleus</groupId>
      <artifactId>datanucleus-maven-plugin</artifactId>
      <version>3.3.0-m1</version>
      <configuration>
        <api>JPA</api>
        <persistenceUnitName>MyUnit</persistenceUnitName>
        <log4jConfiguration>${basedir}/log4j.properties</log4jConfiguration>
        <verbose>true</verbose>
      </configuration>
    </plugin>
  </plugins>
  ...
</build>
```

So with these properties when I run SchemaTool it uses properties from the file `datanucleus.properties` at the root of the Maven project. I am also specifying a log4j configuration file defining the logging for the SchemaTool process. I then can invoke any of the Maven goals

<table>
<thead>
<tr>
<th>Maven Goal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mvn datanucleus:schema-create</td>
<td>Create the Schema</td>
</tr>
<tr>
<td>mvn datanucleus:schema-delete</td>
<td>Delete the schema</td>
</tr>
<tr>
<td>mvn datanucleus:schema-deletecreate</td>
<td>Delete and create the schema</td>
</tr>
<tr>
<td>mvn datanucleus:schema-validate</td>
<td>Validate the Schema</td>
</tr>
<tr>
<td>mvn datanucleus:schema-info</td>
<td>Output info for the Schema</td>
</tr>
<tr>
<td>mvn datanucleus:schema-dbinfo</td>
<td>Output info for the datastore</td>
</tr>
</tbody>
</table>

**156.2.2 Ant**

An Ant task is provided for using DataNucleus SchemaTool. It has classname `org.datanucleus.store.schema.SchemaToolTask`, and accepts the following parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>api</td>
<td>API that we are using in our use of DataNucleus. Set this to JPA typically</td>
<td>JDO</td>
</tr>
<tr>
<td>persistenceUnit</td>
<td>Name of the persistence-unit that we should manage the schema for (defines the classes and the properties defining the datastore).</td>
<td></td>
</tr>
</tbody>
</table>
The SchemaTool task extends the Apache Ant Java task, thus all parameters available to the Java task are also available to the SchemaTool task.

In addition to the parameters that the Ant task accepts, you will need to set up your CLASSPATH to include the classes and MetaData files, and to define the following system properties via the sysproperty parameter (not required when specifying the persistence props via the properties file, or when providing the persistence-unit)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>datanucleus.ConnectionDriverName</td>
<td>Name of JDBC driver class</td>
<td>Mandatory</td>
</tr>
<tr>
<td>datanucleus.ConnectionURL</td>
<td>URL for the database</td>
<td>Mandatory</td>
</tr>
<tr>
<td>datanucleus.ConnectionUserName</td>
<td>User name for the database</td>
<td>Mandatory</td>
</tr>
<tr>
<td>datanucleus.ConnectionPassword</td>
<td>Password for the database</td>
<td>Mandatory</td>
</tr>
<tr>
<td>datanucleus.Mapping</td>
<td>ORM Mapping name</td>
<td>Optional</td>
</tr>
<tr>
<td>log4j.configuration</td>
<td>Log4J configuration file, for SchemaTool's Log</td>
<td>Optional</td>
</tr>
</tbody>
</table>

So you could define something like the following, setting up the parameters schematool.classpath, datanucleus.ConnectionDriverName, datanucleus.ConnectionURL, datanucleus.ConnectionUserName, and datanucleus.ConnectionPassword to suit your situation.

You define the jdo files to create the tables using fileset.
156.2.3 Manual Usage

If you wish to call DataNucleus SchemaTool manually, it can be called as follows

```bash
java [-cp classpath] [system_props] org.datanucleus.store.schema.SchemaTool [modes] [options]
```

where system_props (when specified) should include

- `Ddatanucleus.ConnectionURL=db_url`
- `Ddatanucleus.ConnectionDriverName=db_driver_name`
- `Ddatanucleus.ConnectionUserName=db_username`
- `Ddatanucleus.ConnectionPassword=db_password`
- `Dlog4j.configuration=file:{log4j.properties}` (optional)

where modes can be

- `-create` : Create the tables specified by the mapping-files/class-files
- `-delete` : Delete the tables specified by the mapping-files/class-files
- `-deletecreate` : Delete the tables specified by the mapping-files/class-files and then create them
- `-validate` : Validate the tables specified by the mapping-files/class-files
- `-dbinfo` : Detailed information about the database
- `-schemainfo` : Detailed information about the database schema

where options can be

- `-api` : The API that is being used (default is JDO, but set this to JPA)
- `-pu {persistence-unit-name}` : Name of the persistence unit to manage the schema for
- `-ddlFile {filename}` : RDBMS - only for use with "create"/"delete" mode to dump the DDL to the specified file
- `-completeDdl` : RDBMS - when using "ddlFile" in "create" mode to get all DDL output and not just missing tables/constraints
- `-includeAutoStart` : whether to include any auto-start mechanism in SchemaTool usage
- `-v` : verbose output

All classes, MetaData files, "persistence.xml" files must be present in the CLASSPATH. In terms of the schema to use, you either specify the "props" file (recommended), or you specify the
System properties defining the database connection, or the properties in the "persistence-unit". You should only specify one of the [modes] above. Let's make a specific example and see the output from SchemaTool. So we have the following files in our application:

```
src/java/...  (source files and MetaData files)
target/classes/...  (enhanced classes, and MetaData files)
lib/log4j.jar  (optional, for Log4J logging)
lib/datanucleus-core.jar
lib/datanucleus-api-jpa.jar
lib/datanucleus-rdbms.jar, lib/datanucleus-hbase.jar, etc
lib/jdo-api.jar
lib/persistence-api.jar
lib/mysql-connector-java.jar (driver for our database)
log4j.properties
```

So we want to create the schema for our persistent classes. So let's invoke DataNucleus SchemaTool to do this, from the top level of our project. In this example we're using Linux (change the CLASSPATH definition to suit for Windows):

```
     lib/mysql-connector-java.jar
-Dlog4j.configuration=file:log4j.properties
org.datanucleus.store.schema.SchemaTool -create
-api JPA -pu MyUnit
```

DataNucleus SchemaTool (version 3.3.0.m1) : Creation of the schema

```
DataNucleus SchemaTool : Classpath
>> /home/andy/work/DataNucleus/samples/packofcards/target/classes
>> /home/andy/work/DataNucleus/samples/packofcards/lib/log4j.jar
>> /home/andy/work/DataNucleus/samples/packofcards/lib/datanucleus-core.jar
>> /home/andy/work/DataNucleus/samples/packofcards/lib/datanucleus-api-jpa.jar
>> /home/andy/work/DataNucleus/samples/packofcards/lib/datanucleus-rdbms.jar
>> /home/andy/work/DataNucleus/samples/packofcards/lib/jdo-api.jar
>> /home/andy/work/DataNucleus/samples/packofcards/lib/persistence-api.jar
>> /home/andy/work/DataNucleus/samples/packofcards/lib/mysql-connector-java.jar
```

DataNucleus SchemaTool : Persistence-Unit="MyUnit"

So as you see, DataNucleus SchemaTool prints out our input, the properties used, and finally a success message. If an error occurs, then something will be printed to the screen, and more information will be written to the log.

### 156.2.4 SchemaTool API

DataNucleus SchemaTool can also be called programmatically from an application. You need to get hold of the StoreManager and cast it to SchemaAwareStoreManager. The API is shown below.
package org.datanucleus.store.schema;

public interface SchemaAwareStoreManager {
    public int createSchema(Set<String> classNames, Properties props);
    public int deleteSchema(Set<String> classNames, Properties props);
    public int validateSchema(Set<String> classNames, Properties props);
}

So for example to create the schema for classes mydomain.A and mydomain.B you would do something like this:

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("MyUnit");
NucleusContext nucCtx = emf.unwrap(NucleusContext.class);
...
List classNames = new ArrayList();
classNames.add("mydomain.A");
classNames.add("mydomain.B");
try {
    Properties props = new Properties();
    // Set any properties for schema generation
    ((SchemaAwareStoreManager)nucCtx.getStoreManager()).createSchema(classNames, props);
} catch(Exception e) {
    ...
}
```
157 Bean Validation

157.1 JPA : Bean Validation

The Bean Validation API (JSR0303) can be hooked up with JPA so that you have validation of an objects values prior to persistence, update and deletion. To do this

- Put the `javax.validation` "validation-api" jar in your CLASSPATH, along with the Bean Validation implementation jar of your choice
- Set the persistence property `javax.persistence.validation.mode` to one of `auto` (default), `none`, or `callback`
- Optionally set the persistence property(s) `javax.persistence.validation.group.pre-persist`, `javax.persistence.validation.group.pre-update`, `javax.persistence.validation.group.pre-remove` to fine tune the behaviour (the default is to run validation on pre-persist and pre-update if you don't specify these).
- Use JPA as you normally would for persisting objects

To give a simple example of what you can do with the Bean Validation API

```java
@Entity
public class Person {
    @Id
    @NotNull
    private Long id;

    @NotNull
    @Size(min = 3, max = 80)
    private String name;

    ...
}
```

So we are validating that instances of the `Person` class will have an "id" that is not null and that the "name" field is not null and between 3 and 80 characters. If it doesn't validate then at persist/update an exception will be thrown. You can add bean validation annotations to classes marked as `@Entity`, `@MappedSuperclass` or `@Embeddable`. 
158 EntityManagerFactory

158.1 JPA : Entity Manager Factory

Any JPA-enabled application will require at least one EntityManagerFactory. Typically applications create one per datastore being utilised. An EntityManagerFactory provides access to EntityManagers which allow objects to be persisted, and retrieved. The EntityManagerFactory can be configured to provide particular behaviour.

Important: an EntityManagerFactory is designed to be thread-safe. An EntityManager is not

158.1.1 Create an EMF in JavaSE

The simplest way of creating an EntityManagerFactory in a JavaSE environment is as follows

```java
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
...
EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPU");
```

So you simply provide the name of the persistence-unit which defines the properties, classes, metadata etc to be used. An alternative is to specify the properties to use along with the persistence-unit name. In that case the passed properties will override any that are specified for the persistence unit itself.

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPU", overridingProps);
```

158.1.2 Create an EMF in JavaEE

If you want an application-managed EMF then you create it by injection like this, providing the name of the required persistence-unit

```java
@PersistenceUnit(unitName="myPU")
EntityManagerFactory emf;
```

If you want a container-managed EM then you create it by injection like this, providing the name of the required persistence-unit

```java
@PersistenceContext(unitName="myPU")
EntityManager em;
```
Please refer to the docs for your JavaEE server for more details.

### 158.2 Persistence Unit

When designing an application you can usually nicely separate your persistable objects into independent groupings that can be treated separately, perhaps within a different DAO object, if using DAOs. JPA introduces the idea of a *persistence-unit*. A *persistence-unit* provides a convenient way of specifying a set of metadata files, and classes, and jars that contain all classes to be persisted in a grouping. The persistence-unit is named, and the name is used for identifying it. Consequently this name can then be used when defining what classes are to be enhanced, for example.

To define a *persistence-unit* you first need to add a file `persistence.xml` to the `META-INF/` directory of your application jar. This file will be used to define your *persistence-units*. Let's show an example:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence_2_1.xsd" version="2.1">

    <!-- Online Store -->
    <persistence-unit name="OnlineStore">
        <provider>org.datanucleus.api.jpa.PersistenceProviderImpl</provider>
        <class>org.datanucleus.samples.metadata.store.Product</class>
        <class>org.datanucleus.samples.metadata.store.Book</class>
        <class>org.datanucleus.samples.metadata.store.CompactDisc</class>
        <class>org.datanucleus.samples.metadata.store.Customer</class>
        <class>org.datanucleus.samples.metadata.store.Supplier</class>
        <exclude-unlisted-classes/>
        <properties>
            <property name="javax.persistence.jdbc.url" value="jdbc:h2:datanucleus"/>
            <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
            <property name="javax.persistence.jdbc.user" value="sa"/>
            <property name="javax.persistence.jdbc.password" value=""/>
        </properties>
    </persistence-unit>

    <!-- Accounting -->
    <persistence-unit name="Accounting">
        <provider>org.datanucleus.api.jpa.PersistenceProviderImpl</provider>
        <mapping-file>com/datanucleus/samples/metadata/accounts/orm.xml</mapping-file>
        <properties>
            <property name="javax.persistence.jdbc.url" value="jdbc:h2:datanucleus"/>
            <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
            <property name="javax.persistence.jdbc.user" value="sa"/>
            <property name="javax.persistence.jdbc.password" value=""/>
        </properties>
    </persistence-unit>

</persistence>
```
In this example we have defined 2 persistence-units. The first has the name "OnlineStore" and contains 5 classes (annotated). The second has the name "Accounting" and contains a metadata file called "orm.xml" in a particular package (which will define the classes being part of that unit). This means that once we have defined this we can reference these persistence-units in our persistence operations. You can find the XSD for persistence.xml here.

There are several sub-elements of this persistence.xml file:

- **provider** - the JPA persistence provider to be used. The JPA persistence "provider" for DataNucleus is `org.datanucleus.api.jpa.PersistenceProviderImpl`
- **jta-data-source** - JNDI name for JTA connections
- **non-jta-data-source** - JNDI name for non-JTA connections. Note that if using a JTA datasource as the primary connection, you ought to provide a non-jta-data-source also since any schema generation and/or sequence handling will need to use that.
- **jar-file** - name of a JAR file to scan for annotated classes to include in this persistence-unit.
- **mapping-file** - name of an XML "mapping" file containing persistence information to be included in this persistence-unit.
- **class** - name of an annotated class to include in this persistence-unit
- **properties** - properties defining the persistence factory to be used. Please refer to Persistence Properties Guide for details

### 158.2.1 Specifying the datastore properties

With a persistence-unit you have 2 ways of specifying the datastore to use:

- **Specify the connection URL/driverName/userName/password** and it will internally create a DataSource for this URL (with optional connection pooling). This is achieved by specifying `javax.persistence.jdbc.url`, `javax.persistence.jdbc.driver`, `javax.persistence.jdbc.user`, and `javax.persistence.jdbc.password` properties
- **Specify the JNDI name of the connectionFactory** This is achieved by specifying `javax.persistence.jtaDataSource`, and `javax.persistence.nonJtaDataSource` (for secondary operations) or by specifying the element(s) `jta-data-source/ non-jta-data-source`

### 158.2.2 Restricting to specific classes

If you want to just have specific classes in the persistence-unit you can specify them using the class element, and then add `exclude-unlisted-classes`, like this:

```xml
<persistence-unit name="Store">
    <provider>org.datanucleus.api.jpa.PersistenceProviderImpl</provider>
    <class>org.datanucleus.samples.metadata.store.Product</class>
    <class>org.datanucleus.samples.metadata.store.Book</class>
    <class>org.datanucleus.samples.metadata.store.CompactDisc</class>
    <exclude-unlisted-classes/>

    ...
</persistence-unit>
```

If you don't include the `exclude-unlisted-classes` then DataNucleus will search for annotated classes starting at the root of the persistence-unit (the root directory in the CLASSPATH that contains the "META-INF/persistence.xml" file).
158.2.3 Dynamically generated Persistence-Unit

DataNucleus allows an extension to JPA to dynamically create persistence-units at runtime. Use the following code sample as a guide. Obviously any classes defined in the persistence-unit need to have been enhanced.

```java
import org.datanucleus.metadata.PersistenceUnitMetaData;
import org.datanucleus.api.jpa.JPAEntityManagerFactory;

PersistenceUnitMetaData pumd = new PersistenceUnitMetaData("dynamic-unit", "RESOURCE_LOCAL", null);
    pumd.setExcludeUnlistedClasses();
    pumd.addProperty("javax.persistence.jdbc.url", "jdbc:h2:mem:nucleus");
    pumd.addProperty("javax.persistence.jdbc.driver", "org.h2.Driver");
    pumd.addProperty("javax.persistence.jdbc.user", "sa");
    pumd.addProperty("javax.persistence.jdbc.password", "");
    pumd.addProperty("datanucleus.autoCreateSchema", "true");

EntityManagerFactory emf = new JPAEntityManagerFactory(pumd, null);
```

It should be noted that if you call `pumd.toString()` then this returns the text that would have been found in a `persistence.xml` file.

158.2.4 Standard JPA Properties

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.persistence.provider</td>
<td>Class name of the provider to use. DataNucleus has a provider name of <code>org.datanucleus.api.jpa.PersistenceProviderImpl</code> if you only have 1 persistence provider in the CLASSPATH then this doesn't need specifying.</td>
<td></td>
</tr>
<tr>
<td>javax.persistence.transactionType</td>
<td>RESOURCE_LOCAL</td>
<td>Type of transactions to use. In Java SE the default is RESOURCE_LOCAL. In Java EE the default is JTA. Note that if using a JTA datasource as the primary connection, you ought to provide a non-jta-data-source also since any schema generation and/or sequence handling will need to use that.</td>
</tr>
<tr>
<td>javax.persistence.jtaDataSource</td>
<td>JNDI name of a (transactional) JTA data source. Note that if using a JTA datasource as the primary connection, you ought to provide a non-jta-data-source also since any schema generation and/or sequence handling will need to use that.</td>
<td></td>
</tr>
<tr>
<td>Java Property</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>--------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>javax.persistence.nonJtaDataSource</td>
<td>JNDI name of a (non-transactional) data source.</td>
<td></td>
</tr>
<tr>
<td>javax.persistence.jdbc.url</td>
<td>Alias for datanucleus.ConnectionURL. Note that this is (also) used to define which type of datastore is being used.</td>
<td></td>
</tr>
<tr>
<td>javax.persistence.jdbc.driver</td>
<td>Alias for datanucleus.ConnectionDriverName</td>
<td></td>
</tr>
<tr>
<td>javax.persistence.jdbc.user</td>
<td>Alias for datanucleus.ConnectionUserName</td>
<td></td>
</tr>
<tr>
<td>javax.persistence.jdbc.password</td>
<td>Alias for datanucleus.ConnectionPassword</td>
<td></td>
</tr>
<tr>
<td>javax.persistence.query.timeout</td>
<td>Alias for datanucleus.query.timeout</td>
<td></td>
</tr>
<tr>
<td>javax.persistence.sharedCache.mode</td>
<td>Alias for datanucleus.cache.level2.mode</td>
<td></td>
</tr>
<tr>
<td>javax.persistence.validation.mode</td>
<td>Alias for datanucleus.validation.mode</td>
<td></td>
</tr>
<tr>
<td>javax.persistence.validation.group.prePersist</td>
<td>Alias for datanucleus.validation.group.prePersist</td>
<td></td>
</tr>
<tr>
<td>javax.persistence.validation.group.preUpdate</td>
<td>Alias for datanucleus.validation.group.preUpdate</td>
<td></td>
</tr>
<tr>
<td>javax.persistence.validation.group.preRemove</td>
<td>Alias for datanucleus.validation.group.preRemove</td>
<td></td>
</tr>
<tr>
<td>javax.persistence.validation.factory</td>
<td>Alias for datanucleus.validation.factory</td>
<td></td>
</tr>
<tr>
<td>javax.persistence.schema génération.database.action</td>
<td>create</td>
<td>drop</td>
</tr>
<tr>
<td>javax.persistence.schema génération.scripts.action</td>
<td>create</td>
<td>drop</td>
</tr>
<tr>
<td>javax.persistence.schema génération.scripts.create-target</td>
<td>{filename}</td>
<td></td>
</tr>
<tr>
<td>javax.persistence.schema génération.scripts.drop-target</td>
<td>{filename}</td>
<td></td>
</tr>
<tr>
<td>javax.persistence.schema génération.scripts.create-script-source</td>
<td>{filename}</td>
<td></td>
</tr>
<tr>
<td>javax.persistence.schema génération.scripts.drop-script-source</td>
<td>{filename}</td>
<td></td>
</tr>
<tr>
<td>javax.persistence.sql.load-script-source</td>
<td>{filename}</td>
<td></td>
</tr>
</tbody>
</table>

©2015, DataNucleus • ALL RIGHTS RESERVED.
158.2.5 Extension DataNucleus Properties

DataNucleus provides many properties to extend the control that JPA gives you. These can be used alongside the above standard JPA properties, but will only work with DataNucleus. Please consult the Persistence Properties Guide for full details. In addition we have the following properties explicitly for JPA.

**datanucleus.jpa.addClassTransformer**

<table>
<thead>
<tr>
<th>Description</th>
<th>When running with JPA in a JavaEE environment if you wish to have your classes enhanced at runtime you can enable this by setting this property to <code>true</code>. The default is to bytecode enhance your classes before deployment.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range of Values</td>
<td><code>false</code></td>
</tr>
</tbody>
</table>

**datanucleus.jpa.persistenceContextType**

<table>
<thead>
<tr>
<th>Description</th>
<th>JPA defines two lifecycle options. JavaEE usage defaults to &quot;transaction&quot; where objects are detached when a transaction is committed. JavaSE usage defaults to &quot;extended&quot; where objects are detached when the EntityManager is closed. This property allows control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range of Values</td>
<td><code>transaction</code></td>
</tr>
</tbody>
</table>

**datanucleus.jpa.txnMarkForRollbackOnException**

<table>
<thead>
<tr>
<th>Description</th>
<th>JPA requires that any persistence exception should mark the current transaction for rollback. This persistence property allows that inflexible behaviour to be turned off leaving it to the user to decide when a transaction is needing to be rolled back.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range of Values</td>
<td><code>true</code></td>
</tr>
</tbody>
</table>
159 L2 Cache

159.1 JPA : Caching

Caching is an essential mechanism in providing efficient usage of resources in many systems. Caching allows objects to be retained and returned rapidly without having to make an extra call to the datastore. JPA defines caching at 2 levels, with the second level as optional (some JPA providers don't see the need to provide this out of the box, but DataNucleus does). The 2 levels of caching available are:

- **Level 1 Cache** - represents the caching of instances within an EntityManager
- **Level 2 Cache** - represents the caching of instances within an EntityManagerFactory (across multiple EntityManager's)

You can think of a cache as a Map, with values referred to by keys. In the case of JPA, the key is the object identity (identity is unique in JPA).

159.1.1 Level 2 Cache

By default the **Level 2 Cache** is enabled. The user can configure the Level 2 (L2) Cache if they so wish; by use of the persistence property `datanucleus.cache.level2.type`. You set this to "type" of cache required. With the L2 Cache you currently have the following options.

- **none** - turn OFF Level 2 caching.
- **weak** - use the internal (weak reference based) L2 cache. Provides support for the JPA2 interface of being able to put objects into the cache, and evict them when required. This option does not support distributed caching, solely running within the JVM of the client application. Weak references are held to non pinned objects.
- **soft** - use the internal (soft reference based) L2 cache. Provides support for the JPA2 interface of being able to put objects into the cache, and evict them when required. This option does not support distributed caching, solely running within the JVM of the client application. Soft references are held to non pinned objects.
- **EHCache** - a simple wrapper to EHCache's caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.
- **EHCacheClassBased** - similar to the EHCache option but class-based.
- **OSCache** - a simple wrapper to OSCache's caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.
- **SwarmCache** - a simple wrapper to SwarmCache's caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.
- **Oracle Coherence** - a simple wrapper to Oracle's Coherence caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning. Oracle's caches support distributed caching, so you could, in principle, use DataNucleus in a distributed environment with this option.
- **javax.cache** - a simple wrapper to standard javax.cache's caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.
- **JCache** - a simple wrapper to an old version of javax.cache's caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.
spymemcached - a simple wrapper to Spymemcached java client for memcached caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.

xmemcached - a simple wrapper to Xmemcached java client for memcached caching product. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.

cacheonix - a simple wrapper to the Cacheonix distributed caching software. Provides basic support for adding items to the cache and retrieval from the cache. Doesn't support pinning and unpinning.

The javax.cache cache is available in the datanucleus-core plugin. The EHCache, OSCache, SwarmCache, Coherence, JCache, Cacheonix, and Memcache caches are available in the datanucleus-cache plugin.

In addition you can control the mode of operation of the L2 cache. You do this using the persistence property datanucleus.cache.level2.mode (or javax.persistence.sharedCache.mode). The default is UNSPECIFIED which means that DataNucleus will cache all objects of entities unless the entity is explicitly marked as not cacheable. The other options are NONE (don't cache ever), ALL (cache all entities regardless of annotations), ENABLE_SELECTIVE (cache entities explicitly marked as cacheable), or DISABLE_SELECTIVE (cache entities unless explicitly marked as not cacheable - i.e same as our default).

Objects are placed in the L2 cache when you commit() the transaction of a EntityManager. This means that you only have datastore-persisted objects in that cache. Also, if an object is deleted during a transaction then at commit it will be removed from the L2 cache if it is present.

The Level 2 cache is a DataNucleus plugin point allowing you to provide your own cache where you require it. Use the examples of the EHCache, Coherence caches etc as reference.

159.1.2 Controlling the Level 2 Cache

The majority of times when using a JPA-enabled system you will not have to take control over any aspect of the caching other than specification of whether to use a Level 2 Cache or not. With JPA and DataNucleus you have the ability to control which objects remain in the cache. This is available via a method on the EntityManagerFactory.

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory(persUnitName, props);
Cache cache = emf.getCache();
```

The Cache interface provides methods to control the retention of objects in the cache. You have 2 types of methods

- **contains** - check if an object of a type with a particular identity is in the cache
- **evict** - used to remove objects from the Level 2 Cache

You can also control which classes are put into a Level 2 cache. So with the following JPA2 annotation @Cacheable, no objects of type MyClass will be put in the L2 cache.
If you want to control which fields of an object are put in the Level 2 cache you can do this using an extension annotation on the field. This setting is only required for fields that are relationships to other persistable objects. Like this

```java
public class MyClass
{
    ...
    Collection values;
    @Extension(vendorName="datanucleus", key="cacheable", value="false")
    Collection elements;
}
```

So in this example we will cache "values" but not "elements". If a field is *cacheable* then

- If it is a persistable object, the "identity" of the related object will be stored in the Level 2 cache for this field of this object
- If it is a Collection of persistable elements, the "identity" of the elements will be stored in the Level 2 cache for this field of this object
- If it is a Map of persistable keys/values, the "identity" of the keys/values will be stored in the Level 2 cache for this field of this object

When pulling an object in from the Level 2 cache and it has a reference to another object Access Platform uses the "identity" to find that object in the Level 1 or Level 2 caches to re-relate the objects.

### 159.1.3 L2 Cache using javax.cache

DataNucleus provides a simple wrapper to *javax.cache's caches*. To enable this you should set the persistence properties

```
datanucleus.cache.level2.type=javax.cache
datanucleus.cache.level2.cacheName={cache name}
datanucleus.cache.level2.timeout={expiration time in millis - optional}
```

### 159.1.4 L2 Cache using JCache

DataNucleus provides a simple wrapper to *JCache's caches*. This is an old version of what will become javax.cache (separate option). To enable this you should set the persistence properties
159.1.5 L2 Cache using Oracle Coherence

DataNucleus provides a simple wrapper to Oracle's Coherence caches. This currently takes the NamedCache interface in Coherence and instantiates a cache of a user provided name. To enabled this you should set the following persistence properties

```java
datanucleus.cache.level2.type=coherence
datanucleus.cache.level2.cacheName={coherence cache name}
```

The *Coherence cache name* is the name that you would normally put into a call to CacheFactory.getCache(name). You have the benefits of Coherence's distributed/serialized caching. If you require more control over the Coherence cache whilst using it with DataNucleus, you can just access the cache directly via

```java
DataStoreCache cache = pmf.getDataStoreCache();
NamedCache tangosolCache = ((TangosolLevel2Cache)cache).getTangosolCache();
```

159.1.6 L2 Cache using EHCache

DataNucleus provides a simple wrapper to EHCache's caches. To enable this you should set the persistence properties

```java
datanucleus.cache.level2.type=ehcache
datanucleus.cache.level2.cacheName={cache name}
datanucleus.cache.level2.configurationFile={EHCache configuration file (in classpath)}
```

The EHCache plugin also provides an alternative L2 Cache that is class-based. To use this you would need to replace "ehcache" above with "ehcacheclassbased".

159.1.7 L2 Cache using OSCache

DataNucleus provides a simple wrapper to OSCache's caches. To enable this you should set the persistence properties

```java
datanucleus.cache.level2.type=oscache
datanucleus.cache.level2.cacheName={cache name}
```
159.1.8 L2 Cache using SwarmCache
DataNucleus provides a simple wrapper to SwarmCache's caches. To enable this you should set the persistence properties

```
datanucleus.cache.level2.type=swarmcache
datanucleus.cache.level2.cacheName={cache name}
```

159.1.9 L2 Cache using Spymemcached/Xmemcached
DataNucleus provides a simple wrapper to Spymemcached caches and Xmemcached caches. To enable this you should set the persistence properties

```
datanucleus.cache.level2.type=spymemcached         [or "xmemcached"]
datanucleus.cache.level2.cacheName={prefix for keys, to avoid clashes with other memcached objects}
datanucleus.cache.level2.memcached.servers=...
datanucleus.cache.level2.memcached.expireSeconds=...
```

`datanucleus.cache.level2.memcached.servers` is a space separated list of memcached hosts/ports, e.g. host:port host2:port. `datanucleus.cache.level2.memcached.expireSeconds` if not set or set to 0 then no expire

159.1.10 L2 Cache using Cacheonix
DataNucleus provides a simple wrapper to Cacheonix. To enable this you should set the persistence properties

```
datanucleus.cache.level2.type=cacheonix
datanucleus.cache.level2.cacheName={cache name}
```

Note that you can optionally also specify

```
datanucleus.cache.level2.timeout={timeout-in-millis (default=60)}
datanucleus.cache.level2.configurationFile={Cacheonix configuration file (in classpath)}
```

and define a `cacheonix-config.xml` like
<?xml version="1.0"?>
<cacheonix>
  <local>
    <!-- One cache per class being stored. -->
    <localCache name="mydomain.MyClass">
      <store>
        <lru maxElements="1000" maxBytes="1mb"/>
        <expiration timeToLive="60s"/>
      </store>
    </localCache>
    <!-- Fallback cache for classes indeterminable from their id. -->
    <localCache name="datanucleus">
      <store>
        <lru maxElements="1000" maxBytes="10mb"/>
        <expiration timeToLive="60s"/>
      </store>
    </localCache>
    <localCache name="default" template="true">
      <store>
        <lru maxElements="10" maxBytes="10mb"/>
        <overflowToDisk maxOverflowBytes="1mb"/>
        <expiration timeToLive="1s"/>
      </store>
    </localCache>
  </local>
</cacheonix>
160 Entity Manager

160.1 JPA : Entity Manager

As you read in the guide for EntityManagerFactory, to control the persistence of your objects you will require at least one EntityManagerFactory. Once you have obtained this object you then use this to obtain an EntityManager. An EntityManager provides access to the operations for persistence of your objects. This short guide will demonstrate some of the more common operations.

Important : An EntityManagerFactory is designed to be thread-safe. An EntityManager is not...

You obtain an EntityManager from an EntityManagerFactory as follows

```java
EntityManager em = emf.createEntityManager();
```

In the case of using container-managed JavaEE, you would instead obtain the EntityManager by injection

```java
@PersistenceContext(unitName="myPU")
EntityManager em;
```

In general you will be performing all operations on a EntityManager within a transaction, whether your transactions are controlled by your JavaEE container, by a framework such as Spring, or by locally defined transactions. In the examples below we will omit the transaction demarcation for clarity.

160.1.1 Persisting an Object

The main thing that you will want to do with the data layer of a JPA-enabled application is persist your objects into the datastore. As we mentioned earlier, a EntityManagerFactory represents the datastore where the objects will be persisted. So you create a normal Java object in your application, and you then persist this as follows

```java
em.persist(obj);
```

This will result in the object being persisted into the datastore, though clearly it will not be persistent until you commit the transaction. The LifecycleState of the object changes from Transient to PersistentClean (after persist()), to Hollow (at commit).

160.1.2 Persisting multiple Objects in one call

When you want to persist multiple objects with standard JPA you have to call persist multiple times. Fortunately DataNucleus extends this to take in a Collection or an array of entities, so you can do

```java
em.persist(coll);
```

As above, the objects are persisted to the datastore. The LifecycleState of the objects change from Transient to PersistentClean (after persist()), to Hollow (at commit).
160.1.3 Finding an object by its identity

Once you have persisted an object, it has an “identity”. This is a unique way of identifying it. When you specify the persistence for the class you specified an id class so you can create the identity from that. So what? Well the identity can be used to retrieve the object again at some other part in your application. So you pass the identity into your application, and the user clicks on some button on a web page and that button corresponds to a particular object identity. You can then go back to your data layer and retrieve the object as follows

```
Object obj = em.find(cls, id);
```

where cls is the class of the object you want to find, and id is the identity. Note that the first argument could be a base class and the real object could be an instance of a subclass of that. Note that the second argument is either the value of the single primary-key field (when it has only one primary key field), or is the value of the object-id-class (when it has multiple primary key fields).

160.1.4 Deleting an Object

When you need to delete an object that you had previous persisted, deleting it is simple. Firstly you need to get the object itself, and then delete it as follows

```
Object obj = em.find(cls, id);  // Retrieves the object to delete
em.remove(obj);
```

160.1.5 Deleting multiple Objects

When you want to delete multiple objects with standard JPA you have to call remove multiple times. Fortunately DataNucleus extends this to take in a Collection or an array of entities, so you can do

```
Collection objsToRemove = new HashSet();
objsToRemove.add(obj1);
objsToRemove.add(obj2);
em.remove(objsToRemove);
```

160.1.6 Modifying a persisted Object

To modify a previously persisted object you take the object and update it in your code. When you are ready to persist the changes you do the following

```
Object updatedObj = em.merge(obj)
```

160.1.7 Modifying multiple persisted Objects

©2015, DataNucleus • ALL RIGHTS RESERVED.
When you want to attach multiple modified objects with standard JPA you have to call *merge* multiple times. Fortunately DataNucleus extends this to take in a Collection or an array of entities, so you can do

Object updatedObj = em.merge(coll)

160.1.8 Refreshing a persisted Object

When you think that the datastore has more up-to-date values than the current values in a retrieved persisted object you can refresh the values in the object by doing the following

em.refresh(obj)

This will do the following

- Refresh all fields that are to be eagerly fetched from the datastore
- Unload all loaded fields that are to be lazily fetched.

If the object had any changes they will be thrown away by this step, and replaced by the latest datastore values.

160.1.9 Getting EntityManager for an object

JPA doesn't provide a method for getting the EntityManager of an object as such. Fortunately DataNucleus provides the following

EntityManager em = NucleusJPAHelper.getEntityManager(obj);

160.1.10 Level 1 Cache

Each EntityManager maintains a cache of the objects that it has encountered (or have been "enlisted") during its lifetime. This is termed the Level 1 Cache. It is enabled by default and you should only ever disable it if you really know what you are doing. There are inbuilt types for the Level 1 (L1) Cache available for selection. DataNucleus supports the following types of L1 Cache:

- **weak** - uses a weak reference backing map. If JVM garbage collection clears the reference, then the object is removed from the cache.
- **soft** - uses a soft reference backing map. If the map entry value object is not being actively used, then garbage collection *may* garbage collect the reference, in which case the object is removed from the cache.
- **strong** - uses a normal HashMap backing. With this option all references are strong meaning that objects stay in the cache until they are explicitly removed by calling remove() on the cache.

You can specify the type of L1 Cache by providing the persistence property `datanucleus.cache.level1.type`. You set this to the value of the type required. If you want to remove all objects from the L1 cache programmatically you should use `em.clear()` but bear in mind the other things that this will impact on.

Objects are placed in the L1 Cache (and updated there) during the course of the transaction. This provides rapid access to the objects in use in the users application and is used to guarantee that there...
is only one object with a particular identity at any one time for that EntityManager. When the EM is closed the cache is cleared.

The L1 cache is a DataNucleus plugin point allowing you to provide your own cache where you require it.
161 Object Lifecycle

161.1 JPA : Object Lifecycle

During the persistence process an object goes through lifecycle changes. Below we demonstrate the primary object lifecycle changes for JPA. With JPA these lifecycles are referred to as "persistence contexts". There are two: transaction (default for JavaEE usage) and extended (default for JavaSE usage). DataNucleus allows control over which to use by specification of the persistence property datanucleus.jpa.persistenceContextType

161.1.1 Transaction

A newly created object is transient. You then persist it and it becomes persistent. You then commit the transaction and it is detached for use elsewhere in the application, in detached state. You then attach any changes back to persistence and it becomes persistent again. Finally when you delete the object from persistence and commit that transaction it is in transient state.

161.1.2 Extended

So a newly created object is transient. You then persist it and it becomes persistent. You then commit the transaction and it remains managed in persistent state. When you close the EntityManager it becomes detached. Finally when you delete the object from persistence and commit that transaction it is in transient state.

161.1.3 Detachment

When you detach an object (and its graph) either explicitly (using em.detach()) or implicitly via the PersistenceContext above, you need to be careful about which fields are detached. If you detach everything then you can end up with a huge graph that could impact on the performance of your
application. On the other hand you need to ensure that you have all fields that you will be needing access to whilst detached. Should you access a field that was not detached an `IllegalAccessException` is thrown. All fields that are loaded will be detached so make sure you either load all required when retrieving the object using Entity Graphs or you access fields whilst attached (which will load them).

**Important**: Please note that some people interpret the JPA spec as implying that an object which has a primary key field set to a value as being detached. DataNucleus does not take this point of view, since the only way you can have a detached object is to detach it from persistence (i.e it was once managed/attached). To reinforce our view of things, what state is an object in which has a primitive primary key field? Using the logic above of these other people any object of such a class would be in detached state (when not managed) since its PK is set. **An object that has a PK field set is transient unless it was detached from persistence.** Note that you can merge a transient object by setting the persistence property `datanucleus.allowAttachOfTransient` to `true`.

Note that DataNucleus does not use the "CascadeType.DETACH" flag explicitly, and instead detaches the fields that are loaded (or marked for eager loading). In addition it allows the user to make use of the `FetchPlan` extension for controlling the fine details of what is loaded (and hence detached).

### 161.1.4 Helper Methods

JPA provides nothing to determine the lifecycle state of an object. Fortunately DataNucleus does consider this useful, so you can call the following

```java
String state = NucleusJPAHelper.getObjectState(entity);
boolean detached = NucleusJPAHelper.isDetached(entity);
boolean persistent = NucleusJPAHelper.isPersistent(entity);
boolean deleted = NucleusJPAHelper.isDeleted(entity);
boolean transactional = NucleusJPAHelper.isTransactional(entity);
```

When an object is detached it is often useful to know which fields are loaded/dirty. You can do this with the following helper methods

```java
Object[] detachedState = NucleusJPAHelper.getDetachedStateForObject(entity);
// detachedState[0] is the identity, detachedState[1] is the version when detached
// detachedState[2] is a BitSet for loaded fields
// detachedState[3] is a BitSet for dirty fields

String[] dirtyFieldNames = NucleusJPAHelper.getDirtyFields(entity, em);

String[] loadedFieldNames = NucleusJPAHelper.getLoadedFields(entity, em);
```
162 Lifecycle Callbacks

162.1 JPA: Lifecycle Callbacks

JPA1 defines a mechanism whereby an Entity can be marked as a listener for lifecycle events. Alternatively a separate entity listener class can be defined to receive these events. Thereafter when entities of the particular class go through lifecycle changes events are passed to the provided methods. Let’s look at the two different mechanisms

162.1.1 Entity Callbacks

An Entity itself can have several methods defined to receive events when any instances of that class pass through lifecycles changes. Let’s take an example

```java
@Entity
public class Account {
    @Id
    Long accountId;

    Integer balance;
    boolean preferred;

    public Integer getBalance() { ... }

    @PrePersist
    protected void validateCreate() {
        if (getBalance() < MIN_REQUIRED_BALANCE)
        {   
            throw new AccountException("Insufficient balance to open an account");
        }
    }

    @PostLoad
    protected void adjustPreferredStatus() {
        preferred = (getBalance() >= AccountManager.getPreferredStatusLevel());
    }
}
```

So in this example just before any "Account" object is persisted the validateCreate method will be called. In the same way, just after the fields of any "Account" object are loaded the adjustPreferredStatus method is called. Very simple.

You can register callbacks for the following lifecycle events

- PrePersist
- PostPersist
- PreRemove
- PostRemove
- PreUpdate
• PostUpdate
• PostLoad

The only other rule is that any method marked to be a callback method has to take no arguments as input, and have void return.

162.1.2 Entity Listener

As an alternative to having the actual callback methods in the Entity class itself you can define a separate class as an EntityListener. So lets take the example shown before and do it for an EntityListener.

```java
@Entity
@EntityListeners(org.datanucleus.MyEntityListener.class)
public class Account {
    @Id
    Long accountId;

    Integer balance;
    boolean preferred;

    public Integer getBalance() { ... }
}
```

```java
public class MyEntityListener {
    @PostPersist
    public void newAccountAlert(Account acct) {
        ... do something when we get a new Account
    }
}
```

So we define our "Account" entity as normal but mark it with an EntityListener, and then in the EntityListener we define the callbacks we require. As before we can define any of the 7 callbacks as we require. The only difference is that the callback method has to take an argument of type "Object" that it will be called for, and have void return.
163 Datastore Connection

163.1 JPA : Datastore Connections

DataNucleus utilises datastore connections as follows

- EMF : single connection at any one time for datastore-based value generation. Obtained just for the operation, then released
- EMF : single connection at any one time for schema-generation. Obtained just for the operation, then released
- EM : single connection at any one time. When in a transaction the connection is held from the point of retrieval until the transaction commits or rolls back. The exact point at which the connection is obtained is defined more fully below. When used for non-transactional operations the connection is obtained just for the specific operation (unless configured to retain it).

If you have multiple threads using the same EntityManager then you can get "ConnectionInUse" problems where another operation on another thread comes in and tries to perform something while that first operation is still in use. This happens because the JPA spec requires an implementation to use a single datastore connection at any one time. When this situation crops up the user ought to use multiple EntityManagers.

Another important aspect is use of queries for Optimistic transactions, or for non-transactional contexts. In these situations it isn’t possible to keep the datastore connection open indefinitely and so when the Query is executed the ResultSet is then read into core making the queried objects available thereafter.

163.1.1 Transactional Context

For pessimistic/datastore transactions a connection will be obtained from the datastore when the first persistence operation is initiated. This datastore connection will be held for the duration of the transaction until such time as either commit() or rollback() are called.

For optimistic transactions the connection is only obtained when flush()/commit() is called. When flush() is called, or the transaction committed a datastore connection is finally obtained and it is held open until commit/rollback completes. when a datastore operation is required. The connection is typically released after performing that operation. So datastore connections, in general, are held for much smaller periods of time. This is complicated slightly by use of the persistence property datanucleus.IgnoreCache. When this is set to false, the connection, once obtained, is not released until the call to commit()/rollback().

Note that for Neo4j/MongoDB a single connection is used for the duration of the EM for all transactional and nontransactional operations.

163.1.2 Nontransactional Context

When performing non-transactional operations, the default behaviour is to obtain a connection when needed, and release it after use. With RDBMS you have the option of retaining this connection ready for the next operation to save the time needed to obtain it; this is enabled by setting the persistence property datanucleus.connection.nontx.releaseAfterUse to false.

Note that for Neo4j/MongoDB a single connection is used for the duration of the EM for all transactional and nontransactional operations.
163.2 Connection Pooling

When you create an `EntityManagerFactory` using the connection URL, driver name and the username/password to use, this doesn't necessarily pool the connections. For some of the supported datastores DataNucleus allows you to utilise a connection pool to efficiently manage the connections to the datastore. We currently provide support for the following:

- **RDBMS**: DBCP we allow use of externally-defined DBCP, but also provide a builtin DBCP v1.4
- **RDBMS**: C3P0
- **RDBMS**: Proxool
- **RDBMS**: BoneCP
- **RDBMS**: Tomcat
- **RDBMS**: Manually creating a DataSource for a 3rd party software package
- **RDBMS**: Custom Connection Pooling Plugins using the DataNucleus ConnectionPoolFactory interface
- **RDBMS**: Using JNDI, and lookup a connection DataSource.
- **LDAP**: Using JNDI

You need to specify the persistence property `datanucleus.connectionPoolingType` to be whichever of the external pooling libraries you wish to use (or "None" if you explicitly want no pooling). DataNucleus provides two sets of connections to the datastore - one for transactional usage, and one for non-transactional usage. If you want to define a different pooling for nontransactional usage then you can also specify the persistence property `datanucleus.connectionPoolingType.nontx` to whichever is required.

163.2.1 RDBMS : JDBC driver properties with connection pool

If using RDBMS and you have a JDBC driver that supports custom properties, you can still use DataNucleus connection pooling and you need to specify the properties in with your normal persistence properties, but add the prefix `datanucleus.connectionPool.driver.` to the property name that the driver requires. For example if an Oracle JDBC driver accepts `defaultRowPrefetch` then you would specify something like

```
datanucleus.connectionPool.driver.defaultRowPrefetch=50
```

and it will pass in `defaultRowPrefetch` as "50" into the driver used by the connection pool.

163.2.2 RDBMS : Apache DBCP

DataNucleus allows you to utilise a connection pool using Apache DBCP to efficiently manage the connections to the datastore. DBCP is a third-party library providing connection pooling. This is accessed by specifying the persistence property `datanucleus.connectionPoolingType`. To utilise DBCP-based connection pooling we do this
So the EMF will use connection pooling using DBCP. To do this you will need commons-dbcp, commons-pool and commons-collections JARs to be in the CLASSPATH.

You can also specify persistence properties to control the actual pooling. The currently supported properties for DBCP are shown below

```
// Specify our persistence properties used for creating our EMF
Properties props = new Properties();
props.setProperty("javax.persistence.jdbc.driver","com.mysql.jdbc.Driver");
props.setProperty("javax.persistence.jdbc.url","jdbc:mysql://localhost/myDB");
props.setProperty("javax.persistence.jdbc.user","login");
props.setProperty("javax.persistence.jdbc.password","password");
props.setProperty("datanucleus.connectionPoolingType", "DBCP");
```

### 163.2.3 RDBMS : C3P0

DataNucleus allows you to utilise a connection pool using C3P0 to efficiently manage the connections to the datastore. C3P0 is a third-party library providing connection pooling. This is accessed by specifying the persistence property `datanucleus.connectionPoolingType`. To utilise C3P0-based connection pooling we do this

```
// Specify our persistence properties used for creating our EMF
Properties props = new Properties();
props.setProperty("javax.persistence.jdbc.driver","com.mysql.jdbc.Driver");
props.setProperty("javax.persistence.jdbc.url","jdbc:mysql://localhost/myDB");
props.setProperty("javax.persistence.jdbc.user","login");
props.setProperty("javax.persistence.jdbc.password","password");
props.setProperty("datanucleus.connectionPoolingType", "C3P0");
```

So the EMF will use connection pooling using C3P0. To do this you will need the C3P0 JAR to be in the CLASSPATH. If you want to configure C3P0 further you can include a "c3p0.properties" in your CLASSPATH - see the C3P0 documentation for details.

You can also specify persistence properties to control the actual pooling. The currently supported properties for C3P0 are shown below

```
# Pooling of Connections
datanucleus.connectionPool.maxIdle=10
datanucleus.connectionPool.minIdle=3
datanucleus.connectionPool.maxActive=5
datanucleus.connectionPool.maxWait=60

# Pooling of PreparedStatements
datanucleus.connectionPool.maxStatements=0

datanucleus.connectionPool.testSQL=SELECT 1
datanucleus.connectionPool.timeBetweenEvictionRunsMillis=2400000
datanucleus.connectionPool.minEvictableIdleTimeMillis=18000000
```
163.2.4 RDBMS: Proxool

DataNucleus allows you to utilise a connection pool using Proxool to efficiently manage the
collections to the datastore. **Proxool** is a third-party library providing connection pooling. This is
accessed by specifying the persistence property `datanucleus.connectionPoolingType`. To utilise
Proxool-based connection pooling we do this

```java
// Specify our persistence properties used for creating our EMF
Properties props = new Properties();
properties.setProperty("javax.persistence.jdbc.driver", "com.mysql.jdbc.Driver");
properties.setProperty("javax.persistence.jdbc.url", "jdbc:mysql://localhost/myDB");
properties.setProperty("javax.persistence.jdbc.user", "login");
properties.setProperty("javax.persistence.jdbc.password", "password");
properties.setProperty("datanucleus.connectionPoolingType", "Proxool");
```

So the **EMF** will use connection pooling using Proxool. To do this you will need the **proxool** and
**commons-logging** JARs to be in the CLASSPATH.

You can also specify persistence properties to control the actual pooling. The currently supported
properties for Proxool are shown below

```java
datanucleus.connectionPool.maxConnections=10
datanucleus.connectionPool.testSQL=SELECT 1
```

163.2.5 RDBMS: BoneCP

DataNucleus allows you to utilise a connection pool using BoneCP to efficiently manage the
collections to the datastore. **BoneCP** is a third-party library providing connection pooling. This is
accessed by specifying the persistence property `datanucleus.connectionPoolingType`. To utilise
BoneCP-based connection pooling we do this

```java
// Specify our persistence properties used for creating our EMF
Properties props = new Properties();
properties.setProperty("javax.persistence.jdbc.driver", "com.mysql.jdbc.Driver");
properties.setProperty("javax.persistence.jdbc.url", "jdbc:mysql://localhost/myDB");
properties.setProperty("javax.persistence.jdbc.user", "login");
properties.setProperty("javax.persistence.jdbc.password", "password");
properties.setProperty("datanucleus.connectionPoolingType", "BoneCP");
```
So the *EMF* will use connection pooling using BoneCP. To do this you will need the *BoneCP JAR* (and SLF4J, google-collections) to be in the CLASSPATH.

You can also specify persistence properties to control the actual pooling. The currently supported properties for BoneCP are shown below

```java
# Pooling of Connections
datanucleus.connectionPool.maxPoolSize=5
datanucleus.connectionPool.minPoolSize=3

# Pooling of PreparedStatements
datanucleus.connectionPool.maxStatements=20
```

### 163.2.6 RDBMS : Tomcat

DataNucleus allows you to utilise a connection pool using Tomcat JDBC Pool to efficiently manage the connections to the datastore. This is accessed by specifying the persistence property `datanucleus.connectionPoolingType`. To utilise Tomcat-based connection pooling we do this

```java
// Specify our persistence properties used for creating our EMF
Properties props = new Properties();
props.setProperty("javax.persistence.jdbc.driver","com.mysql.jdbc.Driver");
props.setProperty("javax.persistence.jdbc.url","jdbc:mysql://localhost/myDB");
props.setProperty("javax.persistence.jdbc.user","login");
props.setProperty("javax.persistence.jdbc.password","password");
props.setProperty("datanucleus.connectionPoolingType","tomcat");
```

So the *EMF* will use a DataSource with connection pooling using Tomcat. To do this you will need the *tomcat-jdbc* JAR to be in the CLASSPATH.

You can also specify persistence properties to control the actual pooling, like with the other pools.

### 163.2.7 RDBMS : Manually create a DataSource ConnectionFactory

We could have used the built-in DBCP support which internally creates a DataSource ConnectionFactory, alternatively the support for external DBCP, C3P0, Proxool, BoneCP etc, however we can also do this manually if we so wish. Let's demonstrate how to do this with one of the most used pools *Apache Commons DBCP*

With DBCP you need to generate a `javax.sql.DataSource`, which you will then pass to DataNucleus. You do this as follows
// Load the JDBC driver
Class.forName(dbDriver);

// Create the actual pool of connections
ObjectPool connectionPool = new GenericObjectPool(null);

// Create the factory to be used by the pool to create the connections
ConnectionFactory connectionFactory = new DriverManagerConnectionFactory(dbURL, dbUser, dbPassword);

// Create a factory for caching the PreparedStatements
KeyedObjectPoolFactory kpf = new StackKeyedObjectPoolFactory(null, 20);

// Wrap the connections with pooled variants
PoolableConnectionFactory pcf =
    new PoolableConnectionFactory(connectionFactory, connectionPool, kpf, null, false, true);

// Create the datasource
DataSource ds = new PoolingDataSource(connectionPool);

// Create our EMF
Map properties = new HashMap();
    properties.put("datanucleus.ConnectionFactory", ds);
EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPersistenceUnit", properties);

Note that we haven't passed the dbUser and dbPassword to the EMF since we no longer need to specify them - they are defined for the pool so we let it do the work. As you also see, we set the data source for the EMF. Thereafter we can sit back and enjoy the performance benefits. Please refer to the documentation for DBCP for details of its configurability (you will need commons-dbcp, commons-pool, and commons-collections in your CLASSPATH to use this above example).

163.2.8 RDBMS : Lookup a DataSource using JNDI

DataNucleus allows you to use connection pools (java.sql.DataSource) bound to a javax.naming.InitialContext with a JNDI name. You first need to create the DataSource in the container (application server/web server), and secondly you define the javax.persistence.jtaDataSource property with the DataSource JNDI name. Please read more about this in RDBMS DataSources.

163.2.9 LDAP : JNDI

If using an LDAP datastore you can use the following persistence properties to enable connection pooling

datanucleus.connectionPoolingType=JNDI

Once you have turned connection pooling on if you want more control over the pooling you can also set the following persistence properties

- datanucleus.connectionPool.maxPoolSize : max size of pool
- datanucleus.connectionPool.initialPoolSize : initial size of pool
163.3 RDBMS : Data Sources

DataNucleus allows use of a data source that represents the datastore in use. This is often just a URL defining the location of the datastore, but there are in fact several ways of specifying this data source depending on the environment in which you are running.

- Nonmanaged Context - Java Client
- Managed Context - Servlet
- Managed Context - JavaEE

163.3.1 Java Client Environment : Non-managed Context

DataNucleus permits you to take advantage of using database connection pooling that is available on an application server. The application server could be a full JEE server (e.g WebLogic) or could equally be a servlet engine (e.g Tomcat, Jetty). Here we are in a non-managed context, and we use the following properties when creating our EntityManagerFactory, and refer to the JNDI data source of the server.

If the data source is available in WebLogic, the simplest way of using a data source outside the application server is as follows.

```java
Hashtable ht = new Hashtable();
ht.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFactory");
ht.put(Context.PROVIDER_URL, "t3://localhost:7001");

Context ctx = new InitialContext(ht);
DataSource ds = (DataSource) ctx.lookup("jdbc/datanucleus");

Map properties = new HashMap();
properties.setProperty("datanucleus.ConnectionFactory", ds);
EntityManagerFactory emf = ...
```

If the data source is available in WebSphere, the simplest way of using a data source outside the application server is as follows.

```java
Hashtable ht = new Hashtable();
ht.put(Context.INITIAL_CONTEXT_FACTORY, "com.ibm.websphere.naming.WsnInitialContextFactory");
ht.put(Context.PROVIDER_URL, "iiop://server:orb port");

Context ctx = new InitialContext(ht);
DataSource ds = (DataSource) ctx.lookup("jdbc/datanucleus");

Map properties = new HashMap();
properties.setProperty("datanucleus.ConnectionFactory", ds);
EntityManagerFactory emf = ...
```
163.3.2 Servlet Environment : Managed Context

As an example of setting up such a JNDI data source for Tomcat 5.0, here we would add the following file to $TOMCAT/conf/Catalina/localhost/ as "datanucleus.xml"

```
<?xml version='1.0' encoding='utf-8'?>
<Context docBase="/home/datanucleus/" path="/datanucleus">
   <Resource name="jdbc/datanucleus" type="javax.sql.DataSource"/>
   <ResourceParams name="jdbc/datanucleus">
      <parameter>
         <name>maxWait</name>
         <value>5000</value>
      </parameter>
      <parameter>
         <name>maxActive</name>
         <value>20</value>
      </parameter>
      <parameter>
         <name>maxIdle</name>
         <value>2</value>
      </parameter>
      <parameter>
         <name>url</name>
         <value>jdbc:mysql://127.0.0.1:3306/datanucleus?autoReconnect=true</value>
      </parameter>
      <parameter>
         <name>driverClassName</name>
         <value>com.mysql.jdbc.Driver</value>
      </parameter>
      <parameter>
         <name>username</name>
         <value>mysql</value>
      </parameter>
      <parameter>
         <name>password</name>
         <value></value>
      </parameter>
   </ResourceParams>
</Context>
```

With this Tomcat JNDI data source we would then specify the data source (name) as `java:comp/env/jdbc/datanucleus`.

```
Properties properties = new Properties();
properties.setProperty("javax.persistence.jtaDataSource","java:comp/env/jdbc/datanucleus");
EntityManagerFactory emf = ... 
```
### 163.3.3 JEE Environment: Managed Context

As in the above example, we can also run in a managed context, in a JEE/Servlet environment, and here we would make a minor change to the specification of the JNDI data source depending on the application server or the scope of the jndi: global or component.

Using JNDI deployed in global environment:

```java
Properties properties = new Properties();
properties.setProperty("javax.persistence.jtaDataSource","jdbc/datanucleus");
EntityManagerFactory emf = ...
```

Using JNDI deployed in component environment:

```java
Properties properties = new Properties();
properties.setProperty("javax.persistence.jtaDataSource","java:comp/env/jdbc/datanucleus");
EntityManagerFactory emf = ...
```
164 Transactions

164.1 JPA : Transactions

A Transaction forms a unit of work. The Transaction manages what happens within that unit of work, and when an error occurs the Transaction can roll back any changes performed. Transactions can be managed by the users application, or can be managed by a framework (such as Spring), or can be managed by a JEE container. These are described below.

- **Local transactions**: managed using the JPA Transaction API
- **JTA transactions**: managed using the JTA UserTransaction API
- **Container-managed transactions**: managed by a JEE environment
- **Spring-managed transactions**: managed by SpringFramework
- **No transactions**: "auto-commit" mode
- **Flushing a Transaction**
- **Controlling transaction isolation level**
- **Read-Only transactions**

164.1.1 Locally-Managed Transactions

If using DataNucleus JPA in a J2SE environment the normal type of transaction is **RESOURCE_LOCAL**. With this type of transaction the user manages the transactions themselves, starting, committing or rolling back the transaction. With these transactions with JPA you would do something like

```java
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
try
{
    tx.begin();

    // (users code to persist objects)

    tx.commit();
}
finally
{
    if (tx.isActive())
    {  
        tx.rollback();
    }
}
em.close();
```

In this case you will have defined your **persistence-unit** to be like this
The basic idea with **Locally-Managed transactions** is that you are managing the transaction start and end.

### 164.1.2 JTA Transactions

The other type of transaction with JPA is using JTA. With this type, where you have a JTA data source from which you have a `UserTransaction`. This `UserTransaction` can have resources "joined" to it. In the case of JPA, you have two scenarios. The first scenario is where you have the `UserTransaction` created before you create your `EntityManager`. The create of the `EntityManager` will automatically join it to the current `UserTransaction`, like this

```java
UserTransaction ut = (UserTransaction) new InitialContext().lookup("java:comp/UserTransaction");
ut.setTransactionTimeout(300);

EntityManager em = emf.createEntityManager();
try {
    ut.begin();
    // perform persistence/query operations
    ut.commit();
} finally {
    em.close();
}
```

so we control the transaction using the `UserTransaction`. The second scenario is where the `UserTransaction` is started after you have the `EntityManager`. In this case we need to join our `EntityManager` to the newly created `UserTransaction`, like this

```java
<non-jta-data-source>java:comp/env/myDS</properties>
</persistence-unit>
```

or

```java
<persistence-unit name="MyUnit" transaction-type="RESOURCE_LOCAL">
    <properties>
        <property key="javax.persistence.jdbc.url" value="jdbc:mysql:..."/>
    </properties>
</persistence-unit>
```
EntityManager em = emf.createEntityManager();
try {
   // perform persistence, query operations
   UserTransaction ut = (UserTransaction)new InitialContext().lookup("java:comp/UserTransaction");
   ut.setTransactionTimeout(300);
   ut.begin();
   // Join the EntityManager operations to this UserTransaction
   em.joinTransaction();
   // Commit the persistence/query operations performed above
   ut.commit();
} finally {
   em.close();
}

In the JTA case you will have defined your persistence-unit to be like this

<persistence-unit name="MyUnit" transaction-type="JTA">
   <jta-data-source>java:comp/env/myDS</properties>
   ...
</persistence-unit>

164.1.3 Container-Managed Transactions
When using a JEE container you are giving over control of the transactions to the container. Here you have Container-Managed Transactions. In terms of your code, you would do like the above examples except that you would OMIT the tx.begin(), tx.commit(), tx.rollback() since the JEE container will be doing this for you.

164.1.4 Spring-Managed Transactions
When you use a framework like Spring you would not need to specify the tx.begin(), tx.commit(), tx.rollback() since that would be done for you.

164.1.5 No Transactions
DataNucleus allows the ability to operate without transactions. With JPA this is enabled by default (see the 2 properties datanucleus.NontransactionalRead, datanucleus.NontransactionalWrite set to true). This means that you can read objects and make updates outside of transactions. This is effectively an "auto-commit" mode.
When using non-transactional operations, you need to pay attention to the persistence property `datanucleus.nontx.atomic`. If this is true then any persist/delete/update will be committed to the datastore immediately. If this is false then any persist/delete/update will be queued up until the next transaction (or `em.close()`) and committed with that.

### 164.1.6 Flushing
During a transaction, depending on the configuration, operations don't necessarily go to the datastore immediately, often waiting until `commit`. In some situations you need persists/uploads/deletes to be in the datastore so that subsequent operations can be performed that rely on those being handled first. In this case you can **flush** all outstanding changes to the datastore using

```
em.flush();
```

A convenient vendor extension is to find which objects are waiting to be flushed at any time, like this

```
List<ObjectProvider> objs =
    ((JPAEntityManager)pm).getExecutionContext().getObjectsToBeFlushed();
```

### 164.1.7 Transaction Isolation
DataNucleus also allows specification of the transaction isolation level. This is specified via the `EntityManagerFactory` property `datanucleus.transactionIsolation`. It accepts the standard JDBC values of

- **read-uncommitted** (1): dirty reads, non-repeatable reads and phantom reads can occur
- **read-committed** (2): dirty reads are prevented; non-repeatable reads and phantom reads can occur
- **repeatable-read** (4): dirty reads and non-repeatable reads are prevented; phantom reads can occur
- **serializable** (8): dirty reads, non-repeatable reads and phantom reads are prevented

The default is read-committed. If the datastore doesn't support a particular isolation level then it will silently be changed to one that is supported. As an alternative you can also specify it on a per-transaction basis as follows (using the values in parentheses above).
164.1.8 Read-Only Transactions

Obviously transactions are intended for committing changes. If you come across a situation where you don't want to commit anything under any circumstances you can mark the transaction as "read-only" by calling

```java
EntityManager em = emf.createEntityManager();
Transaction tx = em.getTransaction();
try {
    tx.begin();
    tx.setRollbackOnly();
    // users code to persist objects
    tx.rollback();
} finally {
    if (tx.isActive()) {
        tx.rollback();
    }
}
em.close();
```

Any call to `commit` on the transaction will throw an exception forcing the user to roll it back.

164.2 JPA : Transaction Locking

A Transaction forms a unit of work. The Transaction manages what happens within that unit of work, and when an error occurs the Transaction can roll back any changes performed. There are the following locking types for a transaction.

- Transactions can lock all records in a datastore and keep them locked until they are ready to commit their changes. These are known as Pessimistic (or datastore) Locking.
- Transactions can simply assume that things in the datastore will not change until they are ready to commit, not lock any records and then just before committing make a check for changes. This is known as Optimistic Locking.

164.2.1 Pessimistic (Datastore) Locking

©2015, DataNucleus • ALL RIGHTS RESERVED.
**Pessimistic** locking isn't directly supported in JPA but are provided as a vendor extension. It is suitable for short lived operations where no user interaction is taking place and so it is possible to block access to datastore entities for the duration of the transaction. You would select pessimistic locking by adding the persistence property `datanucleus.Optimistic` as `false`.

By default DataNucleus does not currently lock the objects fetched in pessimistic locking, but you can configure this behaviour for RDBMS datastores by setting the persistence property `datanucleus.rdbms.useUpdateLock` to true. This will result in all "SELECT ... FROM ..." statements being changed to be "SELECT ... FROM ... FOR UPDATE". This will be applied only where the underlying RDBMS supports the "FOR UPDATE" syntax.

With pessimistic locking DataNucleus will grab a datastore connection at the first operation, and maintain it for the duration of the transaction. A single connection is used for the transaction (with the exception of any **Identity Generation** operations which need datastore access, so these can use their own connection).

In terms of the process of pessimistic (datastore) locking, we demonstrate this below.

<table>
<thead>
<tr>
<th>Operation</th>
<th>DataNucleus process</th>
<th>Datastore process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start transaction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Persist object</td>
<td>Prepare object (1) for persistence</td>
<td>Open connection. Insert the object (1) into the datastore</td>
</tr>
<tr>
<td>Update object</td>
<td>Prepare object (2) for update</td>
<td>Update the object (2) into the datastore</td>
</tr>
<tr>
<td>Persist object</td>
<td>Prepare object (3) for persistence</td>
<td>Insert the object (3) into the datastore</td>
</tr>
<tr>
<td>Update object</td>
<td>Prepare object (4) for update</td>
<td>Update the object (4) into the datastore</td>
</tr>
<tr>
<td>Flush</td>
<td>No outstanding changes so do nothing</td>
<td></td>
</tr>
<tr>
<td>Perform query</td>
<td>Generate query in datastore language</td>
<td>Query the datastore and return selected objects</td>
</tr>
<tr>
<td>Persist object</td>
<td>Prepare object (5) for persistence</td>
<td>Insert the object (5) into the datastore</td>
</tr>
<tr>
<td>Update object</td>
<td>Prepare object (6) for update</td>
<td>Update the object (6) into the datastore</td>
</tr>
<tr>
<td>Commit transaction</td>
<td></td>
<td>Commit connection</td>
</tr>
</tbody>
</table>

So here whenever an operation is performed, DataNucleus pushes it straight to the datastore. Consequently any queries will always reflect the current state of all objects in use. However this mode of operation has no version checking of objects and so if they were updated by external processes in the meantime then they will overwrite those changes.

It should be noted that DataNucleus provides two persistence properties that allow an amount of control over when flushing happens with pessimistic locking:

- `datanucleus.flush.mode` when set to MANUAL will try to delay all datastore operations until commit/flush.
- `datanucleus.datastoreTransactionFlushLimit` represents the number of dirty objects before a flush is performed. This defaults to 1.
164.2.2 Optimistic Locking

*Optimistic* locking is the only official option in JPA. It is suitable for longer lived operations maybe where user interaction is taking place and where it would be undesirable to block access to datastore entities for the duration of the transaction. The assumption is that data altered in this transaction will not be updated by other transactions during the duration of this transaction, so the changes are not propagated to the datastore until commit()/flush(). The data is checked just before commit to ensure the integrity in this respect. The most convenient way of checking data for updates is to maintain a column on each table that handles optimistic locking data. The user will decide this when generating their MetaData.

Rather than placing version/timestamp columns on all user datastore tables, JPA allows the user to notate particular classes as requiring *optimistic* treatment. This is performed by specifying in MetaData or annotations the details of the field/column to use for storing the version - see *versioning for JPA*. With JPA1 you must have a field in your class ready to store the version.

In JPA1 you can read the version by inspecting the field marked as storing the version value.

In terms of the process of optimistic locking, we demonstrate this below.

<table>
<thead>
<tr>
<th>Operation</th>
<th>DataNucleus process</th>
<th>Datastore process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start transaction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Persist object</td>
<td>Prepare object (1) for persistence</td>
<td></td>
</tr>
<tr>
<td>Update object</td>
<td>Prepare object (2) for update</td>
<td></td>
</tr>
<tr>
<td>Persist object</td>
<td>Prepare object (3) for persistence</td>
<td></td>
</tr>
<tr>
<td>Update object</td>
<td>Prepare object (4) for update</td>
<td></td>
</tr>
<tr>
<td>Flush</td>
<td>Flush all outstanding changes to the datastore</td>
<td>Open connection.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perform query</td>
<td>Generate query in datastore language</td>
<td></td>
</tr>
<tr>
<td>Persist object</td>
<td>Prepare object (5) for persistence</td>
<td></td>
</tr>
<tr>
<td>Update object</td>
<td>Prepare object (6) for update</td>
<td></td>
</tr>
<tr>
<td>Commit transaction</td>
<td>Flush all outstanding changes to the datastore</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here no changes make it to the datastore until the user either commits the transaction, or they invoke flush(). The impact of this is that when performing a query, by default, the results may not contain the modified objects unless they are flushed to the datastore before invoking the query. Depending
on whether you need the modified objects to be reflected in the results of the query governs what you do about that. If you invoke flush() just before running the query the query results will include the changes. The obvious benefit of optimistic locking is that all changes are made in a block and version checking of objects is performed before application of changes, hence this mode copes better with external processes updating the objects.

Please note that for some datastores (e.g RDBMS) the version check followed by update/delete is performed in a single statement.

See also :-

- JPA MetaData reference for <version> element
- JPA Annotations reference for @Version
165 Entity Graphs

165.1 JPA : Entity Graphs
When an object is retrieved from the datastore by JPA typically not all fields are retrieved immediately. This is because for efficiency purposes only particular field types are retrieved in the initial access of the object, and then any other objects are retrieved when accessed (lazy loading). The group of fields that are loaded is called an entity graph. There are 3 types of "entity graphs" to consider

• "Default Entity Graph" : implicitly defined in all JPA specs, specifying the fetch setting for each field/property (LAZY/EAGER).
• Named Entity Graphs : a new feature in JPA 2.1 allowing the user to define Named Entity Graphs in metadata, via annotations or XML
• Unnamed Entity Graphs : a new feature in JPA 2.1 allowing the user to define Entity Graphs via the JPA API at runtime

165.1.1 Default Entity Graph
JPA provides an initial entity graph, comprising the fields that will be retrieved when an object is retrieved if the user does nothing to define the required behaviour. You define this "default" by setting the fetch attribute in metadata for each field/property.

165.1.2 Named Entity Graphs
You can predefine Named Entity Graphs in metadata which can then be used at runtime when retrieving objects from the datastore (via find/query). For example, if we have the following class

```java
class MyClass
{
    String name;
    HashSet coll;
    MyOtherClass other;
}
```

and we want to have the option of the other field loaded whenever we load objects of this class, we define our annotations as

```java
@Entity
@NamedEntityGraph(name="includeOther", attributeNodes={@NamedAttributeNode("other")})
public class MyClass
{
    ...
}
```

So we have defined an EntityGraph called "includeOther" that just includes the field with name other. We can retrieve this and then use it in our persistence code, as follows
Here we have made use of the `EntityManager.find` method and provided the property `javax.persistence.loadgraph` to be our EntityGraph. This means that it will fetch all fields in the default EntityGraph, plus all fields in the `includeOther` EntityGraph. If we had provided the property `javax.persistence.fetchgraph` set to our EntityGraph it would have fetched just the fields defined in that EntityGraph.

Note that you can also make use of EntityGraphs when using the JPA Query API, specifying the same properties above but as query hints.

### 165.1.3 Unnamed Entity Graphs

You can define **Entity Graphs** at runtime, programmatically. For example, if we have the following class

```java
class MyClass {
    String name;
    HashSet coll;
    MyOtherClass other;
}
```

and we want to have the option of the `other` field loaded whenever we load objects of this class, we do the following

```java
EntityGraph includeOtherGraph = em.createEntityGraph(MyClass.class);
includeOtherGraph.addAttributeNodes("other");
```

So we have defined an EntityGraph that just includes the field with name `other`. We can then use this at runtime in our persistence code, as follows

```java
Properties props = new Properties();
props.put("javax.persistence.loadgraph", includeOtherGraph);
MyClass myObj = em.find(MyClass.class, id, props);
```

Here we have made use of the `EntityManager.find` method and provided the property `javax.persistence.loadgraph` to be our EntityGraph. This means that it will fetch all fields in the default EntityGraph, plus all fields in this EntityGraph. If we had provided the property `javax.persistence.fetchgraph` set to our EntityGraph it would have fetched just the fields defined in that EntityGraph.

Note that you can also make use of EntityGraphs when using the JPA Query API, specifying the same properties above but as query hints.
166 Query API

166.1 JPA : Query API

Once you have persisted objects you need to query them. For example if you have a web application representing an online store, the user asks to see all products of a particular type, ordered by the price. This requires you to query the datastore for these products. JPA specifies support for a pseudo-OO query language (JPQL), a relational query language (SQL) and (RDBMS) Stored Procedures (JPA2.1+).

Which query language is used is down to the developer. The data-tier of an application could be written by a primarily Java developer, who would typically think in an object-oriented way and so would likely prefer JPQL. On the other hand the data-tier could be written by a datastore developer who is more familiar with SQL concepts and so could easily make more use of SQL. This is the power of an implementation like DataNucleus in that it provides the flexibility for different people to develop the data-tier utilising their own skills to the full without having to learn totally new concepts.

There are 2 categories of queries with JPA :-

- **Programmatic Query** where the query is defined using the JPA Query API.
- **Named Query** where the query is defined in MetaData and referred to by its name at runtime (for JPQL, SQL and Stored Procedures).

Let's now try to understand the Query API in JPA.

### 166.1.1 JPQL Query

Let's create a JPQL query to highlight its usage

```java
Query q = em.createQuery("SELECT p FROM Product p WHERE p.param2 < :threshold ORDER BY p.param1 ascending");
q.setParameter("threshold", my_threshold);
List results = q.getResultList();
```

In this Query, we implicitly select JPQL by using the method `EntityManager.createQuery()`, and the query is specified to return all objects of type `Product` (or subclasses) which have the field `param2` less than some threshold value ordering the results by the value of field `param1`. We've specified the query like this because we want to pass the threshold value in as a parameter (so maybe running it once with one value, and once with a different value). We then set the parameter value of our `threshold` parameter. The Query is then executed to return a List of results. The example is to highlight the typical methods specified for a (JPQL) Query.

### 166.1.2 SQL Query

Let's create an SQL query to highlight its usage

```java
Query q = em.createNativeQuery("SELECT * FROM Product p WHERE p.param2 < ?1");
q.setParameter(1, my_threshold);
List results = q.getResultList();
```

© 2015, DataNucleus • ALL RIGHTS RESERVED.
So we implicitly select SQL by using the method `EntityManager.createNativeQuery()`, and the query is specified like in the JPQL case to return all instances of type `Product` (using the table name in this SQL query) where the column `param2` is less than some threshold value.

### 166.1.3 `setFirstResult()`, `setMaxResults()`

In JPA to specify the range of a query you have two methods available. So you could do

```java
Query q = em.createQuery("SELECT p FROM Product p WHERE p.param2 < :threshold ORDER BY p.param1 ascending");
q.setFirstResult(1);
q.setMaxResults(3);
```

so we will get results 1, 2, and 3 returned only. The first result starts at 0 by default.

### 166.1.4 `setHint()`

JPA's query API allows implementations to support extensions ("hints") and provides a simple interface for enabling the use of such extensions on queries.

```java
q.setHint("extension_name", value);
```

DataNucleus provides various extensions for different types of queries.

### 166.1.5 `setParameter()`

JPA's query API supports named and numbered parameters and provides method for setting the value of particular parameters. To set a named parameter, for example, you could do

```java
Query q = em.createQuery("SELECT p FROM Product p WHERE p.param2 < :threshold ORDER BY p.param1 ascending");
q.setParameter("threshold", value);
```

To set a numbered parameter you could do

```java
Query q = em.createQuery("SELECT p FROM Product p WHERE p.param2 < ?1 ORDER BY p.param1 ascending");
q.setParameter(1, value);
```

Numbered parameters are numbered from 1.

### 166.1.6 `getResultList()`

To execute a JPA query you would typically call `getResultList`. This will return a List of results. This should not be called when the query is an "UPDATE"/"DELETE".
166.1.7 getSingleResult()

To execute a JPA query where you are expecting a single value to be returned you would call `getSingleResult`. This will return the single Object. If the query returns more than one result then you will get an Exception. This should not be called when the query is an "UPDATE"/"DELETE".

```java
Query q = em.createQuery("SELECT p FROM Product p WHERE p.param2 = :value");
q.setParameter("value", val1);
Product prod = q.getSingleResult();
```

166.1.8 executeUpdate()

To execute a JPA UPDATE/DELETE query you would call `executeUpdate`. This will return the number of objects changed by the call. This should not be called when the query is a "SELECT".

```java
Query q = em.createQuery("DELETE FROM Product p");
int number = q.executeUpdate();
```

166.1.9 setFlushMode()

By default, when a query is executed it will be evaluated against the contents of the datastore at the point of execution. If there are any outstanding changes waiting to be flushed then these will not feature in the results. To make sure all outstanding changes are respected

```java
q.setFlushMode(FlushModeType.AUTO);
```

166.1.10 setLockMode()

JPA allows control over whether objects found by a fetch (JPQL query) are locked during that transaction so that other transactions can't update them in the meantime. For example

```java
q.setLockMode(LockModeType.PESSIMISTIC_READ);
```

You can also specify this for all queries for all EntityManagers using a persistence property `datanucleus.rdbms.useUpdateLock`.
167 Query Cache

167.1 JPA: Query Caching

JPA doesn't currently define a mechanism for caching of queries. DataNucleus provides 3 levels of caching:

- **Generic Compilation**: when a query is compiled it is initially compiled *generically* into expression trees. This generic compilation is independent of the datastore in use, so can be used for other datastores. This can be cached.

- **Datastore Compilation**: after a query is compiled into expression trees (above) it is then converted into the native language of the datastore in use. For example with RDBMS, it is converted into SQL. This can be cached.

- **Results**: when a query is run and returns objects of the candidate type, you can cache the identities of the result objects.

167.1.1 Generic Query Compilation Cache

This cache is by default set to *weak*, meaning that the generic query compilation is cached using weak references. This is set using the persistence property `datanucleus.cache.queryCompilation.type`. You can also set it to *strong* meaning that strong references are used, or *soft* meaning that soft references are used, or finally to *none* meaning that there is no caching of generic query compilation information.

You can turn caching on/off (default = on) on a query-by-query basis by specifying the query extension `datanucleus.query.compilation.cached` as true/false.

167.1.2 Datastore Query Compilation Cache

This cache is by default set to *weak*, meaning that the datastore query compilation is cached using weak references. This is set using the persistence property `datanucleus.cache.queryCompilationDatastore.type`. You can also set it to *strong* meaning that strong references are used, or *soft* meaning that soft references are used, or finally to *none* meaning that there is no caching of datastore-specific query compilation information.

You can turn caching on/off (default = on) on a query-by-query basis by specifying the query extension `datanucleus.query.compilation.cached` as true/false. As a finer degree of control, where cached results are used, you can omit the validation of object existence in the datastore by setting the query extension `datanucleus.query.resultCache.validateObjects`. 
167.1.3 Query Results Cache

This cache is by default set to weak, meaning that the datastore query results are cached using weak references. This is set using the persistence property `datanucleus.cache.queryResult.type`. You can also set it to strong meaning that strong references are used, or soft meaning that soft references are used, or finally to none meaning that there is no caching of query results information. You can also specify `datanucleus.cache.queryResult.cacheName` to define the name of the cache used for the query results cache.

You can turn caching on/off (default = off) on a query-by-query basis by specifying the query extension `datanucleus.query.results.cached` as true/false.

Obviously with a cache of query results, you don't necessarily want to retain this cached over a long period. In this situation you can evict results from the cache like this.

```java
import org.datanucleus.api.jpa.JPAQueryCache;
import org.datanucleus.api.jpa.EntityManagerFactoryImpl;
...
JPAQueryCache cache = ((EntityManagerFactoryImpl)emf).getQueryCache();
cache.evict(query);
```

which evicts the results of the specific query. The JPAQueryCache has more options available should you need them ... .
168.1 JPA : JPQL SELECT Queries

The JPA specification defines JPQL (a pseudo-OO query language, with SQL-like syntax), for selecting objects from the datastore. To provide a simple example, this is what you would do

```java
Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = 'Jones'");
List results = (List)q.getResultList();
```

This finds all "Person" objects with surname of "Jones". You specify all details in the query.

168.1.1 SELECT Syntax

In JPQL queries you define the query in a single string, defining the result, the candidate class(es), the filter, any grouping, and the ordering. This string has to follow the following pattern

```
SELECT [\<result\>]
[FROM \<candidate-class(es)\>]
[WHERE \<filter\>]
[GROUP BY \<grouping\>]
[HAVING \<having\>]
[ORDER BY \<ordering\>]
```

The "keywords" in the query are shown in UPPER CASE are case-insensitive.

168.1.2 Entity Name

In the example shown you note that we did not specify the full class name. We used `Person p` and thereafter could refer to `p` as the alias. The `Person` is called the entity name and in JPA MetaData this can be defined against each class in its definition. For example

```xml
<entity class="org.datanucleus.company.Person" name="Person">
  ...
</entity>
```

In strict JPA the entity name cannot be a `MappedSuperclass` entity name. That is, if you have an abstract superclass that is persistable, you cannot query for instances of that superclass and its subclasses. We consider this a significant shortcoming of the querying capability, and allow the entity name to also be of a `MappedSuperclass`. You are unlikely to find this supported in other JPA implementations, but then maybe that's why you chose DataNucleus?
168.1.3 From Clause
The FROM clause allows a user to add some explicit joins to related entities, and assign aliases to the joined entities. These are then usable in the filter/ordering/result etc. If you don't add any joins DataNucleus will add joins where they are implicit from the filter expression for example

168.1.4 Fetched Fields
By default a query will fetch fields according to their defined EAGER/LAZY setting, so fields like primitives, wrappers, Dates, and 1-1/N-1 relations will be fetched, whereas 1-N/M-N fields will not be fetched. JPQL allows you to include FETCH JOIN as a hint to include 1-N/M-N fields where possible. For RDBMS datastores any multi-valued field will be bulk-fetched if it is defined to be EAGER or is placed in the current EntityGraph. By bulk-fetched we mean that there will be a single SQL issued per collection field (hence avoiding the N+1 problem). Note that you can disable this by either not putting multi-valued fields in the FetchPlan, or by setting the query extension "datanucleus.multivaluedFetch" to "none" (default is "bulk-fetch" using the single SQL per field). All non-RDBMS datastores do respect this FETCH JOIN setting, since a collection/map is stored in a single "column" in the object and so is readily retrievable.

Note that you can also make use of Entity Graphs to have fuller control over what is retrieved from each query.

168.1.5 Filter
The most important thing to remember when defining the filter for JPQL is that think how you would write it in SQL, and its likely the same except for field names instead of column names. The filter has to be a boolean expression, and can include the candidate entity, fields/properties, literals, functions, parameters, operators and subqueries

168.1.6 Fields/Properties
In JPQL you refer to fields/properties in the query by referring to the field/bean name. For example, if you are querying a candidate entity called Product and it has a field "price", then you access it like this

```
price < 150.0
```

Note that if you want to refer to a field/property of an entity you can prefix the field by its alias

```
p.price < 150.0
```

You can also chain field references if you have an entity Product (alias = p) with a field of (persistable) type Inventory, which has a field name, so you could do

```
p.inventory.name = 'Backup'
```

168.1.7 Operators
The operators are listed below in order of decreasing precedence.

- Navigation operator (.)
• Arithmetic operators:
  • +, - unary
  • *, / multiplication and division
  • +, - addition and subtraction
• Comparison operators: :=, >, >=, <, <=, <> (not equal), [NOT] BETWEEN, [NOT] LIKE,
• Logical operators:
  • NOT
  • AND
  • OR

168.1.8 Literals
JPQL supports the following literals: IntegerLiteral, FloatingPointLiteral, BooleanLiteral,
CharacterLiteral, StringLiteral, and NullLiteral. When String literals are specified using single-string
format they should be surrounded by single-quotes ‘.

168.1.9 Input Parameters
In JPQL queries it is convenient to pass in parameters so we don't have to define the same query for
different values. Let's take two examples

```
168.1.10 JPQL Functions
JPQL provides an SQL-like query language. Just as with SQL, JPQL also supports a range of
functions to enhance the querying possibilities. The tables below also mark whether a particular
method is supported for evaluation in-memory.
```
Please note that you can easily add support for other functions for evaluation "in-memory" using this DataNucleus plugin point

Please note that you can easily add support for other functions with RDBMS datastore using this DataNucleus plugin point

168.1.10.1 Aggregate Functions

There are a series of aggregate functions for aggregating the values of a field for all rows of the results.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Standard</th>
<th>In-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT(field)</td>
<td>Returns the aggregate count of the field (Long)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIN(field)</td>
<td>Returns the minimum value of the field (type of the field)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAX(field)</td>
<td>Returns the maximum value of the field (type of the field)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVG(field)</td>
<td>Returns the average value of the field (Double)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUM(field)</td>
<td>Returns the sum of the field value(s) (Long, Double, BigInteger, BigDecimal)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

168.1.10.2 String Functions

There are a series of functions to be applied to String fields.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Standard</th>
<th>In-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONCAT(str_field, str_field2 [, str_fieldX])</td>
<td>Returns the concatenation of the string fields</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBSTRING(str_field, num1 [, num2])</td>
<td>Returns the substring of the string field starting at position num1, and optionally with the length of num2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRIM([trim_spec] [trim_char] [FROM] str_field)</td>
<td>Returns trimmed form of the string field</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOWER(str_field)</td>
<td>Returns the lower case form of the string field</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
168.1.10.3 Temporal Functions
There are a series of functions for use with temporal values

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Standard</th>
<th>In-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT_DATE</td>
<td>Returns the current date (day month year) of the datastore server</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>CURRENT_TIME</td>
<td>Returns the current time (hour minute second) of the datastore server</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>CURRENT_TIMESTAMP</td>
<td>Returns the current timestamp of the datastore server</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>YEAR(dateField)</td>
<td>Returns the year of the specified date</td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>MONTH(dateField)</td>
<td>Returns the month of the specified date</td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>DAY(dateField)</td>
<td>Returns the day of the month of the specified date</td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>HOUR(dateField)</td>
<td>Returns the hour of the specified date</td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>MINUTE(dateField)</td>
<td>Returns the minute of the specified date</td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>SECOND(dateField)</td>
<td>Returns the second of the specified date</td>
<td></td>
<td>✗</td>
</tr>
</tbody>
</table>

168.1.10.4 Collection Functions
There are a series of functions for use with collection values

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Standard</th>
<th>In-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX(field)</td>
<td>Returns index number of the field element when that is the element of an indexed List field.</td>
<td>✔️</td>
<td>✗</td>
</tr>
</tbody>
</table>
168.1.10.5 Arithmetic Functions

There are a series of functions for arithmetic use

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Standard</th>
<th>In-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS(numeric_field)</td>
<td>Returns the absolute value of the numeric field</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRT(numeric_field)</td>
<td>Returns the square root of the numeric field</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOD(num_field1, num_field2)</td>
<td>Returns the modulus of the two numeric fields (num_field1 % num_field2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

168.1.10.6 Other Functions

You have a further function available

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Standard</th>
<th>In-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION(name, [arg1 [arg2 ...]])</td>
<td>Executes the specified SQL function &quot;name&quot; with the defined arguments</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

168.1.11 Ordering of Results

By default your results will be returned in the order determined by the datastore, so don't rely on any particular order. You can, of course, specify the order yourself. You do this using field/property names and ASC/DESC keywords. For example

```
field1 ASC, field2 DESC
```

which will sort primarily by field1 in ascending order, then secondarily by field2 in descending order.

Although it is not (yet) standard JPQL, DataNucleus also supports specifying a directive for where NULL values of the ordered field/property go in the order, so the full syntax supported is

```
fieldName [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Note that this is only supported for a few RDBMS (H2, HSQLDB, PostgreSQL, DB2, Oracle, Derby, Firebird, SQLServer v11+).
168.1.12 Subqueries

With JPQL the user has a very flexible query syntax which allows for querying of the vast majority of data components in a single query. In some situations it is desirable for the query to utilise the results of a separate query in its calculations. JPQL also allows the use of subqueries. Here's an example

```java
SELECT Object(e) FROM org.datanucleus.Employee e
WHERE e.salary > (SELECT avg(f.salary) FROM org.datanucleus.Employee f)
```

So we want to find all Employees that have a salary greater than the average salary. The subquery must be in parentheses (brackets). Note that we have defined the subquery with an alias of "f", whereas in the outer query the alias is "e".

168.1.12.1 ALL/ANY Expressions

One use of subqueries with JPQL is where you want to compare with some or all of a particular expression. To give an example

```java
SELECT emp FROM Employee emp
WHERE emp.salary > ALL (SELECT m.salary FROM Manager m WHERE m.department = emp.department)
```

So this returns all employees that earn more than all managers in the same department! You can also compare with some/any, like this

```java
SELECT emp FROM Employee emp
WHERE emp.salary > ANY (SELECT m.salary FROM Manager m WHERE m.department = emp.department)
```

So this returns all employees that earn more than any one Manager in the same department.

168.1.12.2 EXISTS Expressions

Another use of subqueries in JPQL is where you want to check on the existence of a particular thing. For example

```java
SELECT DISTINCT emp FROM Employee emp
WHERE EXISTS (SELECT emp2 FROM Employee emp2 WHERE emp2 = emp.spouse)
```

So this returns the employees that have a partner also employed.

168.1.13 Range of Results

With JPQL you can select the range of results to be returned. For example if you have a web page and you are paginating the results of some search, you may want to get the results from a query in blocks of 20 say, with results 0 to 19 on the first page, then 20 to 39, etc. You can facilitate this as follows

```java
Query q = em.createQuery("SELECT p FROM Person p WHERE p.age > 20");
q.setFirstResult(0);
q.setMaxResults(20);
```

So with this query we get results 0 to 19 inclusive.
168.1.14 Query Result

Whilst the majority of the time you will want to return instances of a candidate class, JPQL also allows you to return customised results. Consider the following example

```java
Query q = em.createQuery("SELECT p.firstName, p.lastName FROM Person p WHERE p.age > 20");
List<Object[]> results = q.getResultList();
```

this returns the first and last name for each Person meeting that filter. Obviously we may have some container class that we would like the results returned in, so if we change the query to this

```java
Query<PersonName> q = em.createQuery(
   "SELECT p.firstName, p.lastName FROM Person p WHERE p.age > 20", PersonName.class);
List<PersonName> results = q.getResultList();
```

so each result is a PersonName, holding the first and last name. This result class needs to match one of the following structures

- Constructor taking arguments of the same types and the same order as the result clause. An instance of the result class is created using this constructor. For example

```java
public class PersonName{
    protected String firstName = null;
    protected String lastName = null;

    public PersonName(String first, String last){
        this.firstName = first;
        this.lastName = last;
    }

    ...
}
```

- Default constructor, and setters for the different result columns, using the alias name for each column as the property name of the setter. For example

```java
public class PersonName{
    protected String firstName = null;
    protected String lastName = null;

    public PersonName(){
    }

    public void setFirstName(String first) {this.firstName = first;}
    public void setLastName(String last) {this.lastName = last;}

    ...
}
```
Note that if the setter property name doesn’t match the query result component name, you should use `AS {alias}` in the query so they are the same.

### 168.1.15 Query Execution

There are two ways to execute a JPQL query. When you know it will return 0 or 1 results you call

```java
Object result = query.getSingleResult();
```

If however you know that the query will return multiple results, or you just don’t know then you would call

```java
List results = query.getResultList();
```

### 168.2 JPQL In-Memory queries

The typical use of a JPQL query is to translate it into the native query language of the datastore and return objects matched by the query. For many datastores it is simply impossible to support the full JPQL syntax in the datastore native query language and so it is necessary to evaluate the query in-memory. This means that we evaluate as much as we can in the datastore and then instantiate those objects and evaluate further in-memory. Here we document the current capabilities of in-memory evaluation in DataNucleus.

- Subqueries using ALL, ANY, SOME, EXISTS are not currently supported
- MEMBER OF syntax is not currently supported.

To enable evaluation in memory you specify the query hint `datanucleus.query.evaluateInMemory` to `true` as follows

```java
query.setHint("datanucleus.query.evaluateInMemory","true");
```

### 168.3 Named Query

With the JPA API you can either define a query at runtime, or define it in the MetaData/annotations for a class and refer to it at runtime using a symbolic name. This second option means that the method of invoking the query at runtime is much simplified. To demonstrate the process, let’s say we have a class called `Product` (something to sell in a store). We define the JPA Meta-Data for the class in the normal way, but we also have some query that we know we will require, so we define the following in the Meta-Data.

```xml
<entity class="Product">
  ...
  <named-query name="SoldOut"><![CDATA[
    SELECT p FROM Product p WHERE p.status = "Sold Out"
  ]]></named-query>
</entity>
```

or using annotations
@Entity
@NamedQuery(name="SoldOut", query="SELECT p FROM Product p WHERE p.status == 'Sold Out'")
public class Product {...}

Note that DataNucleus also supports specifying this using annotations in non-Entity classes. This is beyond the JPA spec, but is very useful in real applications.

So we have a JPQL query called "SoldOut" defined for the class Product that returns all Products (and subclasses) that have a status of "Sold Out". Out of interest, what we would then do in our application to execute this query would be

```java
Query query = em.createNamedQuery("SoldOut");
List<Product> results = query.getResultList();
```

### 168.4 JPQL DELETE Queries

The JPA specification defines a mode of JPQL for deleting objects from the datastore.

#### 168.4.1 DELETE Syntax

The syntax for deleting records is very similar to selecting them

```sql
DELETE FROM [<candidate-class>]
 [WHERE <filter>]
```

The "keywords" in the query are shown in UPPER CASE are case-insensitive.

```java
Query query = em.createQuery("DELETE FROM Person p WHERE firstName = 'Fred'" );
int numRowsDeleted = query.executeUpdate();
```

### 168.5 JPQL UPDATE Queries

The JPA specification defines a mode of JPQL for updating objects in the datastore.

#### 168.5.1 UPDATE Syntax

The syntax for updating records is very similar to selecting them

```sql
UPDATE [<candidate-class>] SET item1=value1, item2=value2
 [WHERE <filter>]
```

The "keywords" in the query are shown in UPPER CASE are case-insensitive.

```java
Query query = em.createQuery("UPDATE Person p SET p.salary = 10000 WHERE age = 18" );
int numRowsUpdated = query.executeUpdate();
```
168.6 JPQL BNF Notation

The BNF defining the JPQL query language is shown below.
QL_statement ::= select_statement | update_statement | delete_statement
select_statement ::= select_clause from_clause [where_clause] [groupby_clause] [having_clause] [orderby_clause]

update_statement ::= update_clause [where_clause]
delete_statement ::= delete_clause [where_clause]

from_clause ::= FROM identification_variable_declaration
   {, (identification_variable_declaration | collection_member_declaration)}*
identification_variable_declaration ::= range_variable_declaration { join | fetch_join }*
range_variable_declaration ::= entity_type_expression [AS] identification_variable

join ::= join_spec join_association_path_expression [AS] identification_variable
fetch_join ::= join_spec FETCH join_association_path_expression
join_spec ::= [ LEFT | OUTER | INNER ] JOIN
join_association_path_expression ::= join_collection_valued_path_expression | join_single_valued_path_expression
join_collection_valued_path_expression ::= identification_variable.(collection_valued_field)*
collection_valued_field ::= identification_variable.(collection_valued_object_field.*
collection_valued_path_expression ::= IN (collection_valued_path_expression) [AS] identification_variable
qualified_identification_variable ::= KEY(identification_variable) | VALUE(identification_variable)
   | ENTRY(identification_variable)
state_field_path_expression ::= qualified_identification_variable |
   state_field_object_path_expression
single_valued_field ::= qualified_identification_variable |
   single_valued_object_field

general_identification_variable ::= identification_variable | KEY(identification_variable) |
   VALUE(identification_variable)
state_field_object_path_expression ::= general_identification_variable.(single_valued_object_field.*
single_valued_object_field

update_clause ::= UPDATE entity_name [(AS] identification_variable) SET update_item {, update_item}*
update_item ::= identification_variable.[state_field | single_valued_object_field] = new_value
new_value ::= scalar_expression | simple_entity_expression | NULL
delete_clause ::= DELETE FROM entity_name [(AS] identification_variable)

select_clause ::= SELECT [DISTINCT] select_item {, select_item}*
select_item ::= select_expression [(AS] result_variable]
select_expression ::= single_valued_path_expression | scalar_expression | aggregate_expression |
   identification_variable | OBJECT(identification_variable) | constructor_expression
constructor_expression ::= NEW constructor_name ( constructor_item {, constructor_item} *)
constructor_item ::= single_valued_path_expression | scalar_expression | aggregate_expression |
   identification_variable
aggregate_expression ::= [ AVG | MAX | MIN | SUM ] {[DISTINCT] state_field_path_expression} | COUNT {[DISTINCT] identification_variable | state_field_path_expression |
   single_valued_object_path_expression}

where_clause ::= WHERE conditional_expression
groupby_clause ::= GROUP BY groupby_item {, groupby_item}*
groupby_item ::= single_valued_path_expression | identification_variable
having_clause ::= HAVING conditional_expression
orderby_clause ::= ORDER BY orderby_item {, orderby_item}*
orderby_item ::= state_field_path_expression | result_variable [ ASC | DESC ]

subquery ::= simple_select_clause subquery_from_clause [where_clause] [groupby_clause] [having_clause]
169  JPQL Criteria

169.1  JPA : JPQL Criteria Queries

In JPA2 there is a query API referred to as "criteria". This is really an API allowing the construction of queries expression by expression, and optionally making it type safe. It provides two ways of specifying a field/property. The first way is using Strings, and the second using a MetaModel. The advantage of the MetaModel is that it means that your queries are refactorable if you rename a field. Each example will be expressed in both ways where appropriate so you can see the difference.

169.1.1  Creating a Criteria query

To use the JPA Criteria API, firstly you need to create a CriteriaQuery object for the candidate in question, and set the candidate, its alias, and the result to be of the candidate type

```java
CriteriaBuilder cb = emf.getCriteriaBuilder();
CriteriaQuery<Person> crit = cb.createQuery(Person.class);
Root<Person> candidateRoot = crit.from(Person.class);
candidateRoot.alias("p");
crit.select(candidateRoot);
```

So what we have there equates to

```
SELECT p FROM mydomain.Person p
```

For a complete list of all methods available on CriteriaBuilder, refer to javadoc

For a complete list of all methods available on CriteriaQuery, refer to javadoc

169.1.2  JPQL equivalent of the Criteria query

If you ever want to know what is the equivalent JPQL string-based query for your Criteria, just print out criteriaQuery.toString(). This is not part of the JPA spec, but something that we feel is very useful so is provided as a DataNucleus vendor extension. So, for example, the criteria query above would result in the following from crit.toString()

```
SELECT p FROM mydomain.Person p
```

169.1.3  Criteria API : Result clause

The basic Criteria query above is fine, but you may want to define a result other than the candidate. To do this we need to use the Criteria API.
Path nameField = candidateRoot.get("name");
crit.select(nameField);

which equates to

SELECT p.name

Note that here we accessed a field by its name (as a String). We could easily have accessed it via the Criteria MetaModel too.

### 169.1.4 Criteria API: From clause joins

The basic Criteria query above is fine, but you may want to define some explicit joins. To do this we need to use the Criteria API.

```java
Metamodel model = emf.getMetamodel();
ManagedType personType = model.type(Person.class);
Attribute addressAttr = personType.getAttribute("address");
Join addressJoin = candidateRoot.join((SingularAttribute)addressAttr);
addressJoin.alias("a");
```

which equates to

```
FROM mydomain.Person p JOIN p.address a
```

### 169.1.5 Criteria API: Filter

The basic Criteria query above is fine, but in the majority of cases we want to define a filter. To do this we need to use the Criteria API.

**String-based:**

```java
Predicate nameEquals = cb.equal(candidateRoot.get("name"), "First");
crit.where(nameEquals);
```

**MetaModel-based:**

```java
Predicate nameEquals = cb.equal(candidateRoot.get(Person_.name), "First");
crit.where(nameEquals);
```

You can also invoke methods, so a slight variation on this clause would be

**String-based:**

```java
Predicate nameUpperEquals = cb.equal(cb.upper(candidateRoot.get("name")), "FIRST");
```

**MetaModel-based:**

```java
Predicate nameUpperEquals = cb.equal(cb.upper(candidateRoot.get(Person_.name)), "FIRST");
```

which equates to

```
WHERE (UPPER(p.name) = 'FIRST')
```
169.1.6 Criteria API: Ordering

The basic Criteria query above is fine, but in many cases we want to define ordering. To do this we need to use the Criteria API.

```java
String-based:
Order orderFirstName = cb.desc(candidateRoot.get("name"));
crit.orderBy(orderFirstName);

MetaModel-based:
Order orderFirstName = cb.desc(candidateRoot.get(Person_.name));
crit.orderBy(orderFirstName);
```

which equates to

```
ORDER BY p.name DESC
```

169.1.7 Criteria API: Parameters

Another common thing we would want to do is specify input parameters. To do this we need to use the Criteria API. Let's take an example of a filter with parameters.

```java
String-based:
ParameterExpression param1 = cb.parameter(String.class, "myParam1");
Predicate nameEquals = cb.equal(candidateRoot.get("name"), param1);
crit.where(nameEquals);

MetaModel-based:
ParameterExpression param1 = cb.parameter(String.class, "myParam1");
Predicate nameEquals = cb.equal(candidateRoot.get(Person_.name), param1);
crit.where(nameEquals);
```

which equates to

```
WHERE (p.name = :myParam)
```

Don't forget to set the value of the parameters before executing the query!

169.1.8 Executing a Criteria query

Ok, so we've seen how to generate a Criteria query. So how can we execute it? This is simple; convert it into a standard JPA query, set any parameter values and execute it.

```java
Query query = em.createQuery(crit);
List<Person> results = query.getResultList();
```

169.1.9 Criteria API: UPDATE query

So the previous examples concentrated on SELECT queries. Let's now do an UPDATE
169 JPQL Criteria

String-based:
CriteriaUpdate<Person> crit = qb.createCriteriaUpdate(Person.class);
Root<Person> candidate = crit.from(Person.class);
candidate.alias("p");
crit.set(candidate.get("firstName"), "Freddie");
Predicate teamName = qb.equal(candidate.get("firstName"), "Fred");
crit.where(teamName);
Query q = em.createQuery(crit);
int num = q.executeUpdate();

MetaModel-based:
CriteriaUpdate<Person> crit = qb.createCriteriaUpdate(Person.class);
Root<Person> candidate = crit.from(Person.class);
candidate.alias("p");
crit.set(candidate.get(Person.firstName), "Freddie");
Predicate teamName = qb.equal(candidate.get(Person.firstName), "Fred");
crit.where(teamName);
Query q = em.createQuery(crit);
int num = q.executeUpdate();

which equates to

UPDATE Person p SET p.firstName = 'Freddie' WHERE p.firstName = 'Fred'

169.1.10 Criteria API : DELETE query

So the previous examples concentrated on SELECT queries. Let's now do a DELETE

String-based:
CriteriaDelete<Person> crit = qb.createCriteriaDelete(Person.class);
Root<Person> candidate = crit.from(Person.class);
candidate.alias("p");
Predicate teamName = qb.equal(candidate.get("firstName"), "Fred");
crit.where(teamName);
Query q = em.createQuery(crit);
int num = q.executeUpdate();

MetaModel-based:
CriteriaDelete<Person> crit = qb.createCriteriaDelete(Person.class);
Root<Person> candidate = crit.from(Person.class);
candidate.alias("p");
Predicate teamName = qb.equal(candidate.get(Person.firstName), "Fred");
crit.where(teamName);
Query q = em.createQuery(crit);
int num = q.executeUpdate();

which equates to

DELETE FROM Person p WHERE p.firstName = 'Fred'
169.1.11 MetaModel

As we mentioned at the start of this section, there is a MetaModel allowing refactorability. In JPA the MetaModel is a static metamodel of generated classes that mirror the applications persistable classes and have persistable fields marked as public and static so that they can be accessed when generating the queries. In the examples above you saw reference to a class with name with suffix "_.". This is a metamodel class. It is defined below.

The JPA2 spec contains the following description of the static metamodel.  
For every managed class in the persistence unit, a corresponding metamodel class is produced as follows:

- For each managed class X in package p, a metamodel class X_ in package p is created.
- The name of the metamodel class is derived from the name of the managed class by appending "_" to the name of the managed class.
- The metamodel class X_ must be annotated with the javax.persistence.StaticMetamodel annotation
- If class X extends another class S, where S is the most derived managed class (i.e., entity or mapped superclass) extended by X, then class X_ must extend class S_, where S_ is the metamodel class created for S.
- For every persistent non-collection-valued attribute y declared by class X, where the type of y is Y, the metamodel class must contain a declaration as follows:
  
  public static volatile SingularAttribute<X, Y> y;

- For every persistent collection-valued attribute z declared by class X, where the element type of z is Z, the metamodel class must contain a declaration as follows:
  
  - if the collection type of z is java.util.Collection, then
    public static volatile CollectionAttribute<X, Z> z;
  - if the collection type of z is java.util.Set, then
    public static volatile SetAttribute<X, Z> z;
  - if the collection type of z is java.util.List, then
    public static volatile ListAttribute<X, Z> z;
  - if the collection type of z is java.util.Map, then
    public static volatile MapAttribute<X, K, Z> z;
    where K is the type of the key of the map in class X

Let's take an example, for the following class

```java
package org.datanucleus.samples.jpa2.metamodel;

import java.util.*;
import javax.persistence.*;

@Entity
public class Person {
  @Id
  long id;

  String name;

  @OneToMany
  List<Address> addresses;
}
```
the static metamodel class will be

```java
package org.datanucleus.samples.jpa2.metamodel;

import javax.persistence.metamodel.*;

@StaticMetamodel(Person.class)
public class Person_
{
    public static volatile SingularAttribute<Person, Long> id;
    public static volatile SingularAttribute<Person, String> name;
    public static volatile ListAttribute<Person, Address> addresses;
}
```

So how do we generate this metamodel definition for our query classes? DataNucleus provides an annotation processor in the jar `datanucleus-jpa-query` that can be used when compiling your model classes to generate the static metamodel classes. What this does is when the compile is invoked, all classes that have persistence annotations will be passed to the annotation processor and a Java file generated for its metamodel. Then all classes (original + metamodel) are compiled.

To enable this in Maven2 you would need the above jar, plus `datanucleus-core` and `datanucleus-api-jpa` (as well as persistence-api.jar) to be in the CLASSPATH at compile.

```xml
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <source>1.6</source>
        <target>1.6</target>
    </configuration>
</plugin>
```

To enable this in Eclipse you would need to do the following

- Go to Java Compiler and make sure the compiler compliance level is 1.6 or above
- Go to Java Compiler -> Annotation Processing and enable the project specific settings and enable annotation processing
- Go to Java Compiler -> Annotation Processing -> Factory Path, enable the project specific settings and then add the following jars to the list: `datanucleus-jpa-query.jar`, `datanucleus-api-jpa.jar`, `datanucleus-core.jar`, `jpa-api.jar`, `jdo-api.jar`
170 SQL

170.1 JPA : SQL Queries

The JPA specification defines its interpretation of SQL, for selecting objects from the datastore. To provide a simple example, this is what you would do

```java
Query q = em.createNativeQuery("SELECT p.id, o.firstName, o.lastName FROM Person p, Job j " +
  "WHERE (p.job = j.id) AND j.name = 'Cleaner'");
List results = (List)q.getResultsList();
```

This finds all "Person" objects that do the job of "Cleaner". The syntax chosen has to be runnable on the RDBMS that you are using (and since SQL is anything but "standard" you will likely have to change your query when moving to another datastore).

170.1.1 Input Parameters

In JPQL queries it is convenient to pass in parameters so we dont have to define the same query for different values. Here's an example

```java
Numbered Parameters :
Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = ?1 AND p.firstName = ?2");
q.setParameter(1, theSurname);
q.setParameter(2, theForename);
```

So we have parameters that are prefixed by ? (question mark) and are numbered starting at 1. We then use the numbered position when calling `Query.setParameter()`. With SQL queries we can't use named parameters. This is known as numbered parameters.

DataNucleus also supports use of named parameters where you assign names just like in JPQL. This is not defined by the JPA specification so dont expect other JPA implementations to support it. Let's take the previous example and rewrite it using named parameters, like this

```java
Named Parameters :
Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = :firstParam AND p.firstName = :otherParam");
q.setParameter("firstParam", theSurname);
q.setParameter("otherParam", theForename);
```

170.1.2 Range of Results

With SQL you can select the range of results to be returned. For example if you have a web page and you are paginating the results of some search, you may want to get the results from a query in blocks of 20 say, with results 0 to 19 on the first page, then 20 to 39, etc. You can facilitate this as follows

```java
Query q = em.createNativeQuery("SELECT p FROM Person p WHERE p.age &gt; 20");
q.setFirstResult(0);
q.setMaxResults(20);
```
So with this query we get results 0 to 19 inclusive.

170.1.3 Query Execution
There are two ways to execute a SQL query. When you know it will return 0 or 1 results you call

```java
Object result = query.getSingleResult();
```

If however you know that the query will return multiple results, or you just don't know then you would call

```java
List results = query.getResultList();
```

170.2 Named Native Query
With the JPA API you can either define a query at runtime, or define it in the MetaData/annotations for a class and refer to it at runtime using a symbolic name. This second option means that the method of invoking the query at runtime is much simplified. To demonstrate the process, lets say we have a class called `Product` (something to sell in a store). We define the JPA Meta-Data for the class in the normal way, but we also have some query that we know we will require, so we define the following in the Meta-Data.

```xml
<entity class="Product">
  ...
  <named-native-query name="PriceBelowValue"> <![CDATA[
    SELECT NAME FROM PRODUCT WHERE PRICE < ?1
  ]]></named-native-query>
</entity>
```

or using annotations

```java
@Entity
@NamedNativeQuery(name="PriceBelowValue", query="SELECT NAME FROM PRODUCT WHERE PRICE < ?1")
public class Product {...}
```

So here we have an SQL query that will return the names of all Products that have a price less than a specified value. This leaves us the flexibility to specify the value at runtime. So here we run our named query, asking for the names of all Products with price below 20 euros.

```java
Query query = em.createNamedQuery("PriceBelowValue");
query.setParameter(1, new Double(20.0));
List results = query.getResultList();
```
171 Stored Procedures

171.1 JPA: Stored Procedures

The JPA 2.1 specification adds support for calling stored procedures through its API. It allows some flexibility in the type of stored procedure being used, supporting IN/OUT/INOUT parameters as well as result sets being returned. Obviously if a datastore does not support stored procedures then this functionality will not apply.

You start off by creating a stored procedure query, like this, referencing the stored procedure name in the datastore.

```java
StoredProcedureQuery spq = em.createStoredProcedureQuery("PERSON_SP_1");
```

If we have any parameters in this stored procedure we need to register them, for example

```java
spq.registerStoredProcedureParameter("PARAM1", String.class, ParameterMode.IN);
spq.registerStoredProcedureParameter("PARAM2", Integer.class, ParameterMode.OUT);
```

If you have any result class, or result set mapping then you can specify those in the `createStoredProcedureQuery` call. Now we are ready to execute the query and access the results.

171.1.1 Simple execution, returning a result set

A common form of stored procedure will simply return a single result set. You execute such a procedure as follows

```java
List results = spq.getResultList();
```

or if expecting a single result, then

```java
Object result = spq.getSingleResult();
```

171.1.2 Simple execution, returning output parameters

A common form of stored procedure will simply return output parameter(s). You execute such a procedure as follows

```java
spq.execute();
Object paramVal = spq.getOutputParameterValue("PARAM2");
```

or you can also access the output parameters via position (if specified by position).
171.1.3 Generalised execution, for multiple result sets

A more complicated, yet general, form of execution of the stored procedure is as follows

```java
boolean isResultSet = spq.execute(); // returns true when we have a result set from the proc
List results1 = spq.getResultList(); // get the first result set
if (spq.hasMoreResults())
{
    List results2 = spq.getResultList(); // get the second result set
}
```

So the user can get hold of multiple result sets returned by their stored procedure.

171.2 Named Stored Procedure Queries

Just as with normal queries, you can also register a stored procedure query at development time and then access it via name from the EntityManager. So we define one like this (not important on which class it is defined)

```java
@NamedStoredProcedureQuery(name="myTestProc", procedureName="MY_TEST_SP_1",
    parameters={@StoredProcedureParameter(name="PARAM1", type=String.class, mode=ParameterMode.IN)})
@Entity
public class MyClass {...}
```

and then create the query from the EntityManager

```java
StoredProcedureQuery spq = em.createNamedStoredProcedureQuery("myTestProc");
```
172 Guides

172.1 Guides for JPA

The following guides demonstrate the use of JPA using DataNucleus. If you have a guide that you think would be useful in educating users in some concepts of JPA, please contribute it via our website.

- Datastore Replication
- JavaEE Environments
- OSGi Environments
- Security
- Troubleshooting
- Performance Tuning
- Monitoring
- Logging
- Maven with DataNucleus
- Eclipse with DataNucleus
- Tutorial with RDBMS
- Tutorial with ODF
- Tutorial with Excel
- Tutorial with MongoDB
- Tutorial with HBase
- Tutorial with Neo4J
- Eclipse Dali
- JPA Tutorial (TheServerSide)
173 Datastore Replication

173.1 JPA : Datastore Replication

Many applications make use of multiple datastores. It is a common requirement to be able to replicate parts of one datastore in another datastore. Obviously, depending on the datastore, you could make use of the datastores own capabilities for replication. DataNucleus provides its own extension to JPA to allow replication from one datastore to another. This extension doesn’t restrict you to using 2 datastores of the same type. You could replicate from RDBMS to XML for example, or from MySQL to HSQLDB.

You need to make sure you have the persistence property `datanucleus.attachSameDatastore` set to `false` if using replication

Note that the case of replication between two RDBMS of the same type is usually way more efficiently replicated using the capabilities of the datastore itself

The following sample code will replicate all objects of type `Product` and `Employee` from EMF1 to EMF2. These EMFs are created in the normal way so, as mentioned above, EMF1 could be for a MySQL datastore, and EMF2 for XML. By default this will replicate the complete object graphs reachable from these specified types.

```java
import org.datanucleus.api.jpa.JPAReplicationManager;
...

JPAReplicationManager replicator = new JPAReplicationManager(emf1, emf2);
replicator.replicate(new Class[]{Product.class, Employee.class});
```
174 JavaEE Environments

174.1 JPA : Usage of DataNucleus within a JavaEE environment

JPA is designed to allow easy deployment into a JavaEE container. The JavaEE container takes care of integration of the JPA implementation (DataNucleus), so there is no JCA connector required.

Key points to remember when deploying your JPA application to use DataNucleus under JavaEE

- Define a JTA datasource for your persistence operations
- Define a non-JTA datasource for your schema and sequence operations. These are cross-EntityManager and so need their own datasource that is not affected by transactions.

Individual guides for specific JavaEE servers are listed below. If you have a guide for some other server, please notify us via the DataNucleus forum and it will be added to this list.

174.2 JBoss AS7

This guide was provided by Nicolas Seyvet.

JBoss AS7 is the latest JavaEE server from JBoss. Despite searching in multiple locations, I could not find a comprehensive guide on how to switch from the default JBoss Hibernate JPA provider to Datanucleus 3. If you try this guide, please PM the author (or add a comment) and let me know how it worked out. Your feedback will be used to improve this guide. This guide is cross-referenced as part of the JBoss JPA Reference Guide.

174.2.1 Download JBoss AS 7 and DataNucleus 3.2+

- JBoss : At the time I am writing this "How To", the latest JBoss AS available from the main JBoss community site is 7.1.1.Final aka Brontes. In this guide, the latest 7.x SNAPSHOT was used but the steps will work with any JBoss 7.x version.
- DataNucleus : Version 3.2.0 was used, from SourceForge.

174.2.2 Install JBoss AS 7

Install JBoss AS 7 by unzipping the downloaded JBoss zip file in the wanted folder to be used as the JBoss home root folder (example: /local/jboss). From this point, the path where JBoss is unzipped will be referred to as "$JBOSS_HOME".

Note: JBoss AS 7 configuration is controlled by either standalone.xml ( $JBOSS_HOME/standalone/configuration ) or domain.xml ( $JBOSS_HOME/domain/configuration ) depending on the operation mode (standalone or domain) of the application server. The domain mode is typically used for cases where the AS is deployed in a cluster environment. In this tutorial, a single AS instance is used, as such, the standalone mode is selected and all configuration changes will be applied to the "standalone.xml" file.

174.2.2.1 Start JBoss

To start the server, use:
On Linux:

$ cd $JBOSS_HOME/bin/
$ ./standalone.sh

On Windows:

$ cd $JBOSS_HOME/bin/
$ standalone.bat

After a few seconds, a message should indicate the server is started.

17:23:00,251 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015874: JBoss AS 7.2.0.Alpha1-SNAPSHOT "Steropes" started in 3717ms - Started 198 of 257 services (56 services are passive or on-demand)

To verify, access the administration GUI located at http://localhost:9990/, and expect to see a "Welcome to AS 7" banner. On the first start up, a console will show that an admin user must first be created in order to be able to access the management UI. Follow the steps and create a user.

On Linux:

$JBOSS_HOME/bin$ add-user.sh

On Windows:

$JBOSS_HOME/bin$ add-user.bat

174.2.2.2 Add a JDBC DataSource (Optional)
This step is only necessary if an RDBMS solution is used as a data store, or if external drivers are required. This tutorial will use MySQL as the RDBMS storage, and the required drivers and data source will be added. For more information, about data sources under JBoss AS 7, refer to the JBoss docs

174.2.2.3 Add MySQL drivers
For MySQL, it is recommended to use Connector/J, which can be found here. Note that this tutorial uses version 5.1.20. Note: JBoss uses OSGI to define a set of modules, further info about class loading in JBoss. In short, the configuration files binds the services and the modules, defining what is available in the class loader for a specific service or application.

While dropping the drivers in the $JBOSS_HOME/standalone/deployments directory works, this approach is not recommended. The proper approach is to add the drivers by defining a new module containing the required libraries. The full instructions are available under here.

Short walk through for MySQL:

- Get the drivers
- create a "mysql" directory under $JBOSS_HOME/modules/com/
- create a "main" directory under $JBOSS_HOME/modules/com/mysql
• Copy the "mysql-connector-java-5.1.20-bin.jar" drivers under $JBOSS_HOME/modules/com/mysql/main
• Add a "module.xml" file under $JBOSS_HOME/modules/com/mysql/main

```xml
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.0" name="com.mysql">
  <resources>
    <resource-root path="mysql-connector-java-5.1.20-bin.jar"/>
  </resources>
  <dependencies>
    <module name="javax.api"/>
  </dependencies>
</module>
```

The name is important as it defines the module name and is used in the "standalone.xml" configuration file. Now, let's say the URL to the MySQL database to be used is "jdbc:mysql://localhost:3306/simple", there are three ways to add that to the server, either through the management console at localhost or, by modifying the "standalone.xml" configuration file, or by using the Command Line Interface (CLI).

Let's modify the "standalone.xml" file. Verify the AS is stopped. Open "standalone.xml" for editing. Search for "subsystem xmlns="urn:jboss:domain:datasources:1.1", the section defines data sources and driver references. Let's add our data source and drivers. Add the following in the datasources section:
The above defines two data sources (MySQL-DS and MySQL-DS-NonJTA) referring to the same database. The difference between the two is that MySQL-DS has JTA enabled while MySQL-DS-NonJTA does not. This is useful to separate operations during the database automated schema generation phase. Any change to a schema should be made outside the scope of JTA. Many JDBC drivers (for example) will fall apart (assorted type of SQLException) if you try to commit a connection with DDL and SQL mixed, or SQL first then DDL after. Consequently it is recommended to have a separate data source for such operations, hence using the non-jta-data-source.

In the <datasource> section, add:

```xml
<datasource jta="false" jndi-name="java:jdbc/simple-nonjta" pool-name="MySQL-DS-NonJTA" enabled="true">
<connection-url>jdbc:mysql://localhost:3306/simple</connection-url>
<driver>com.mysql</driver>
<transaction-isolation>TRANSACTION_READ_COMMITTED</transaction-isolation>
<security>
  <user-name>[A valid DB user name]</user-name>
  <password>[A valid DB password]</password>
</security>
</datasource>
```

The above defines which drivers to use for the data sources MySQL-DS and MySQL-DS-NonJTA. More info is available as part of the JBoss documentation, refer to the section describing how to setup a new data source.
174.2.3 Add DataNucleus to JBoss

This step adds the DataNucleus libraries as a JBoss module.

- Create a directory to store the DataNucleus libraries, as $JBOSS_HOME/modules/org/datanucleus/main
- Add the following jars from the lib directory of the datanucleus-accessplatform-full-deps ZIP file lib directory: datanucleus-api-jpa-XXX.jar, datanucleus-core-XXX.jar, datanucleus-rdbms-XXX.jar, datanucleus-jpa-query-XXX.jar, and jdo-api-3.1-rc1.jar
- Add a "module.xml" file in the $JBOSS_HOME/modules/org/datanucleus/main directory like this

```xml
<module xmlns="urn:jboss:module:1.1" name="org.datanucleus">
    <dependencies>
        <module name="javax.api"/>
        <module name="javax.persistence.api"/>
        <module name="javax.transaction.api"/>
        <module name="javax.validation.api"/>
    </dependencies>
    <resources>
        <resource-root path="datanucleus-api-jpa-3.2.7.jar"/>
        <resource-root path="datanucleus-core-3.2.11.jar"/>
        <resource-root path="datanucleus-rdbms-3.2.10.jar"/>
        <resource-root path="datanucleus-jpa-query-3.0.3.jar"/>
        <resource-root path="jdo-api-3.1-rc1.jar"/>
    </resources>
</module>
```

At this point, all the JPA dependencies are resolved.

174.2.4 A simple example using DataNucleus JPA and JBoss AS7

Now you simply need to define persistence.xml and use JPA as you normally would. In order to use DataNucleus as a persistence provider, the "persistence.xml" file must contain the "jboss.as.jpa.providerModule" property. Using the datasource defined above, an example of a "persistence.xml" file could be:
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
   <persistence-unit name="[Persistence Unit Name]" transaction-type="JTA">
      <provider>org.datanucleus.api.jpa.PersistenceProviderImpl</provider>
      <!-- MySQL DS -->
      <jta-data-source>java:/jdbc/simple</jta-data-source>
      <non-jta-data-source>java:/jdbc/simple-nonjta</non-jta-data-source>
      <class>[Entities must be listed here]</class>
      <properties>
      <!-- Magic JBoss property for specifying the persistence provider -->
      <property name="jboss.as.jpa.providerModule" value="org.datanucleus"/>

      <!-- following is probably not useful... but it ensures we bind to the JTA transaction manager -->
      <property name="datanucleus.jtaLocator" value="custom_jndi"/>
      <property name="datanucleus.jtaJndiLocation" value="java:/TransactionManager"/>

      <property name="datanucleus.autoCreateSchema" value="true"/>
      <property name="datanucleus.metadata.validate" value="false"/>
      <property name="datanucleus.validateTables" value="false"/>
      <property name="datanucleus.validateConstraints" value="false"/>
   </properties>
</persistence-unit>
</persistence>
175 OSGi Environments

175.1 JPA: Usage of DataNucleus within an OSGi environment

DataNucleus jars are OSGi bundles, and as such, can be deployed in an OSGi environment. Being an OSGi environment care must be taken with respect to class-loading. In particular the persistence property `datanucleus.primaryClassLoader` will need setting. Please refer to the associated guides for JDO to assist you further.

An important thing to note: any dependent jar that is required by DataNucleus needs to be OSGi enabled. By this we mean the jar needs to have the MANIFEST.MF file including `ExportPackage` for the packages required by DataNucleus. Failure to have this will result in `ClassNotFoundException` when trying to load its classes.

Use `jdo-api.jar` v3.0.1 or later since those are OSGi-enabled. Also the Geronimo "jpa" jar that is included in the DataNucleus distribution is OSGi enabled too.

When using DataNucleus in an Eclipse Equinox OSGi environment you can use the persistence property `datanucleus.plugin.pluginRegistryClassName` as `org.datanucleus.plugin.EclipsePluginRegistry` and when in other OSGi environments set that property to `org.datanucleus.plugin.OSGiPluginRegistry`.

175.1.1 JPA and OSGi

In a non OSGi world the persistence provider implementation is loaded using the service provider pattern. The full qualified name of the implementation is stored in a file under `META-INF/services/javax.persistence.spi.PersistenceProvider` (inside the jar of the implementation) and each time the persistence provider is required it gets loaded with a `Class.forName` using the name of the implementing class found inside the `META-INF/services/javax.persistence.spi.PersistenceProvider`.

In the OSGi world that doesn't work. The bundle that needs to load the persistence provider implementation cannot load `META-INF/services/javax.persistence.spi.PersistenceProvider`. A workaround is to copy that file inside each bundle that requires access to the persistence provider. Another workaround is to export the persistence provider as OSGi service. This is what the DataNucleus JPA jar does.

Further reading available on this link

175.1.2 Sample using OSGi and JPA

If you go to DataNucleus sample downloads you will find a sample called `datanucleus-samples-osgi-jpa`. This provides a simple example that you can build and load into such as Apache Karaf to demonstrate JPA persistence. Here we attempt to highlight the key aspects specific to OSGi in this sample.

Model classes are written in the exact same way as you would for any application.

Creation of the EMF is specified in a persistence-unit as normal except that we need to provide two overriding properties

```java
Map<Object, Object> overrideProps = new HashMap();
overrideProps.put("datanucleus.primaryClassLoader", this.getClass().getClassLoader());
overrideProps.put("datanucleus.plugin.pluginRegistryClassName", "org.datanucleus.plugin.OSGiPluginRegistry");
EntityManagerFactory emf = Persistence.createEntityManagerFactory("PU", overrideProps);
```
so we have provided a class loader for the OSGi context of the application, and also specified that we want to use the OSGiPluginRegistry.

All persistence and query operations using EntityManager etc thereafter are identical to what you would use in a normal JSE/JEE application.

The pom.xml also defines the imports/exports for our OSGi application bundle, so look at this if wanting guidance on what these could look like when using Maven and the "felix bundle" plugin.

If you read the file README.txt you can see basic instructions on how to deploy this application into a fresh download of Apache Karaf, and run it. It makes uses of Spring DM to start the JPA "application".

175.2 LocalContainerEntityManagerFactoryBean class for use in Virgo 3.0 OSGi environment

When using DataNucleus 3.x in a Virgo 3.0.x OSGi environment, which is essentially Eclipse Equinox + Spring dm Server with Spring 3.0.5.RELEASE included, the following class is working for me to use in your Spring configuration. You can use this class as a drop-in replacement for Spring's org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean. It was inspired by the code-ish sample at HOWTO Use Datanucleus with OSGi and Spring DM.
import java.util.HashMap;
import java.util.Map;

import javax.persistence.EntityManagerFactory;
import javax.persistence.PersistenceException;
import javax.persistence.spi.PersistenceUnitInfo;

import org.datanucleus.util.StringUtils;
import org.osgi.framework.Bundle;
import org.osgi.framework.BundleContext;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.osgi.context.BundleContextAware;

public class DataNucleusOsgiLocalContainerEntityManagerFactoryBean extends
LocalContainerEntityManagerFactoryBean implements BundleContextAware{

public static final String DEFAULT_JPA_API_BUNDLE_SYMBOLIC_NAME = "org.datanucleus.api.jpa";
public static final String DEFAULT_PERSISTENCE_PROVIDER_CLASS_NAME = "org.datanucleus.api.jpa.PersistenceProviderImpl";

public static final String DEFAULT_OSGI_PLUGIN_REGISTRAR_CLASS_NAME = "org.datanucleus.plugin.OSGiPluginRegistry";
public static final String DEFAULT_OSGI_PLUGIN_REGISTRAR_PROPERTY_NAME = "datanucleus.plugin.pluginRegistryClassName";

protected BundleContext bundleContext;
protected ClassLoader classLoader;
protected String jpaApiBundleSymbolicName = DEFAULT_JPA_API_BUNDLE_SYMBOLIC_NAME;
protected String persistenceProviderClassName = DEFAULT_PERSISTENCE_PROVIDER_CLASS_NAME;
protected String osgiPluginRegistrarClassName = DEFAULT_OSGI_PLUGIN_REGISTRAR_CLASS_NAME;
protected String osgiPluginRegistrarPropertyName = DEFAULT_OSGI_PLUGIN_REGISTRAR_PROPERTY_NAME;

@Override
public void setBundleContext(BundleContext bundleContext) {
this.bundleContext = bundleContext;
}

@Override
protected EntityManagerFactory createNativeEntityManagerFactory() throws PersistenceException {
ClassLoader original = getBeanClassLoader(); // save for later
try {
if (bundleContext != null) {
// default
String name = getPersistenceProviderClassName();
PersistenceUnitInfo info = getPersistenceUnitInfo();
if (info != null && !StringUtils.isEmpty(info.getPersistenceProviderClassName())) {
// use class name of PU
name = info.getPersistenceProviderClassName();
}
if (StringUtils.isEmpty(getJpaApiBundleSymbolicName())) {
throw new IllegalStateException("no DataNucleus JPA API bundle symbolic name given");
}

// set the bean class loader to use it so that Spring can find the persistence provider class
setBeanClassLoader(getBundleClassLoader(getJpaApiBundleSymbolicName(), name));

// since we're in an OSGi environment by virtue of the use of this class, ensure a plugin registration mechanism is being used
if (info == null || (info.getProperties() != null && !info.getProperties().containsKey(getOsgiPluginRegistrarPropertyName()))) {
Map<String, Object> map = getJpaPropertyMap();
map = map == null ? new HashMap<String, Object>() : map;
if (map.get(getOsgiPluginRegistrarPropertyName()) == null) {
map.put(getOsgiPluginRegistrarPropertyName(), getOsgiPluginRegistrarClassName());
}
}
// now let Springy do its thingy
return super.createNativeEntityManagerFactory();
} finally {
setBeanClassLoader(original); // revert bean classloader
}
}

protected ClassLoader getBundleClassLoader(String bundleSymbolicName,String classNameToLoad) {
ClassLoader classloader = null;
Bundle[] bundles = bundleContext.getBundles();
for (int x = 0; x < bundles.length; x++) {
if (bundleSymbolicName.equals(bundles[x].getSymbolicName())) {
try {
classloader = bundles[x].loadClass(classNameToLoad).getClassLoader();
} catch (ClassNotFoundException e) {
throw new IllegalStateException("no DataNucleus JPA API bundle symbolic name given");
}
break;
}
}
return classloader;

public String getJpaApiBundleSymbolicName() {
return jpaApiBundleSymbolicName;
}
public void setJpaApiBundleSymbolicName(String jpaApiBundleSymbolicName) {
this.jpaApiBundleSymbolicName = jpaApiBundleSymbolicName;
}

public String getPersistenceProviderClassName() {
return persistenceProviderClassName;
}
public void setPersistenceProviderClassName(String persistenceProviderClassName) {
this.persistenceProviderClassName = persistenceProviderClassName;
}

public String getOsgiPluginRegistrarClassName() {
return osgiPluginRegistrarClassName;
}
public void setOsgiPluginRegistrarClassName(String osgiPluginRegistrarClassName) {
this.osgiPluginRegistrarClassName = osgiPluginRegistrarClassName;
}

public String getOsgiPluginRegistrarPropertyName() {
return osgiPluginRegistrarPropertyName;
}
public void setOsgiPluginRegistrarPropertyName(String osgiPluginRegistrarPropertyName) {
this.osgiPluginRegistrarPropertyName = osgiPluginRegistrarPropertyName;
}
176 Performance Tuning

176.1 JPA : Performance Tuning

DataNucleus, by default, provides certain functionality. In particular circumstances some of this functionality may not be appropriate and it may be desirable to turn on or off particular features to gain more performance for the application in question. This section contains a few common tips.

176.1.1 Enhancement

You should perform enhancement **before** runtime. That is, do not use `java agent` since it will enhance classes at runtime, when you want responsiveness from your application.

176.1.2 Schema : Creation

DataNucleus provides 4 persistence properties `datanucleus.autoCreateSchema`, `datanucleus.autoCreateTables`, `datanucleus.autoCreateColumns`, and `datanucleus.autoCreateConstraints` that allow creation of the datastore tables. This can cause performance issues at startup. We recommend setting these to `false` at runtime, and instead using `SchemaTool` to generate any required database schema before running DataNucleus (for RDBMS, HBase, etc).

176.1.3 Schema : O/R Mapping

Where you have an inheritance tree it is best to add a **discriminator** to the base class so that it's simple for DataNucleus to determine the class name for a particular row. For RDBMS: this results in cleaner/simpler SQL which is faster to execute, otherwise it would be necessary to do a UNION of all possible tables. For other datastores, a discriminator stores the key information necessary to instantiate the resultant class on retrieval so ought to be more efficient also.

176.1.4 Schema : Validation

DataNucleus provides 3 persistence properties `datanucleus.validateTables`, `datanucleus.validateConstraints`, `datanucleus.validateColumns` that enforce strict validation of the datastore tables against the Meta-Data defined tables. This can cause performance issues at startup. In general this should be run only at schema generation, and should be turned off for production usage. Set all of these properties to `false`. In addition there is a property `datanucleus.rdbms.CheckExistTablesOrViews` which checks whether the tables/views that the classes map onto are present in the datastore. This should be set to `false` if you require fast startup. Finally, the property `datanucleus.rdbms.InitializeColumnInfo` determines whether the default values for columns are loaded from the database. This property should be set to `NONE` to avoid loading database metadata.

To sum up, the optimal settings with schema creation and validation disabled are:
176.1.5 EntityManagerFactory usage

Creation of `EntityManagerFactory` objects can be expensive and should be kept to a minimum. Depending on the structure of your application, use a single factory per datastore wherever possible. Clearly if your application spans multiple servers then this may be impractical, but should be borne in mind.

You can improve startup speed by not specifying all classes in the `persistence-unit` so that they are discovered at runtime. Obviously this may impact on persistence operations later if classes are not known about.

Some RDBMS (such as Oracle) have trouble returning information across multiple catalogs/schemas and so, when DataNucleus starts up and tries to obtain information about the existing tables, it can take some time. This is easily remedied by specifying the catalog/schema name to be used - either for the EMF as a whole (using the persistence properties `datanucleus.Catalog`, `datanucleus.Schema`) or for the package/class using attributes in the MetaData. This subsequently reduces the amount of information that the RDBMS needs to search through and so can give significant speed ups when you have many catalogs/schemas being managed by the RDBMS.

176.1.6 Database Connection Pooling

DataNucleus, by default, will allocate connections when they are required. It then will close the connection. In addition, when it needs to perform something via JDBC (RDBMS datastores) it will allocate a PreparedStatement, and then discard the statement after use. This can be inefficient relative to a database connection and statement pooling facility such as Apache DBCP. With Apache DBCP a Connection is allocated when required and then when it is closed the Connection isn't actually closed but just saved in a pool for the next request that comes in for a Connection. This saves the time taken to establish a Connection and hence can give performance speed ups the order of maybe 30% or more. You can read about how to enable connection pooling with DataNucleus in the Connection Pooling Guide.

As an addendum to the above, you could also turn on caching of PreparedStatements. This can also give a performance boost, depending on your persistence code, the JDBC driver and the SQL being issued. Look at the persistence property `datanucleus.connectionPool.maxStatements`.
176.1.7 EntityManager usage
Clearly the structure of your application will have a major influence on how you utilise an EntityManager. A pattern that gives a clean definition of process is to use a different persistence manager for each request to the data access layer. This reduces the risk of conflicts where one thread performs an operation and this impacts on the successful completion of an operation being performed by another thread. Creation of EM's is not an expensive process and use of multiple threads writing to the same manager should be avoided.

Make sure that you always close the EntityManager after use. It releases all resources connected to it, and failure to do so will result in memory leaks. Also note that when closing the EntityManager if you have the persistence property `datanucleus.detachOnClose` set to `true` (when in an extended PersistenceContext) this will detach all objects in the Level1 cache. Disable this if you don't need these objects to be detached, since it can be expensive when there are many objects.

176.1.8 Persistence Process
To optimise the persistence process for performance you need to analyse what operations are performed and when, to see if there are some features that you could disable to get the persistence you require and omit what is not required. If you think of a typical transaction, the following describes the process

- Start the transaction
- Perform persistence operations. If you are using "optimistic" transactions then all datastore operations will be delayed until commit. Otherwise all datastore operations will default to being performed immediately. If you are handling a very large number of objects in the transaction you would benefit by either disabling "optimistic" transactions, or alternatively setting the persistence property `datanucleus.flush.mode` to `AUTO`, or alternatively, do a manual flush every "n" objects, like this

```java
for (int i=0;i<1000000;i++)
{
    if ((i%10000)/10000 == 0 && i != 0)
    {
        pm.flush();
    }
    ...}
```
- Commit the transaction
  - All dirty objects are flushed.
  - Objects enlisted in the transaction are put in the Level 2 cache. You can disable the level 2 cache with the persistence property `datanucleus.cache.level2.type` set to `none`
  - Objects enlisted in the transaction are detached if you have the persistence property `datanucleus.detachAllOnCommit` set to `true` (when using a transactional PersistenceContext). Disable this if you don't need these objects to be detached at this point

176.1.9 Retrieval of object by identity
If you are retrieving an object by its identity and know that it will be present in the Level2 cache, for example, you can set the persistence property `datanucleus.findObject.validateWhenCached` to `false` and this will skip a separate call to the datastore to validate that the object exists in the datastore.
176.1.10 Identity Generators

DataNucleus provides a series of value generators for generation of identity values. These can have an impact on the performance depending on the choice of generator, and also on the configuration of the generator.

- The max strategy should not really be used for production since it makes a separate DB call for each insertion of an object. Something like the table strategy should be used instead. Better still would be to choose auto and let DataNucleus decide for you.
- The sequence strategy allows configuration of the datastore sequence. The default can be non-optimum. As a guide, you can try setting key-cache-size to 10

The auto identity generator value is the recommended choice since this will allow DataNucleus to decide which identity generator is best for the datastore in use.

176.1.11 Collection/Map caching

DataNucleus has 2 ways of handling calls to SCO Collections/Maps. The original method was to pass all calls through to the datastore. The second method (which is now the default) is to cache the collection/map elements/keys/values. This second method will read the elements/keys/values once only and thereafter use the internally cached values. This second method gives significant performance gains relative to the original method. You can configure the handling of collections/maps as follows :-

- **Globally for the EMF** - this is controlled by setting the persistence property datanucleus.cache.collections. Set it to true for caching the collections (default), and false to pass through to the datastore.

- **For the specific Collection/Map** - this overrides the global setting and is controlled by adding a MetaData <collection> or <map> extension cache. Set it to true to cache the collection data, and false to pass through to the datastore.

The second method also allows a finer degree of control. This allows the use of lazy loading of data, hence elements will only be loaded if they are needed. You can configure this as follows :-

- **Globally for the EMF** - this is controlled by setting the property datanucleus.cache.collections.lazy. Set it to true to use lazy loading, and set it to false to load the elements when the collection/map is initialised.

- **For the specific Collection/Map** - this overrides the global EMF setting and is controlled by adding a MetaData <collection> or <map> extension cache-lazy-loading. Set it to true to use lazy loading, and false to load once at initialisation.

176.1.12 NonTransactional Reads (Reading persistent objects outside a transaction)

Performing non-transactional reads has advantages and disadvantages in performance and data freshness in cache. The objects read are held cached by the EntityManager. The second time an application requests the same objects from the EntityManager they are retrieved from cache. The time spent reading the object from cache is minimum, but the objects may become stale and not represent the database status. If fresh values need to be loaded from the database, then the user application should first call refresh on the object.

Another disadvantage of performing non-transactional reads is that each operation realized opens a new database connection, but it can be minimized with the use of connection pools, and also on some of the datastore the (nontransactional) connection is retained.
176.1.13 Accessing fields of persistent objects when not managed by a EntityManager

Reading fields of unmanaged objects (outside the scope of an EntityManager) is a trivial task, but performed in a certain manner can determine the application performance. The objective here is not give you an absolute response on the subject, but point out the benefits and drawbacks for the many possible solutions.

- Use `datanucleus.RetainValues=true`. This is the default for JPA operation and will ensure that after commit the fields of the object retain their values (rather than being nulled).
- Use `detach` method.

```java
Object copy = null;
try {
    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();

    //retrieve in some way the object, query, find, etc
    Object obj = em.find(MyClass.class, id);
    copy = em.detach(obj);

    em.getTransaction().commit();
} finally {
    em.close();
} //read or change the detached object here
System.out.println(copy.getName());
```

- Use `datanucleus.detachAllOnCommit=true`. Dependent on the persistence context you may automatically have this set.

```java
Object obj = null;
try {
    EntityManager pm = emf.createEntityManager();
    em.getTransaction().begin();

    //retrieve in some way the object, query, find, etc
    obj = em.find(MyClass.class, id);
    em.getTransaction().commit(); // Object "obj" is now detached
} finally {
    em.close();
}
```

The bottom line is to not use detachment if instances will only be used to read values.
176.1.14 Fetch Control
When fetching objects you have control over what gets fetched. This can have an impact if you are then detaching those objects. With JPA the maximum fetch depth is -1 (unlimited). So with JPA you ought to set it to the extent that you want to detach, or better still make use of DataNucleus fetch groups to control the specific fields to detach.

176.1.15 Logging
I/O consumes a huge slice of the total processing time. Therefore it is recommended to reduce or disable logging in production. To disable the logging set the DataNucleus category to OFF in the Log4j configuration. See Logging for more information.

```
log4j.category.DataNucleus=OFF
```

176.2 General Comments on Overall Performance
In most applications, the performance of the persistence layer is very unlikely to be a bottleneck. More likely the design of the datastore itself, and in particular its indices are more likely to have the most impact, or alternatively network latency. That said, it is the DataNucleus projects' committed aim to provide the best performance possible, though we also want to provide functionality, so there is a compromise with respect to resource.

What is a benchmark? This is simply a series of persistence operations performing particular things e.g. persist $n$ objects, or retrieve $n$ objects. If those operations are representative of your application then the benchmark is valid to you.

To find (or create) a benchmark appropriate to your project you need to determine the typical persistence operations that your application will perform. Are you interested in persisting 100 objects at once, or 1 million, for example? Then when you have a benchmark appropriate for that operation, compare the persistence solutions.

The performance tuning guide above gives a good oversight of tuning capabilities, and also refer to the following blog entry for our take on performance of DataNucleus AccessPlatform. And then the later blog entry about how to tune for bulk operations

176.2.1.1 GeeCon JPA provider comparison (Jun 2012)
There is an interesting presentation on JPA provider performance that was presented at GeeCon 2012 by Patrycja Wegzynowicz. This presentation takes the time to look at what operations the persistence provider is performing, and does more than just "persist large number of flat objects into a single table", and so gives you something more interesting to analyse. DataNucleus comes out pretty well in many situations. You can also see the PDF here.

176.2.1.2 PolePosition (Dec 2008)
The PolePosition benchmark is a project on SourceForge to provide a benchmark of the write, read and delete of different data structures using the various persistence tools on the market. JPOX was run against this benchmark just before being renamed as DataNucleus and the results are found in the DataNucleus Wiki. The input data used for that benchmark run is found in JPOX SVN. Some comments on the PolePos benchmark :-
• It is essential that tests for such as Hibernate and DataNucleus performance comparable things. Some of the original tests had the "delete" simply doing a "DELETE FROM TBL." for Hibernate yet doing an Extent followed by delete each object individually for a JDO implementation. This is an unfair comparison and in the source tree in JPOX SVN this is corrected. This fix was pointed out to the PolePos SourceForge project but is not, as yet, fixed.

• It is essential that schema is generated before the test, otherwise the test is no longer a benchmark of just a persistence operation. The source tree in JPOX SVN assumes the schema exists. This fix was pointed out to the PolePos SourceForge project but is not, as yet, fixed.

• Each persistence implementation should have its own tuning options, and be able to add things like discriminators since that is what would happen in a real application. The source tree in JPOX SVN does this for JPOX running. Similarly a JDO implementation would tune the fetch groups being used - this is not present in the SourceForge project but is in JPOX SVN.

• DataNucleus performance is considered to be significantly improved over JPOX particularly due to batched inserts, and due to a rewritten query implementation that does enhanced fetching.
177 Troubleshooting

177.1 JPA : Troubleshooting
This section describes the most common problems found when using DataNucleus in different architectures. It describes symptoms and methods for collecting data for troubleshooting thus reducing time to narrow the problem down and come to a solution.

177.2 Out Of Memory error

177.2.1 Introduction
Java allocate objects in the runtime memory data area called heap. The heap is created on virtual machine start-up. The memory allocated to objects are reclaimed by Garbage Collectors when the object is no longer referenced (See Object References). The heap may be of a fixed size, but can also be expanded when more memory is needed or contracted when no longer needed. If a larger heap is needed and it cannot be allocated an OutOfMemory is thrown. See JVM Specification.

Native memory is used by the JVM to perform its operations like creation of threads, sockets, jdbc drivers using native code, libraries using native code, etc.

The maximum size of heap memory is determined by the -Xmx on the java command line. If Xmx is not set, then the JVM decides for the maximum heap. The heap and native memory are limited to the maximum memory allocated by the JVM. For example, if the JVM Xmx is set to 1GB and currently use of native memory is 256MB then the heap can only use 768MB.

177.2.2 Causes
Common causes of out of memory:

- Not enough heap - The JVM needs more memory to deal with the application requirements. Queries returning more objects than usual can be the cause.
- Not enough PermGen - The JVM needs more memory to load class definitions.
- Memory Leaks - The application does not close the resources, like the EntityManager or Queries, and the JVM cannot reclaim the memory.
- Caching - Caching in the application or inside DataNucleus holding strong references to objects.
- Garbage Collection - If no full garbage collection is performed before the OutOfMemory it can indicate a bug in the JVM Garbage Collector.
- Memory Fragmentation - A large object needs to be placed in the memory, but the JVM cannot allocate a continous space to it because the memory is fragmented.
- JDBC driver - a bug in the JDBC driver not flushing resources or keeping large result sets in memory.

177.2.3 Throubleshooting

177.2.3.1 JVM
Collect garbage collection information by adding -verbosegc to the java command line. The verbosegc flag will print garbage collections to System output.
177.2.3.2 Sun JVM
The Sun JVM 1.4 or upper accepts the flag `-XX:+PrintGCDetails`, which prints detailed information on Garbage Collections. The Sun JVM accepts the flag `-verbose:class`, which prints information about each class loaded. This is useful to troubleshoot issues when OutOfMemory occurs due to lack of space in the PermGen, or when NoClassDefFoundError or Linkage errors occurs. The Sun JVM 1.5 or upper accepts the flag `-XX:+HeapDumpOnOutOfMemoryError`, which creates a hprof binary file head dump in case of an OutOfMemoryError. You can analyse the heap dump using tools such as jhat or YourKit profiler.

177.2.3.3 DataNucleus
DataNucleus keeps in cache persistent objects using weak references by default. Enable debug mode **DataNucleus.Cache** category to investigate the size of the cache in DataNucleus.

177.2.4 Resolution
DataNucleus can be configured to reduce the number of objects in cache. DataNucleus has cache for persistent objects, metadata, datastore metadata, fields of type Collection or Map, or query results.

177.2.4.1 Query Results Cache
The query results hold strong references to the retrieved objects. If a query returns too many objects it can lead to OutOfMemory error. To be able to query over large result sets, change the result set type to **scroll-insensitive** using the persistence property **datanucleus.rdbms.query.resultSetType**.

177.2.4.2 EntityManager leak
It's also a best practice to ensure the EntityManager is closed in a try finally block. The EntityManager has level 1 cache of persistence objects. See the following example:

```java
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
try {
    tx.begin();
    //...
    tx.commit();
} finally {
    if (tx.isActive()) {
        tx.rollback();
    }
    em.close();
}
```

177.2.4.3 Cache for fields of Collection or Map
If collection or map fields have large number of elements, the caching of elements can be disabled with the property **datanucleus.cache.collections** setting it to false.
177.2.4.4 Persistent Objects cache
The cache control of persistent objects is described in the Cache Guide

177.2.4.5 Metadata and Datastore Metadata cache
The metadata and datastore metadata caching cannot be controled by the application, because the memory required for it is insignificant.

177.2.4.6 OutOfMemory when persisting new objects
When persistent many objects, the flush operation should be periodically invoked. This will give a hint to DataNucleus to flush the changes to the database and release the memory. In the below sample the em.flush() operation is invoked on every 10,000 objects persisted.

```java
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
try {
    tx.begin();
    for (int i=0; i<100000; i++)
    {
        Wardrobe wardrobe = new Wardrobe();
        wardrobe.setModel("3 doors");
        pm.makePersistent(wardrobe);
        if (i % 10000 == 0)
        {
            em.flush();
        }
    }
    tx.commit();
} finally {
    if (tx.isActive())
    {
        tx.rollback();
    }
    em.close();
}
```

177.3 Frozen application

177.3.1 Introduction
The application pauses for short or long periods or hangs during very long time.

177.3.2 Causes
Common causes:
- Database Locking - Database waiting other transactions to release locks due to deadlock or locking contentions.
• Garbage Collection Pauses - The garbage collection pauses the application to free memory resources.
• Application Locking - Thread 2 waiting for resources locked by Thread 1.

177.3.3 Troubleshooting

177.3.3.1 Database locking
Use a database specific tool or database scripts to find the current database locks. In Microsoft SQL, the stored procedure `sp_lock` can be used to examine the database locks.

177.3.3.2 Query Timeout
To avoid database locking to hang the application when a query is performed, set the query timeout. See Query Timeout.

177.3.3.3 Garbage Collection pauses
Check if the application freezes when the garbage collection starts. Add `-verbosegc` to the java command line and restart the application.

177.3.3.4 Application Locking
Thread dumps are snapshots of the threads and monitors in the JVM. Thread dumps help to diagnose applications by showing what the application is doing at a certain moment of time. To generate Thread Dumps in MS Windows, press `<ctrl><break>` in the window running the java application. To generate Thread Dumps in Linux/Unix, execute `kill -3 process_id`

To effectively diagnose a problem, take 5 Thread Dumps with 3 to 5 seconds interval between each one. See An Introduction to Java Stack Traces.

177.4 Postgres

177.4.1 ERROR: schema does not exist

177.4.1.1 Problem
Exception `org.postgresql.util.PSQLException: ERROR: schema "PUBLIC" does not exist` raised during transaction.

177.4.1.2 Troubleshooting
• Verify that the schema "PUBLIC" exists. If the name is lowercased ("public"), set `datanucleus.identifier.case=PreserveCase`, since Postgres is case sensitive.
• Via pgAdmin Postgres tool, open a connection to the schema and verify it is accessible with issuing a `SELECT 1` statement.
177.5 Command Line Tools

177.5.1 CreateProcess error=87

177.5.1.1 Problem
CreateProcess error=87 when running DataNucleus tools under Microsoft Windows OS.
Windows has an (antiquated) command line length limitation, between 8K and 64K characters depending on the Windows version, that may be triggered when running tools such as the Enhancer or the SchemaTool with too many arguments.

177.5.1.2 Solution
When running such tools from Maven or Ant, disable the fork mechanism by setting the option fork="false".
178 Monitoring

178.1 JPA : Monitoring

DataNucleus allows a user to enable various MBeans internally. These can then be used for monitoring the number of datastore calls etc.

178.1.1 Via API

The simplest way to monitor DataNucleus is to use its API for monitoring. Internally there are several MBeans (as used by JMX) and you can navigate to these to get the required information. To enable this set the persistence property `datanucleus.enableStatistics` to `true`. There are then two sets of statistics; one for the EMF and one for each EM. You access these as follows.

```java
JPAEntityManagerFactory dnemf = (JPAEntityManagerFactory)emf;
FactoryStatistics stats = dnemf.getNucleusContext().getStatistics();
... (access the statistics information)

JPAEntityManager dnem = (JPAEntityManager)em;
ManagerStatistics stats = dnem.getExecutionContext().getStatistics();
... (access the statistics information)
```

178.1.2 Using JMX

The MBeans used by DataNucleus can be accessed via JMX at runtime. More about JMX here.

An MBean server is bundled with Sun JRE since version 1.5, and you can easily activate DataNucleus MBeans registration by creating your EMF with the persistence property `datanucleus.jmxType` as `default`.

Additionally, setting a few system properties are necessary for configuring the Sun JMX implementation. The minimum properties required are the following:

- `com.sun.management.jmxremote`
- `com.sun.management.jmxremote.authenticate=false`
- `com.sun.management.jmxremote.ssl=false`
- `com.sun.management.jmxremote.port=<port number>`

Usage example:

```java
java -cp TheClassPathInHere
   -Dcom.sun.management.jmxremote
   -Dcom.sun.management.jmxremote.authenticate=false
   -Dcom.sun.management.jmxremote.ssl=false
   -Dcom.sun.management.jmxremote.port=8001
   TheMainClassInHere
```

Once you start your application and DataNucleus is initialized you can browse DataNucleus MBeans using a tool called jconsole (jconsole is distributed with the Sun JDK) via the URL:
Note that the mode of usage is presented in this document as matter of example, and by no means we recommend to disable authentication and secured communication channels. Further details on the Sun JMX implementation and how to configure it properly can be found in here.

DataNucleus MBeans are registered in a MBean Server when DataNucleus is started up (e.g. upon JPA EMF instantiation). To see the full list of DataNucleus MBeans, refer to the javadocs.

To enable management using MX4J you must specify the persistence property `datanucleus.jmxType` as `mx4j` when creating the EMF, and have the `mx4j` and `mx4j-tools` jars in the CLASSPATH.
179 Maven with DataNucleus

179.1 DataNucleus JPA and Maven

Apache Maven is a project management and build tool that is quite common in organisations. Using DataNucleus and JPA with Maven is simple since the DataNucleus jars, JPA API jar, JDO API jar and Maven plugin are present in the Maven central repository, so you don’t need to define any repository to find the artifacts.

The only remaining thing to do is identify which artifacts are required for your project, updating your pom.xml accordingly.

```xml
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>javax.jdo</groupId>
      <artifactId>jdo-api</artifactId>
      <version>3.0.1</version>
    </dependency>
    <dependency>
      <groupId>org.eclipse.persistence</groupId>
      <artifactId>javax.persistence</artifactId>
      <version>2.1.0</version>
    </dependency>
  </dependencies>
  ...
</project>
```

The only distinction to make here is that the above is for compile time since your persistence code (if implementation independent) will only depend on the basic persistence API. At runtime you will need the DataNucleus artifacts present also, so this becomes
<project>
  ...
  <dependencies>
  ...
  <dependency>
    <groupId>javax.jdo</groupId>
    <artifactId>jdo-api</artifactId>
    <version>3.0.1</version>
  </dependency>
  <dependency>
    <groupId>org.eclipse.persistence</groupId>
    <artifactId>javax.persistence</artifactId>
    <version>2.1.0</version>
  </dependency>
  <dependency>
    <groupId>org.datanucleus</groupId>
    <artifactId>datanucleus-core</artifactId>
    <version>[3.2.0, 3.2.99)</version>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>org.datanucleus</groupId>
    <artifactId>datanucleus-api-jpa</artifactId>
    <version>[3.3.0-release, 3.3.99)</version>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>org.datanucleus</groupId>
    <artifactId>datanucleus-rdbms</artifactId>
    <version>[3.2.0, 3.2.99)</version>
    <scope>runtime</scope>
  </dependency>
  ...
  </dependencies>
  ...
</project>

Obviously replace the datanucleus-rdbms jar with the jar for whichever datastore you are using. If you are running the Maven "exec" plugin you may not need the "runtime" specifications.

Please note that you can alternatively use the convenience artifact for JPA+RDBMS (when using RDBMS).
<project>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>org.datanucleus</groupId>
      <artifactId>datanucleus-accessplatform-jpa-rdbms</artifactId>
      <version>3.3.0-release</version>
      <type>pom</type>
    </dependency>
    ...
  </dependencies>
  ...
</project>

179.1.1 Maven2 Plugin : Enhancement and SchemaTool

Now that you have the DataNucleus jars available to you, via the repositories, you want to perform DataNucleus operations. The primary operations are enhancement and SchemaTool. If you want to use the DataNucleus Maven plugin for enhancement or SchemaTool add the following to your pom.xml

<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.datanucleus</groupId>
        <artifactId>datanucleus-maven-plugin</artifactId>
        <version>3.3.0-release</version>
        <configuration>
          <api>JPA</api>
          <persistenceUnitName>MyUnit</persistenceUnitName>
          <log4jConfiguration>${basedir}/log4j.properties</log4jConfiguration>
          <verbose>true</verbose>
        </configuration>
        <executions>
          <execution>
            <phase>process-classes</phase>
            <goals>
              <goal>enhance</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>

Note that this plugin step will automatically try to bring in the latest applicable version of datanucleus-core for use by the enhancer. It does this since you don’t need to have datanucleus-core
in your POM for compilation/enhancement. If you want to use an earlier version then you need to add exclusions to the `maven-datanucleus-plugin`.

The `executions` part of that will make enhancement be performed immediately after compile, so automatic. See also the Enhancer docs.

To run the enhancer manually you do

```
mvn datanucleus:enhance
```

**DataNucleus SchemaTool** is achieved similarly, via

```
mvn datanucleus: schema-create
```
180 Eclipse with DataNucleus

180.1 DataNucleus JPA and Eclipse

Eclipse provides a powerful development environment for Java systems. DataNucleus provides its own plugin for use within Eclipse, giving access to many features of DataNucleus from the convenience of your development environment.

- Installation
- General Preferences
- Preferences : Enhancer
- Preferences : SchemaTool
- Enable DataNucleus Support
- Generate persistence.xml
- Run the Enhancer
- Run SchemaTool

180.1.1 Plugin Installation

The DataNucleus plugin requires Eclipse 3.1 or above. To obtain and install the DataNucleus Eclipse plugin select Help -> Software Updates -> Find and Install On the panel that pops up select Search for new features to install Select New Remote Site, and in that new window set the URL as http://www.datanucleus.org/downloads/eclipse-update/ and the name as DataNucleus. Now select the site it has added "DataNucleus", and click "Finish". This will then find the releases of the DataNucleus plugin. Select the latest version of the DataNucleus Eclipse plugin. Eclipse then downloads and installs the plugin. Easy!

180.1.2 Plugin configuration

The DataNucleus Eclipse plugin allows saving of preferences so that you get nice defaults for all subsequent usage. You can set the preferences at two levels :-

- **Globally for the Plugin**: Go to Window -> Preferences -> DataNucleus Eclipse Plugin and see the options below that
- **For a Project**: Go to {your project} -> Properties -> DataNucleus Eclipse Plugin and select "Enable project-specific properties"

180.1.3 Plugin configuration - General

Firstly open the main plugin preferences page, set the API to be used, and configure the libraries needed by DataNucleus. These are in addition to whatever you already have in your projects CLASSPATH, but to run the DataNucleus Enhancer/SchemaTool you will require the following

- jdo-api.jar : since we use the JDO bytecode enhancement contract
- persistence-api.jar (or equivalent, e.g geronimo-specs-jpa)
- datanucleus-core
- datanucleus-api-jpa
- datanucleus-rdbms : for running SchemaTool
- Datastore driver jar (e.g JDBC) : for running SchemaTool
Below this you can set the location of a configuration file for Log4j to use. This is useful when you want to debug the Enhancer/SchemaTool operations.

180.1.4 Plugin configuration - Enhancer

Open the “Enhancer” page. You have the following settings:

- **Input file extensions**: the enhancer accepts input defining the classes to be enhanced. With JPA you will typically just specify the “persistence-unit” and list the classes and mapping files in there. You can alternatively specify the suffixes of files that define what will be enhanced (e.g. “class” for annotated classes, and “xml” for the ORM mapping file defining entities).
- **Verbose**: selecting this means you get much more output from the enhancer.
- **PersistenceUnit**: Name of the persistence unit if enhancing a persistence-unit.
180.1.5 Plugin configuration - SchemaTool

Open the "SchemaTool" page. You have the following settings

- **Input file extensions**: SchemaTool accepts input defining the classes to have their schema generated. As for the enhancer, you can run this from a "persistence-unit"
- **Verbose**: selecting this means you get much more output from SchemaTool
- **PersistenceUnit**: Name of the persistence unit if running SchemaTool on a persistence-unit
- **Datastore details**: You can either specify the location of a properties file defining the location of your datastore, or you supply the driver name, URL, username and password.
180.1.6 Enabling DataNucleus support

First thing to note is that the DataNucleus plugin is for Eclipse "Java project"s only. After having configured the plugin you can now add DataNucleus support on your projects. Simply right-click on your project in Package Explorer and select DataNucleus->"Add DataNucleus Support" from the context menu.
180.1.7 Defining 'persistence.xml'

You can also use the DataNucleus plugin to generate a "persistence.xml" file adding all classes into a single persistence-unit. You do this by right-clicking on your project, and selecting the option. The "persistence.xml" is generated under META-INF for the source folder. Please note that the wizard will overwrite existing files without further notice.

180.1.8 Enhancing the classes

The DataNucleus Eclipse plugin allows you to easily byte-code enhance your classes using the DataNucleus enhancer. Right-click on your project and select "Enable Auto-Enhancement" from the DataNucleus context menu. Now that you have the enhancer set up you can enable enhancement of your classes. The DataNucleus Eclipse plugin currently works by enabling/disabling automatic enhancement as a follow on process for the Eclipse build step. This means that when you enable it, every time Eclipse builds your classes it will then enhance the classes defined by the available mapping files or what is annotated. Thereafter every time that you build your classes the JPA enabled ones will be enhanced. Easy! Messages from the enhancement process will be written to the Eclipse Console. Make sure that you have your Java files in a source folder, and that the binary class files are written elsewhere If everything is set-up right, you should see the output below.
180.1.9 Generating your database schema

Once your classes have been enhanced you are in a position to create the database schema (assuming you will be using a new schema - omit this step if you already have your schema). Click on the project under "Package Explorer" and under "DataNucleus" there is an option "Run SchemaTool". This brings up a panel to define your database location (URL, login, password etc). You enter these details and the schema will be generated.

Messages from the SchemaTool process will be written to the Eclipse Console.
181 Tutorial with RDBMS

181.1 DataNucleus - Tutorial for JPA for RDBMS

181.1.1 Background

An application can be JPA-enabled via many routes depending on the development process of the project in question. For example the project could use Eclipse as the IDE for developing classes. In that case the project would typically use the Dali Eclipse plugin coupled with the DataNucleus Eclipse plugin. Alternatively the project could use Ant, Maven2 or some other build tool. In this case this tutorial should be used as a guiding way for using DataNucleus in the application. The JPA process is quite straightforward.

1. **Prerequisite**: Download DataNucleus AccessPlatform
2. **Step 1**: Define their persistence definition using Meta-Data.
3. **Step 2**: Define the "persistence-unit"
4. **Step 3**: Compile your classes, and instrument them (using the DataNucleus enhancer).
5. **Step 4**: Write your code to persist your objects within the DAO layer.
6. **Step 5**: Run your application.

The tutorial guides you through this. You can obtain the code referenced in this tutorial from SourceForge (one of the files entitled "datanucleus-samples-jpa-tutorial-*").

181.1.2 Step 0: Download DataNucleus AccessPlatform

You can download DataNucleus in many ways, but the simplest is to download the distribution zip appropriate to your datastore (in this case RDBMS). You can do this from SourceForge DataNucleus download page. When you open the zip you will find DataNucleus jars in the *lib* directory, and dependency jars in the *deps* directory.

181.1.3 Step 1: Take your model classes and mark which are persistable

For our tutorial, say we have the following classes representing a store of products for sale.
package org.datanucleus.samples.jpa.tutorial;

public class Inventory
{
    String name = null;
    Set<Product> products = new HashSet();

    public Inventory(String name)
    {
        this.name = name;
    }

    public Set<Product> getProducts() {return products;}
}

package org.datanucleus.samples.jpa.tutorial;

public class Product
{
    long id;
    String name = null;
    String description = null;
    double price = 0.0;

    public Product(String name, String desc, double price)
    {
        this.name = name;
        this.description = desc;
        this.price = price;
    }
}

package org.datanucleus.samples.jpa.tutorial;

public class Book extends Product
{
    String author=null;
    String isbn=null;
    String publisher=null;

    public Book(String name, String desc, double price, String author,
    String isbn, String publisher)
    {
        super(name,desc,price);
        this.author = author;
        this.isbn = isbn;
        this.publisher = publisher;
    }
}
So we have a relationship (Inventory having a set of Products), and inheritance (Product-Book). Now we need to be able to persist objects of all of these types, so we need to define persistence for them. There are many things that you can define when deciding how to persist objects of a type but the essential parts are

- Mark the class as an Entity so it is visible to the persistence mechanism
- Identify which field(s) represent the identity of the object.

So this is what we do now. Note that we could define persistence using XML metadata, annotations. In this tutorial we will use annotations.

```java
package org.datanucleus.samples.jpa.tutorial;

@Entity
public class Inventory
{
    @Id
    String name = null;

    @OneToMany(cascade={CascadeType.PERSIST, CascadeType.MERGE, CascadeType.DETACH})
    Set<Product> products = new HashSet();
    ...
}

package org.datanucleus.samples.jpa.tutorial;

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Product
{
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE)
    long id;
    ...
}

package org.datanucleus.samples.jpa.tutorial;

@Entity
public class Book extends Product
{
    ...
}
```

Note that we mark each class that can be persisted with @Entity and their primary key field(s) with @Id In addition we defined a valueStrategy for Product field id so that it will have its values generated automatically. In this tutorial we are using application identity which means that all objects of these classes will have their identity defined by the primary key field(s). You can read more in application identity when designing your systems persistence.
181.1.4 Step 2: Define the 'persistence-unit'

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted. You do this via a file META-INF/persistence.xml at the root of the CLASSPATH. Like this

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">
    <!-- JPA tutorial "unit" -->
    <persistence-unit name="Tutorial">
        <class>org.datanucleus.samples.jpa.tutorial.Inventory</class>
        <class>org.datanucleus.samples.jpa.tutorial.Product</class>
        <class>org.datanucleus.samples.jpa.tutorial.Book</class>
        <exclude-unlisted-classes/>
        <properties>
            <property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:mem:datanucleus"/>
            <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver"/>
            <property name="javax.persistence.jdbc.user" value="sa"/>
            <property name="javax.persistence.jdbc.password" value=""/>
            <property name="datanucleus.autoCreateSchema" value="true"/>
            <property name="datanucleus.validateTables" value="false"/>
            <property name="datanucleus.validateConstraints" value="false"/>
        </properties>
    </persistence-unit>
</persistence>
```

181.1.5 Step 3: Enhance your classes

DataNucleus relies on the classes that you want to persist be enhanced to implement the interface PersistenceCapable. You could write your classes manually to do this but this would be laborious. Alternatively you can use a post-processing step to compilation that "enhances" your compiled classes, adding on the necessary extra methods to make them PersistenceCapable. There are several ways to do this, most notably at post-compile, or at runtime. We use the post-compile step in this tutorial. DataNucleus JPA provides its own byte-code enhancer for instrumenting/enhancing your classes (in datanucleus-core) and this is included in the DataNucleus AccessPlatform zip file prerequisite.

To understand on how to invoke the enhancer you need to visualise where the various source and metadata files are stored
The first thing to do is compile your domain/model classes. You can do this in any way you wish, but the downloadable JAR provides an Ant task, and a Maven2 project to do this for you.

Using Ant:
```bash
ant compile
```

Using Maven:
```bash
mvn compile
```

To enhance classes using the DataNucleus Enhancer, you need to invoke a command something like this from the root of your project.

Using Ant:
```bash
ant enhance
```

Using Maven: (this is usually done automatically after the "compile" goal)
```bash
mvn datanucleus:enhance
```

Manually on Linux/Unix:
```bash
org.datanucleus.enhancer.DataNucleusEnhancer
-api JPA -pu Tutorial
```

Manually on Windows:
```bash
java -cp target\classes;lib\datanucleus-core.jar;lib\datanucleus-api-jpa.jar;lib\persistence-api.jar;lib\jdo-api.jar
org.datanucleus.enhancer.DataNucleusEnhancer
-api JPA -pu Tutorial
```

This command enhances all classes defined in the persistence-unit "Tutorial". If you accidentally omitted this step, at the point of running your application and trying to persist an object, you would get a `ClassNotFoundException` thrown. The use of the enhancer is documented in more
detail in the Enhancer Guide. The output of this step are a set of class files that represent persistable classes.

181.1.6 Step 4: Write the code to persist objects of your classes

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted, and when. Interaction with the persistence framework of JPA is performed via an EntityManager. This provides methods for persisting of objects, removal of objects, querying for persisted objects, etc. This section gives examples of typical scenarios encountered in an application.

The initial step is to obtain access to an EntityManager, which you do as follows:

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("Tutorial");
EntityManager em = emf.createEntityManager();
```

So we created an EntityManagerFactory for our "persistence-unit" called "Tutorial" which we defined above. Now that the application has an EntityManager it can persist objects. This is performed as follows:

```java
Transaction tx = em.getTransaction();
try {
    tx.begin();
    Inventory inv = new Inventory("My Inventory");
    Product product = new Product("Sony Discman", "A standard discman from Sony", 49.99);
    inv.getProducts().add(product);
    em.persist(inv);
    tx.commit();
} finally {
    if (tx.isActive()) {
        tx.rollback();
    }
    em.close();
}
```

Please note that the finally step is important in that it tidies up connections to the datastore and the EntityManager.

Now we want to retrieve some objects from persistent storage, so we will use a "Query". In our case we want access to all Product objects that have a price below 150.00 and ordering them in ascending order.
If you want to delete an object from persistence, you would perform an operation something like

```java
Transaction tx = em.getTransaction();
try {
    tx.begin();

    Query q = pm.createQuery("DELETE FROM Person p WHERE p.lastName = 'Jones'");
    int numberInstancesDeleted = q.executeUpdate();

    tx.commit();
} finally {
    if (tx.isActive()) {
        tx.rollback();
    }
}
```

Clearly you can perform a large range of operations on objects. We can't hope to show all of these here. Any good JPA book will provide many examples.
181.1.7 Step 5: Run your application

To run your JPA-enabled application will require a few things to be available in the Java CLASSPATH, these being:

- The "persistence.xml" file (stored under META-INF/)
- Any ORM MetaData files for your persistable classes
- Any JDBC driver classes needed for accessing your datastore
- The JDO API JAR (defining the JDO bytecode enhancement contract)
- The JPA API JAR (defining the JPA interface)
- The DataNucleus Core, DataNucleus JPA API and DataNucleus RDBMS JARs

After that it is simply a question of starting your application and all should be taken care of. You can access the DataNucleus Log file by specifying the `logging` configuration properties, and any messages from DataNucleus will be output in the normal way. The DataNucleus log is a very powerful way of finding problems since it can list all SQL actually sent to the datastore as well as many other parts of the persistence process.

Using Ant (you need the included persistence.xml to specify your database)

```
ant run
```

Using Maven:

```
mvn exec:java
```

Manually on Linux/Unix:

```
```

Manually on Windows:

```
java -cp lib\persistence-api.jar;lib\jdo-api.jar;lib\datanucleus-core.jar;lib\datanucleus-rdbms.jar;lib\datanucleus-api-jpa.jar;lib\{jdbc-driver}.jar;target\classes\; . org.datanucleus.samples.jpa.tutorial.Main
```

Output:

```
DataNucleus Tutorial with JPA
--------------------------
Persisting products
Product and Book have been persisted

Executing Query for Products with price below 150.00
> Book : JRR Tolkien - Lord of the Rings by Tolkien

Deleting all products from persistence

End of Tutorial
```
181.2 Part 2 : Next steps

In the above simple tutorial we showed how to employ JPA and persist objects to an RDBMS. Obviously this just scratches the surface of what you can do, and to use JPA requires minimal work from the user. In this second part we show some further things that you are likely to want to do.

1. Step 6 : Controlling the schema.
2. Step 7 : Generate the database tables where your classes are to be persisted using SchemaTool.

181.2.1 Step 6 : Controlling the schema

In the above simple tutorial we didn't look at controlling the schema generated for these classes. Now let's pay more attention to this part by defining XML Metadata for the schema. We define this in XML to separate schema information from persistence information. So we define a file META-INF/orm.xml at the root of the CLASSPATH.
<?xml version="1.0" encoding="UTF-8" ?>
<entity-mappings>
  <description>DataNucleus JPA tutorial</description>
  <package>org.datanucleus.samples.jpa.tutorial</package>
  <entity class="org.datanucleus.samples.jpa.tutorial.Product" name="Product">
    <table name="JPA PRODUCTS"/>
    <attributes>
      <id name="id">
        <generated-value strategy="TABLE"/>
      </id>
      <basic name="name">
        <column name="PRODUCT_NAME" length="100"/>
      </basic>
      <basic name="description">
        <column length="255"/>
      </basic>
    </attributes>
  </entity>
  <entity class="org.datanucleus.samples.jpa.tutorial.Book" name="Book">
    <table name="JPA_BOOKS"/>
    <attributes>
      <basic name="isbn">
        <column name="ISBN" length="20"/>
      </basic>
      <basic name="author">
        <column name="AUTHOR" length="40"/>
      </basic>
      <basic name="publisher">
        <column name="PUBLISHER" length="40"/>
      </basic>
    </attributes>
  </entity>
  <entity class="org.datanucleus.samples.jpa.tutorial.Inventory" name="Inventory">
    <table name="JPA_INVENTORY"/>
    <attributes>
      <id name="name">
        <column name="NAME" length="40"/>
      </id>
      <one-to-many name="products">
        <join-table name="JPA_INVENTORY_PRODUCTS">
          <join-column name="INVENTORY_ID_OID"/>
          <inverse-join-column name="PRODUCT_ID_EID"/>
        </join-table>
      </one-to-many>
    </attributes>
  </entity>
</entity-mappings>
181.2.2 Step 7: Generate any schema required for your domain classes

This step is optional, depending on whether you have an existing database schema. If you haven't, at this point you can use the DataNucleus SchemaTool to generate the tables where these domain objects will be persisted. DataNucleus RDBMS SchemaTool is a command line utility (it can be invoked from Maven/Ant in a similar way to how the Enhancer is invoked). The first thing that you need is to update the src/main/resources/META-INF/persistence.xml file with your database details. Here we have a sample file (for HSQLDB) that contains

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">
  <!-- Tutorial "unit" -->
  <persistence-unit name="Tutorial">
    <class>org.datanucleus.samples.jpa.tutorial.Inventory</class>
    <class>org.datanucleus.samples.jpa.tutorial.Product</class>
    <class>org.datanucleus.samples.jpa.tutorial.Book</class>
    <exclude-unlisted-classes/>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:mem:datanucleus"/>
      <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver"/>
      <property name="javax.persistence.jdbc.user" value="sa"/>
      <property name="javax.persistence.jdbc.password" value=""/>
      <property name="datanucleus.autoCreateSchema" value="true"/>
      <property name="datanucleus.validateTables" value="false"/>
      <property name="datanucleus.validateConstraints" value="false"/>
    </properties>
  </persistence-unit>
</persistence>
```

Now we need to run DataNucleus RDBMS SchemaTool. For our case above you would do something like this
Using Ant:

ant createschema

Using Maven:

mvn datanucleus:schema-create

Manually on Linux/Unix:

```java
java -cp target/classes:lib/jdo-api.jar:lib/persistence-api.jar:lib/datanucleus-core.jar:
    lib/datanucleus-rdbms.jar:lib/datanucleus-api-jpa.jar:lib/{jdbc_driver.jar}
org.datanucleus.store.schema.SchemaTool
-create -api JPA -pu Tutorial
```

Manually on Windows:

```java
java -cp target\classes;lib\jdo-api.jar;lib\persistence-api.jar;lib\datanucleus-core.jar;
    lib\datanucleus-rdbms.jar;lib\datanucleus-api-jpa.jar;lib\{jdbc_driver.jar}
org.datanucleus.store.schema.SchemaTool
-create -api JPA -pu Tutorial
```

[Command shown on many lines to aid reading. Should be on single line]

This will generate the required tables, indexes, and foreign keys for the classes defined in the annotations and `orm.xml` Meta-Data file.

### 181.2.3 Any questions?

If you have any questions about this tutorial and how to develop applications for use with DataNucleus please read the online documentation since answers are to be found there. If you don't find what you're looking for go to our Forums.

**The DataNucleus Team**
182 Tutorial with ODF

182.1 DataNucleus - Tutorial for JPA for ODF

182.1.1 Background

An application can be JPA-enabled via many routes depending on the development process of the project in question. For example the project could use Eclipse as the IDE for developing classes. In that case the project would typically use the Dali Eclipse plugin coupled with the DataNucleus Eclipse plugin. Alternatively the project could use Ant, Maven2 or some other build tool. In this case this tutorial should be used as a guiding way for using DataNucleus in the application. The JPA process is quite straightforward.

1. Prerequisite: Download DataNucleus AccessPlatform
2. Step 1: Define their persistence definition using Meta-Data.
3. Step 2: Define the "persistence-unit"
4. Step 3: Compile your classes, and instrument them (using the DataNucleus enhancer).
5. Step 4: Write your code to persist your objects within the DAO layer.
6. Step 5: Run your application.

The tutorial guides you through this. You can obtain the code referenced in this tutorial from SourceForge (one of the files entitled "datanucleus-samples-jpa-tutorial-*").

182.1.2 Step 0: Download DataNucleus AccessPlatform

You can download DataNucleus in many ways, but the simplest is to download the distribution zip appropriate to your datastore (in this case ODF). You can do this from SourceForge DataNucleus download page. When you open the zip you will find DataNucleus jars in the lib directory, and dependency jars in the deps directory.

182.1.3 Step 1: Take your model classes and mark which are persistable

For our tutorial, say we have the following classes representing a store of products for sale.
package org.datanucleus.samples.jpa.tutorial;

public class Inventory
{
    String name = null;
    Set<Product> products = new HashSet();

    public Inventory(String name)
    {
        this.name = name;
    }

    public Set<Product> getProducts() {return products;}
}

package org.datanucleus.samples.jpa.tutorial;

public class Product
{
    long id;
    String name = null;
    String description = null;
    double price = 0.0;

    public Product(String name, String desc, double price)
    {
        this.name = name;
        this.description = desc;
        this.price = price;
    }
}

package org.datanucleus.samples.jpa.tutorial;

public class Book extends Product
{
    String author=null;
    String isbn=null;
    String publisher=null;

    public Book(String name, String desc, double price, String author,
        String isbn, String publisher)
    {
        super(name,desc,price);
        this.author = author;
        this.isbn = isbn;
        this.publisher = publisher;
    }
}
So we have a relationship (Inventory having a set of Products), and inheritance (Product-Book). Now we need to be able to persist objects of all of these types, so we need to define persistence for them. There are many things that you can define when deciding how to persist objects of a type but the essential parts are

- Mark the class as an Entity so it is visible to the persistence mechanism
- Identify which field(s) represent the identity of the object.

So this is what we do now. Note that we could define persistence using XML metadata, annotations. In this tutorial we will use annotations.

```java
package org.datanucleus.samples.jpa.tutorial;

@Entity
public class Inventory {
    @Id
    String name = null;

    @OneToMany(cascade={CascadeType.PERSIST, CascadeType.MERGE, CascadeType.DETACH})
    Set<Product> products = new HashSet();
    ...
}
```

```java
package org.datanucleus.samples.jpa.tutorial;

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Product {
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE)
    long id;
    ...
}
```

```java
package org.datanucleus.samples.jpa.tutorial;

@Entity
public class Book extends Product {
    ...
}
```

Note that we mark each class that can be persisted with @Entity and their primary key field(s) with @Id In addition we defined a valueStrategy for Product field id so that it will have its values generated automatically. In this tutorial we are using application identity which means that all objects of these classes will have their identity defined by the primary key field(s). You can read more in application identity when designing your systems persistence.
182.1.4 Step 2: Define the 'persistence-unit'

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted. You do this via a file META-INF/persistence.xml at the root of the CLASSPATH. Like this

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">
    <!-- JPA tutorial "unit" -->
    <persistence-unit name="Tutorial">
        <class>org.datanucleus.samples.jpa.tutorial.Inventory</class>
        <class>org.datanucleus.samples.jpa.tutorial.Product</class>
        <class>org.datanucleus.samples.jpa.tutorial.Book</class>
        <exclude-unlisted-classes/>
        <properties>
            <property name="javax.persistence.jdbc.url" value="odf:file:tutorial.ods"/>
            <property name="datanucleus.autoCreateSchema" value="true"/>
            <property name="datanucleus.validateTables" value="false"/>
            <property name="datanucleus.validateConstraints" value="false"/>
        </properties>
    </persistence-unit>
</persistence>
```

182.1.5 Step 3: Enhance your classes

DataNucleus relies on the classes that you want to persist be enhanced to implement the interface PersistenceCapable. You could write your classes manually to do this but this would be laborious. Alternatively you can use a post-processing step to compilation that "enhances" your compiled classes, adding on the necessary extra methods to make them PersistenceCapable. There are several ways to do this, most notably at post-compile, or at runtime. We use the post-compile step in this tutorial. DataNucleus JPA provides its own byte-code enhancer for instrumenting/enhancing your classes (in datanucleus-core) and this is included in the DataNucleus AccessPlatform zip file prerequisite.

To understand on how to invoke the enhancer you need to visualise where the various source and jdo files are stored.
The first thing to do is compile your domain/model classes. You can do this in any way you wish, but the downloadable JAR provides an Ant task, and a Maven2 project to do this for you.

<table>
<thead>
<tr>
<th>Using Ant</th>
<th>ant compile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using Maven</td>
<td>maven compile</td>
</tr>
</tbody>
</table>

To enhance classes using the DataNucleus Enhancer, you need to invoke a command something like this from the root of your project.

<table>
<thead>
<tr>
<th>Using Ant</th>
<th>ant enhance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using Maven</td>
<td>(this is usually done automatically after the &quot;compile&quot; goal) maven datanucleus:enhance</td>
</tr>
</tbody>
</table>

Manually on Linux/Unix:
```
java -cp target/classes:lib/datanucleus-core.jar:
   lib/datanucleus-api-jpa.jar:lib/persistence-api.jar:lib/jdo-api.jar
org.datanucleus.enhancer.DataNucleusEnhancer
-api JPA -pu Tutorial
```

Manually on Windows:
```
java -cp target\classes;lib\datanucleus-core.jar;
   lib\datanucleus-api-jpa.jar;lib\persistence-api.jar;lib\jdo-api.jar
org.datanucleus.enhancer.DataNucleusEnhancer
-api JPA -pu Tutorial
```

This command enhances all class files specified in the persistence-unit "Tutorial". If you accidentally omitted this step, at the point of running your application and trying to persist an object, you would get a `ClassNotFoundException` thrown. The use of the enhancer is documented in more
detail in the Enhancer Guide. The output of this step are a set of class files that represent persistable classes.

182.1.6 Step 4: Write the code to persist objects of your classes
Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted, and when. Interaction with the persistence framework of JPA is performed via an EntityManager. This provides methods for persisting of objects, removal of objects, querying for persisted objects, etc. This section gives examples of typical scenarios encountered in an application.

The initial step is to obtain access to an EntityManager, which you do as follows

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("Tutorial");
EntityManager em = emf.createEntityManager();
```

So we created an EntityManagerFactory for our "persistence-unit" called "Tutorial" and an EntityManager. Now that the application has an EntityManager it can persist objects. This is performed as follows

```java
Transaction tx = em.getTransaction();
try{
    tx.begin();
    Inventory inv = new Inventory("My Inventory");
    Product product = new Product("Sony Discman", "A standard discman from Sony", 49.99);
    inv.getProducts().add(product);
    em.persist(inv);
    tx.commit();
}finally{
    if (tx.isActive()){
        tx.rollback();
    }
    em.close();
}
```

Please note that the finally step is important in that it tidies up connections to the datastore and the EntityManager.

Now we want to retrieve some objects from persistent storage, so we will use a "Query". In our case we want access to all Product objects that have a price below 150.00 and ordering them in ascending order.
If you want to delete an object from persistence, you would perform an operation something like

```java
Transaction tx = em.getTransaction();
try {
    tx.begin();
    // Find and delete all objects whose last name is 'Jones'
    Query q = em.createQuery("DELETE FROM Person p WHERE p.lastName = 'Jones'"),
    int numberInstancesDeleted = q.executeUpdate();
    tx.commit();
} finally {
    if (tx.isActive())
    { 
        tx.rollback();
    }
    em.close();
}
```

Clearly you can perform a large range of operations on objects. We can’t hope to show all of these here. Any good JPA book will provide many examples.
182.1.7 Step 5 : Run your application

To run your JPA-enabled application will require a few things to be available in the Java CLASSPATH, these being:

- The "persistence.xml" file (stored under META-INF/)
- Any ORM MetaData files for your persistable classes
- ODFDOM jar needed for accessing your datastore
- The JDO API JAR (defining the JDO bytecode enhancement contract)
- The JPA API JAR (defining the JPA interface)
- The DataNucleus Core, DataNucleus JPA API and DataNucleus ODF JARs

After that it is simply a question of starting your application and all should be taken care of. You can access the DataNucleus Log file by specifying the logging configuration properties, and any messages from DataNucleus will be output in the normal way. The DataNucleus log is a very powerful way of finding problems since it can list all SQL actually sent to the datastore as well as many other parts of the persistence process.

Using Ant (you need the included persistence.xml to specify your database)

ant run

Using Maven:

mvn exec:java

Manually on Linux/Unix :

dump -cp lib/persistence-api.jar;lib/jdo-api.jar;lib/datanucleus-core.jar;lib/datanucleus-odf.jar;lib/datanucleus-api-jpa.jar;lib/odfdom.jar;target/classes:/.
d.org.datanucleus.samples.jpa.tutorial.Main

Manually on Windows :

dump -cp lib\persistence-api.jar;lib\jdo-api.jar;lib\datanucleus-core.jar;lib\datanucleus-odf.jar;lib\datanucleus-api-jpa.jar;lib\odfdom.jar;target\classes;.
d.org.datanucleus.samples.jpa.tutorial.Main

Output :

DataNucleus Tutorial with JPA
---------------------------------
Persisting products
Product and Book have been persisted

Executing Query for Products with price below 150.00
> Book : JRR Tolkien - Lord of the Rings by Tolkien

Deleting all products from persistence

End of Tutorial
182.2 Part 2 : Next steps

In the above simple tutorial we showed how to employ JPA and persist objects to an ODF spreadsheet. Obviously this just scratches the surface of what you can do, and to use JPA requires minimal work from the user. In this second part we show some further things that you are likely to want to do.

1. Step 6 : Controlling the schema.
2. Step 7 : Generate the database tables where your classes are to be persisted using SchemaTool.

182.2.1 Step 6 : Controlling the schema

In the above simple tutorial we didn't look at controlling the schema generated for these classes. Now let's pay more attention to this part by defining XML Metadata for the schema. We define this in XML to separate schema information from persistence information. So we define a file META-INF/orm.xml
<?xml version="1.0" encoding="UTF-8"  ?>
<entity-mappings>
  <description>DataNucleus JPA tutorial</description>
  <package>org.datanucleus.samples.jpa.tutorial</package>
  <entity class="org.datanucleus.samples.jpa.tutorial.Product" name="Product">
    <table name="JPA_PRODUCTS"/>
    <attributes>
      <id name="id">
        <generated-value strategy="TABLE"/>
      </id>
      <basic name="name">
        <column name="PRODUCT_NAME" length="100"/>
      </basic>
      <basic name="description">
        <column length="255"/>
      </basic>
    </attributes>
  </entity>

  <entity class="org.datanucleus.samples.jpa.tutorial.Book" name="Book">
    <table name="JPA_BOOKS"/>
    <attributes>
      <basic name="isbn">
        <column name="ISBN" length="20"/>
      </basic>
      <basic name="author">
        <column name="AUTHOR" length="40"/>
      </basic>
      <basic name="publisher">
        <column name="PUBLISHER" length="40"/>
      </basic>
    </attributes>
  </entity>

  <entity class="org.datanucleus.samples.jpa.tutorial.Inventory" name="Inventory">
    <table name="JPA_INVENTORY"/>
    <attributes>
      <id name="name">
        <column name="NAME" length="40"/>
      </id>
      <one-to-many name="products">
        <join-table name="JPA_INVENTORY_PRODUCTS">
          <join-column name="INVENTORY_ID_OID"/>
          <inverse-join-column name="PRODUCT_ID_EID"/>
        </join-table>
      </one-to-many>
    </attributes>
  </entity>
</entity-mappings>

This file should be placed at the root of the CLASSPATH under META-INF.
182.2.2 Step 7: Generate any schema required for your domain classes

This step is optional, depending on whether you have an existing database schema. If you haven’t, at this point you can use the DataNucleus SchemaTool to generate the tables where these domain objects will be persisted. DataNucleus SchemaTool is a command line utility (it can be invoked from Maven/Ant in a similar way to how the Enhancer is invoked). The first thing that you need is to update the src/java/META-INF/persistence.xml file with your database details. Here we have a sample file (for HSQLDB) that contains

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">
   <!-- Tutorial "unit" -->
   <persistence-unit name="Tutorial">
     <class>org.datanucleus.samples.jpa.tutorial.Inventory</class>
     <class>org.datanucleus.samples.jpa.tutorial.Product</class>
     <class>org.datanucleus.samples.jpa.tutorial.Book</class>
     <exclude-unlisted-classes/>
     <properties>
       <property name="javax.persistence.jdbc.url" value="odf:file:tutorial.ods"/>
       <property name="datanucleus.autoCreateSchema" value="true"/>
       <property name="datanucleus.validateTables" value="false"/>
       <property name="datanucleus.validateConstraints" value="false"/>
     </properties>
   </persistence-unit>
</persistence>
```

Now we need to run DataNucleus SchemaTool. For our case above you would do something like this
Using Ant:
ant createschema

Using Maven:
mvn datanucleus:schema-create

Manually on Linux/Unix:
java -cp target/classes:lib/jdo-api.jar:lib/persistence-api.jar:lib/datanucleus-core.jar:
   lib/datanucleus-odf.jar:lib/datanucleus-api-jpa.jar:lib/odfdom.jar
   org.datanucleus.store.schema.SchemaTool
   -create -api JPA -pu Tutorial

Manually on Windows:
java -cp target\classes;lib\jdo-api.jar;lib\persistence-api.jar;lib\datanucleus-core.jar;
   lib\datanucleus-odf.jar;lib\datanucleus-api-jpa.jar;lib\odfdom.jar
   org.datanucleus.store.schema.SchemaTool
   -create -api JPA -pu Tutorial

[Command shown on many lines to aid reading. Should be on single line]

This will generate the required tables, indexes, and foreign keys for the classes defined in the annotations and *orm.xml* Meta-Data file.

182.2.3 Any questions?

If you have any questions about this tutorial and how to develop applications for use with **DataNucleus** please read the online documentation since answers are to be found there. If you don't find what you're looking for go to our [Forums].

**The DataNucleus Team**
183 Tutorial with Excel

183.1 DataNucleus - Tutorial for JPA for Excel

183.1.1 Background
An application can be JPA-enabled via many routes depending on the development process of the project in question. For example the project could use Eclipse as the IDE for developing classes. In that case the project would typically use the Dali Eclipse plugin coupled with the DataNucleus Eclipse plugin. Alternatively the project could use Ant, Maven2 or some other build tool. In this case this tutorial should be used as a guiding way for using DataNucleus in the application. The JPA process is quite straightforward.

1. **Prerequisite**: Download DataNucleus AccessPlatform
2. **Step 1**: Define their persistence definition using Meta-Data.
3. **Step 2**: Define the "persistence-unit"
4. **Step 3**: Compile your classes, and instrument them (using the DataNucleus enhancer).
5. **Step 4**: Write your code to persist your objects within the DAO layer.
6. **Step 5**: Run your application.

The tutorial guides you through this. You can obtain the code referenced in this tutorial from SourceForge (one of the files entitled "datanucleus-samples-jpa-tutorial-*").

183.1.2 Step 0: Download DataNucleus AccessPlatform
You can download DataNucleus in many ways, but the simplest is to download the distribution zip appropriate to your datastore (in this case Excel). You can do this from SourceForge DataNucleus download page. When you open the zip you will find DataNucleus jars in the `lib` directory, and dependency jars in the `deps` directory.

183.1.3 Step 1: Take your model classes and mark which are persistable
For our tutorial, say we have the following classes representing a store of products for sale.
package org.datanucleus.samples.jpa.tutorial;

public class Inventory {
    String name = null;
    Set<Product> products = new HashSet();

    public Inventory(String name) {
        this.name = name;
    }

    public Set<Product> getProducts() { return products; }
}

package org.datanucleus.samples.jpa.tutorial;

public class Product {
    long id;
    String name = null;
    String description = null;
    double price = 0.0;

    public Product(String name, String desc, double price) {
        this.name = name;
        this.description = desc;
        this.price = price;
    }
}

package org.datanucleus.samples.jpa.tutorial;

public class Book extends Product {
    String author=null;
    String isbn=null;
    String publisher=null;

    public Book(String name, String desc, double price, String author,
                String isbn, String publisher) {
        super(name,desc,price);
        this.author = author;
        this.isbn = isbn;
        this.publisher = publisher;
    }
}
So we have a relationship (Inventory having a set of Products), and inheritance (Product-Book). Now we need to be able to persist objects of all of these types, so we need to define persistence for them. There are many things that you can define when deciding how to persist objects of a type but the essential parts are

- Mark the class as an Entity so it is visible to the persistence mechanism
- Identify which field(s) represent the identity of the object.

So this is what we do now. Note that we could define persistence using XML metadata, annotations. In this tutorial we will use annotations.

```java
package org.datanucleus.samples.jpa.tutorial;

@Entity
public class Inventory
{
  @Id
  String name = null;

  @OneToMany(cascade={CascadeType.PERSIST, CascadeType.MERGE, CascadeType.DETACH})
  Set<Product> products = new HashSet();
  ...
}

package org.datanucleus.samples.jpa.tutorial;

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Product
{
  @Id
  @GeneratedValue(strategy=GenerationType.TABLE)
  long id;
  ...
}

package org.datanucleus.samples.jpa.tutorial;

@Entity
public class Book extends Product
{
  ...
}
```

Note that we mark each class that can be persisted with @Entity and their primary key field(s) with @Id In addition we defined a valueStrategy for Product field id so that it will have its values generated automatically. In this tutorial we are using application identity which means that all objects of these classes will have their identity defined by the primary key field(s). You can read more in application identity when designing your systems persistence.
183.1.4 Step 2: Define the 'persistence-unit'

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted. You do this via a file `META-INF/persistence.xml` at the root of the CLASSPATH. Like this

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">

    <!-- JPA tutorial "unit" -->
    <persistence-unit name="Tutorial">
        <class>org.datanucleus.samples.jpa.tutorial.Inventory</class>
        <class>org.datanucleus.samples.jpa.tutorial.Product</class>
        <class>org.datanucleus.samples.jpa.tutorial.Book</class>
        <exclude-unlisted-classes/>
        <properties>
            <property name="javax.persistence.jdbc.url" value="excel:file:tutorial.xls"/>
            <property name="datanucleus.autoCreateSchema" value="true"/>
            <property name="datanucleus.validateTables" value="false"/>
            <property name="datanucleus.validateConstraints" value="false"/>
        </properties>
    </persistence-unit>
</persistence>
```

183.1.5 Step 3: Enhance your classes

DataNucleus relies on the classes that you want to persist be enhanced to implement the interface `PersistenceCapable`. You could write your classes manually to do this but this would be laborious. Alternatively you can use a post-processing step to compilation that "enhances" your compiled classes, adding on the necessary extra methods to make them `PersistenceCapable`. There are several ways to do this, most notably at post-compile, or at runtime. We use the post-compile step in this tutorial. **DataNucleus JPA** provides its own byte-code enhancer for instrumenting/enhancing your classes (in `datanucleus-core`) and this is included in the DataNucleus AccessPlatform zip file prerequisite.

To understand on how to invoke the enhancer you need to visualise where the various source and jdo files are stored
The first thing to do is compile your domain/model classes. You can do this in any way you wish, but the downloadable JAR provides an Ant task, and a Maven2 project to do this for you.

Using Ant:
`ant compile`

Using Maven:
`mvn compile`

To enhance classes using the DataNucleus Enhancer, you need to invoke a command something like this from the root of your project.

Using Ant:
`ant enhance`

Using Maven: (this is usually done automatically after the "compile" goal)
`mvn datanucleus:enhance`

Manually on Linux/Unix:
```
java -cp target/classes:lib/datanucleus-core.jar:
    lib/datanucleus-api-jpa.jar:lib/persistence-api.jar:lib/jdo-api.jar
org.datanucleus.enhancer.DataNucleusEnhancer
-api JPA -pu Tutorial
```

Manually on Windows:
```
java -cp target\classes;lib\datanucleus-core.jar;
    lib\datanucleus-api-jpa.jar;lib\persistence-api.jar;lib\jdo-api.jar
org.datanucleus.enhancer.DataNucleusEnhancer
-api JPA -pu Tutorial
```

[Command shown on many lines to aid reading - should be on single line]
183.1.6 Step 4 : Write the code to persist objects of your classes

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted, and when. Interaction with the persistence framework of JPA is performed via an EntityManager. This provides methods for persisting of objects, removal of objects, querying for persisted objects, etc. This section gives examples of typical scenarios encountered in an application.

The initial step is to obtain access to an EntityManager, which you do as follows:

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("Tutorial");
EntityManager em = emf.createEntityManager();
```

So we created an `EntityManagerFactory` for our "persistence-unit" called "Tutorial". Now that the application has an EntityManager it can persist objects. This is performed as follows:

```java
Transaction tx = em.getTransaction();
try {
    tx.begin();
    Inventory inv = new Inventory("My Inventory");
    Product product = new Product("Sony Discman", "A standard discman from Sony", 49.99);
    inv.getProducts().add(product);
    em.persist(inv);
    tx.commit();
} finally {
    if (tx.isActive()) {
        tx.rollback();
    }
    em.close();
}
```

Please note that the `finally` step is important in that it tidies up connections to the datastore and the EntityManager.

Now we want to retrieve some objects from persistent storage, so we will use a "Query". In our case we want access to all Product objects that have a price below 150.00 and ordering them in ascending order.
Transaction tx = em.getTransaction();
try {
    tx.begin();

    Query q = pm.createQuery("SELECT p FROM Product p WHERE p.price < 150.00");
    List results = q.getResultList();
    Iterator iter = results.iterator();
    while (iter.hasNext())
    {
        Product p = (Product)iter.next();

        ... (use the retrieved object)
    }
    tx.commit();
}
finally {
    if (tx.isActive())
    {
        tx.rollback();
    }
    em.close();
}

If you want to delete an object from persistence, you would perform an operation something like

Transaction tx = em.getTransaction();
try {
    tx.begin();

    // Find and delete all objects whose last name is 'Jones'
    Query q = em.createQuery("DELETE FROM Person p WHERE p.lastName = 'Jones'");
    int numberInstancesDeleted = q.executeUpdate();
    tx.commit();
}
finally{
    if (tx.isActive())
    {
        tx.rollback();
    }
    em.close();
}

Clearly you can perform a large range of operations on objects. We can't hope to show all of these here. Any good JPA book will provide many examples.
183 Tutorial with Excel

183.1.7 Step 5 : Run your application

To run your JPA-enabled application will require a few things to be available in the Java CLASSPATH, these being

- The "persistence.xml" file (stored under META-INF/)
- Any ORM MetaData files for your persistable classes
- Apache POI jar needed for accessing your datastore
- The JDO API JAR (defining the JDO bytecode enhancement contract)
- The JPA API JAR (defining the JPA interface)
- The DataNucleus Core, DataNucleus JPA API and DataNucleus Excel JARs

After that it is simply a question of starting your application and all should be taken care of. You can access the DataNucleus Log file by specifying the logging configuration properties, and any messages from DataNucleus will be output in the normal way. The DataNucleus log is a very powerful way of finding problems since it can list all SQL actually sent to the datastore as well as many other parts of the persistence process.

Using Ant (you need the included persistence.xml to specify your database)

```java
ant run
```

Using Maven:

```java
mvn exec:java
```

Manually on Linux/Unix :

```java
    lib/datanucleus-api-jpa.jar:lib/{poi_jars}:target/classes/:.
    org.datanucleus.samples.jpa.tutorial.Main
```

Manually on Windows :

```java
java -cp lib\persistence-api.jar;lib\jdo-api.jar;lib\datanucleus-core.jar;lib\datanucleus-excel.jar;
    lib\datanucleus-api-jpa.jar;lib\{poi_jars};target\classes\;.
    org.datanucleus.samples.jpa.tutorial.Main
```

Output :

```
DataNucleus Tutorial with JPA
================================
Persisting products
Product and Book have been persisted

Executing Query for Products with price below 150.00
> Book : JRR Tolkien - Lord of the Rings by Tolkien

Deleting all products from persistence

End of Tutorial
```
183.2 Part 2: Next steps

In the above simple tutorial we showed how to employ JPA and persist objects to an Excel spreadsheet. Obviously this just scratches the surface of what you can do, and to use JPA requires minimal work from the user. In this second part we show some further things that you are likely to want to do.

1. **Step 6**: Controlling the schema.
2. **Step 7**: Generate the database tables where your classes are to be persisted using SchemaTool.

183.2.1 Step 6: Controlling the schema

In the above simple tutorial we didn't look at controlling the schema generated for these classes. Now let's pay more attention to this part by defining XML Metadata for the schema. We define this in XML to separate schema information from persistence information. So we define a file `orm.xml`
<?xml version="1.0" encoding="UTF-8" ?>
<entity-mappings>
  <description>DataNucleus JPA tutorial</description>
  <package>org.datanucleus.samples.jpa.tutorial</package>
  <entity class="org.datanucleus.samples.jpa.tutorial.Product" name="Product">
    <table name="JPA_PRODUCTS"/>
    <attributes>
      <id name="id">
        <generated-value strategy="TABLE"/>
      </id>
      <basic name="name">
        <column name="PRODUCT_NAME" length="100"/>
      </basic>
      <basic name="description">
        <column length="255"/>
      </basic>
    </attributes>
  </entity>
  <entity class="org.datanucleus.samples.jpa.tutorial.Book" name="Book">
    <table name="JPA_BOOKS"/>
    <attributes>
      <basic name="isbn">
        <column name="ISBN" length="20"/>
      </basic>
      <basic name="author">
        <column name="AUTHOR" length="40"/>
      </basic>
      <basic name="publisher">
        <column name="PUBLISHER" length="40"/>
      </basic>
    </attributes>
  </entity>
  <entity class="org.datanucleus.samples.jpa.tutorial.Inventory" name="Inventory">
    <table name="JPA_INVENTORY"/>
    <attributes>
      <id name="name">
        <column name="NAME" length="40"/>
      </id>
      <one-to-many name="products">
        <join-table name="JPA_INVENTORY_PRODUCTS">
          <join-column name="INVENTORY_ID_OID"/>
          <inverse-join-column name="PRODUCT_ID_EID"/>
        </join-table>
      </one-to-many>
    </attributes>
  </entity>
</entity-mappings>

This file should be placed at the root of the CLASSPATH under META-INF.
183.2.2 Step 7: Generate any schema required for your domain classes

This step is optional, depending on whether you have an existing database schema. If you haven’t, at this point you can use the DataNucleus SchemaTool to generate the tables where these domain objects will be persisted. DataNucleus SchemaTool is a command line utility (it can be invoked from Maven/Ant in a similar way to how the Enhancer is invoked). The first thing that you need is to update the `src/java/META-INF/persistence.xml` file with your database details. Here we have a sample file (for HSQLDB) that contains

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">
    <!-- Tutorial "unit" -->
    <persistence-unit name="Tutorial">
        <class>org.datanucleus.samples.jpa.tutorial.Inventory</class>
        <class>org.datanucleus.samples.jpa.tutorial.Product</class>
        <class>org.datanucleus.samples.jpa.tutorial.Book</class>
        <properties>
            <property name="javax.persistence.jdbc.url" value="excel:file:tutorial.xls"/>
            <property name="datanucleus.autoCreateSchema" value="true"/>
            <property name="datanucleus.validateTables" value="false"/>
            <property name="datanucleus.validateConstraints" value="false"/>
        </properties>
    </persistence-unit>
</persistence>
```

Now we need to run DataNucleus SchemaTool. For our case above you would do something like this
Using Ant:
    ant createschema

Using Maven:
    mvn datanucleus:schema-create

Manually on Linux/Unix:
    java -cp target/classes:lib/jdo-api.jar:lib/persistence-api.jar:lib/datanucleus-core.jar:
        lib/datanucleus-excel.jar:lib/datanucleus-api-jpa.jar:lib/{poi.jar}
        org.datanucleus.store.schema.SchemaTool
    -create -api JPA -pu Tutorial

Manually on Windows:
    java -cp target\classes;lib\jdo-api.jar;lib\persistence-api.jar;lib\datanucleus-core.jar;
        lib\datanucleus-excel.jar;lib\datanucleus-api-jpa.jar;lib\{poi.jar}
        org.datanucleus.store.schema.SchemaTool
    -create -api JPA -pu Tutorial

[Command shown on many lines to aid reading. Should be on single line]

This will generate the required tables, indexes, and foreign keys for the classes defined in the
annotations and orm.xml Meta-Data file.

183.2.3 Any questions?
If you have any questions about this tutorial and how to develop applications for use with
DataNucleus please read the online documentation since answers are to be found there. If you don't
find what you're looking for go to our Forums.

The DataNucleus Team
184 Tutorial with MongoDB

184.1 DataNucleus - Tutorial for JPA for MongoDB

184.1.1 Background
An application can be JPA-enabled via many routes depending on the development process of the project in question. For example the project could use Eclipse as the IDE for developing classes. In that case the project would typically use the Dali Eclipse plugin coupled with the DataNucleus Eclipse plugin. Alternatively the project could use Ant, Maven2 or some other build tool. In this case this tutorial should be used as a guiding way for using DataNucleus in the application. The JPA process is quite straightforward.

1. Prerequisite: Download DataNucleus AccessPlatform
2. Step 1: Define their persistence definition using Meta-Data.
3. Step 2: Define the "persistence-unit"
4. Step 3: Compile your classes, and instrument them (using the DataNucleus enhancer).
5. Step 4: Write your code to persist your objects within the DAO layer.
6. Step 5: Run your application.

The tutorial guides you through this. You can obtain the code referenced in this tutorial from SourceForge (one of the files entitled "datanucleus-samples-jpa-tutorial-*").

184.1.2 Step 0: Download DataNucleus AccessPlatform
You can download DataNucleus in many ways, but the simplest is to download the distribution zip appropriate to your datastore (in this case MongoDB). You can do this from SourceForge DataNucleus download page. When you open the zip you will find DataNucleus jars in the lib directory, and dependency jars in the deps directory.

184.1.3 Step 1: Take your model classes and mark which are persistable
For our tutorial, say we have the following classes representing a store of products for sale.
package org.datanucleus.samples.jpa.tutorial;

public class Inventory
{
    String name = null;
    Set<Product> products = new HashSet();

    public Inventory(String name)
    {
        this.name = name;
    }

    public Set<Product> getProducts() {return products;}
}

package org.datanucleus.samples.jpa.tutorial;

public class Product
{
    long id;
    String name = null;
    String description = null;
    double price = 0.0;

    public Product(String name, String desc, double price)
    {
        this.name = name;
        this.description = desc;
        this.price = price;
    }
}

package org.datanucleus.samples.jpa.tutorial;

public class Book extends Product
{
    String author=null;
    String isbn=null;
    String publisher=null;

    public Book(String name, String desc, double price, String author,
                String isbn, String publisher)
    {
        super(name,desc,price);
        this.author = author;
        this.isbn = isbn;
        this.publisher = publisher;
    }
}
So we have a relationship (Inventory having a set of Products), and inheritance (Product-Book). Now we need to be able to persist objects of all of these types, so we need to define persistence for them.

There are many things that you can define when deciding how to persist objects of a type but the essential parts are

- Mark the class as an Entity so it is visible to the persistence mechanism
- Identify which field(s) represent the identity of the object.

So this is what we do now. Note that we could define persistence using XML metadata, annotations. In this tutorial we will use annotations.

```java
package org.datanucleus.samples.jpa.tutorial;

@Entity
public class Inventory {
    @Id
    String name = null;

    @OneToMany(cascade={CascadeType.PERSIST, CascadeType.MERGE, CascadeType.DETACH})
    Set<Product> products = new HashSet();
    ...
}

package org.datanucleus.samples.jpa.tutorial;

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Product {
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE)
    long id;
    ...
}

package org.datanucleus.samples.jpa.tutorial;

@Entity
public class Book extends Product {
    ...
}
```

Note that we mark each class that can be persisted with @Entity and their primary key field(s) with @Id In addition we defined a valueStrategy for Product field id so that it will have its values generated automatically. In this tutorial we are using application identity which means that all objects of these classes will have their identity defined by the primary key field(s). You can read more in application identity when designing your systems persistence.
184.1.4 Step 2 : Define the 'persistence-unit'

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted. You do this via a file META-INF/persistence.xml at the root of the CLASSPATH. Like this

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">
    <!-- JPA tutorial "unit" -->
    <persistence-unit name="Tutorial">
        <class>org.datanucleus.samples.jpa.tutorial.Inventory</class>
        <class>org.datanucleus.samples.jpa.tutorial.Product</class>
        <class>org.datanucleus.samples.jpa.tutorial.Book</class>
        <exclude-unlisted-classes/>
        <properties>
            <property name="javax.persistence.jdbc.url" value="mongodb:/nucleus1"/>
            <property name="datanucleus.autoCreateSchema" value="true"/>
            <property name="datanucleus.validateTables" value="false"/>
            <property name="datanucleus.validateConstraints" value="false"/>
        </properties>
    </persistence-unit>
</persistence>
```

184.1.5 Step 3 : Enhance your classes

DataNucleus relies on the classes that you want to persist be enhanced to implement the interface PersistenceCapable. You could write your classes manually to do this but this would be laborious. Alternatively you can use a post-processing step to compilation that "enhances" your compiled classes, adding on the necessary extra methods to make them PersistenceCapable. There are several ways to do this, most notably at post-compile, or at runtime. We use the post-compile step in this tutorial. DataNucleus JPA provides its own byte-code enhancer for instrumenting/enhancing your classes (in datanucleus-core) and this is included in the DataNucleus AccessPlatform zip file prerequisite.

To understand on how to invoke the enhancer you need to visualise where the various source and jdo files are stored
The first thing to do is compile your domain/model classes. You can do this in any way you wish, but the downloadable JAR provides an Ant task, and a Maven2 project to do this for you.

Using Ant:
ant compile

Using Maven:
mvn compile

To enhance classes using the DataNucleus Enhancer, you need to invoke a command something like this from the root of your project.

Using Ant:
ant enhance

Using Maven: (this is usually done automatically after the "compile" goal)
mvn datanucleus:enhance

Manually on Linux/Unix:
java -cp target/classes:lib/datanucleus-core.jar:
   lib/datanucleus-api-jpa.jar:lib/persistence-api.jar:lib/jdo-api.jar:
org.datanucleus.enhancer.DataNucleusEnhancer -api JPA -pu Tutorial

Manually on Windows:
java -cp target\classes;lib\datanucleus-core.jar:
   lib\datanucleus-api-jpa.jar;lib\persistence-api.jar;lib\jdo-api.jar:
org.datanucleus.enhancer.DataNucleusEnhancer -api JPA -pu Tutorial

This command enhances all class files specified in the persistence-unit "Tutorial". If you accidentally omitted this step, at the point of running your application and trying to persist an object, you would get a `ClassNotFoundException` thrown. The use of the enhancer is documented in more detail in the Enhancer Guide. The output of this step are a set of class files that represent persistable classes.
184.1.6 Step 4: Write the code to persist objects of your classes

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted, and when. Interaction with the persistence framework of JPA is performed via an EntityManager. This provides methods for persisting of objects, removal of objects, querying for persisted objects, etc. This section gives examples of typical scenarios encountered in an application.

The initial step is to obtain access to an EntityManager, which you do as follows

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("Tutorial");
EntityManager em = emf.createEntityManager();
```

So we created an `EntityManagerFactory` for our "persistence-unit" called "Tutorial". Now that the application has an EntityManager it can persist objects. This is performed as follows

```java
Transaction tx = em.getTransaction();
try
{
    tx.begin();

    Inventory inv = new Inventory("My Inventory");
    Product product = new Product("Sony Discman", "A standard discman from Sony", 49.99);
    inv.getProducts().add(product);
    em.persist(inv);

    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }

    em.close();
}
```

Please note that the `finally` step is important in that it tidies up connections to the datastore and the EntityManager.

Now we want to retrieve some objects from persistent storage, so we will use a "Query". In our case we want access to all Product objects that have a price below 150.00 and ordering them in ascending order.
Transaction tx = em.getTransaction();
try {
    tx.begin();

    Query q = pm.createQuery("SELECT p FROM Product p WHERE p.price < 150.00");
    List results = q.getResultList();
    Iterator iter = results.iterator();
    while (iter.hasNext())
    {
        Product p = (Product)iter.next();
        ... (use the retrieved object)
    }
    tx.commit();
} finally {
    if (tx.isActive())
    {
        tx.rollback();
    }
    em.close();
}

If you want to delete an object from persistence, you would perform an operation something like

Transaction tx = em.getTransaction();
try {
    tx.begin();

    // Find and delete all objects whose last name is 'Jones'
    Query q = em.createQuery("DELETE FROM Person p WHERE p.lastName = 'Jones'");
    int numberInstancesDeleted = q.executeUpdate();
    tx.commit();
} finally {
    if (tx.isActive())
    {
        tx.rollback();
    }
    em.close();
}

Clearly you can perform a large range of operations on objects. We can't hope to show all of these here. Any good JPA book will provide many examples.
184.1.7 Step 5: Run your application

To run your JPA-enabled application will require a few things to be available in the Java CLASSPATH, these being:

- The "persistence.xml" file (stored under META-INF/)
- Any ORM MetaData files for your persistable classes
- MongoDB Java driver jar needed for accessing your datastore
- The JDO API JAR (defining the JDO interface) - DataNucleus JPA also requires this currently
- The JPA API JAR (defining the JPA interface)
- The DataNucleus Core, DataNucleus JPA API and DataNucleus MongoDB JARs

After that it is simply a question of starting your application and all should be taken care of. You can access the DataNucleus Log file by specifying the `logging` configuration properties, and any messages from DataNucleus will be output in the normal way. The DataNucleus log is a very powerful way of finding problems since it can list all SQL actually sent to the datastore as well as many other parts of the persistence process.

Using Ant (you need the included persistence.xml to specify your database)

```shell
ant run
```

Using Maven:

```shell
mvn exec:java
```

Manually on Linux/Unix:

```shell
org.datanucleus.samples.jpa.tutorial.Main
```

Manually on Windows:

```shell
test -cp lib\persistence-api.jar;lib\jdo-api.jar;lib\datanucleus-core.jar;lib\datanucleus-mongodb.jar;lib\datanucleus-api-jpa.jar;lib\mongo-java.jar;target\classes\.
org.datanucleus.samples.jpa.tutorial.Main
```

Output:

DataNucleus Tutorial with JPA
=============================
Persisting products
Product and Book have been persisted

Executing Query for Products with price below 150.00
> Book : JRR Tolkien - Lord of the Rings by Tolkien

Deleting all products from persistence

End of Tutorial
184.2 Part 2: Next steps

In the above simple tutorial we showed how to employ JPA and persist objects to a MongoDB database. Obviously this just scratches the surface of what you can do, and to use JPA requires minimal work from the user. In this second part we show some further things that you are likely to want to do.

1. Step 6: Controlling the schema.
2. Step 7: Generate the database tables where your classes are to be persisted using SchemaTool.

184.2.1 Step 6: Controlling the schema

In the above simple tutorial we didn't look at controlling the schema generated for these classes. Now let's pay more attention to this part by defining XML Metadata for the schema. We define this in XML to separate schema information from persistence information. So we define a file `orm.xml`
<?xml version="1.0" encoding="UTF-8" ?>
<entity-mappings>
  <description>DataNucleus JPA tutorial</description>
  <package>org.datanucleus.samples.jpa.tutorial</package>
  <entity class="org.datanucleus.samples.jpa.tutorial.Product" name="Product">
    <table name="JPA_PRODUCTS"/>
    <attributes>
      <id name="id">
        <generated-value strategy="TABLE"/>
      </id>
      <basic name="name">
        <column name="PRODUCT_NAME"/>
      </basic>
      <basic name="description">
        <column name="Desc"/>
      </basic>
    </attributes>
  </entity>
  <entity class="org.datanucleus.samples.jpa.tutorial.Book" name="Book">
    <table name="JPA_BOOKS"/>
    <attributes>
      <basic name="isbn">
        <column name="ISBN"/>
      </basic>
      <basic name="author">
        <column name="AUTHOR"/>
      </basic>
      <basic name="publisher">
        <column name="PUBLISHER"/>
      </basic>
    </attributes>
  </entity>
  <entity class="org.datanucleus.samples.jpa.tutorial.Inventory" name="Inventory">
    <table name="JPA_INVENTORY"/>
    <attributes>
      <id name="name">
        <column name="NAME" length="40"/>
      </id>
      <one-to-many name="products"/>
    </attributes>
  </entity>
</entity-mappings>

This file should be placed at the root of the CLASSPATH under META-INF.
184.2.2 Step 7 : Generate any schema required for your domain classes

This step is optional, depending on whether you have an existing database schema. If you haven't, at this point you can use the DataNucleus SchemaTool to generate the tables where these domain objects will be persisted. DataNucleus SchemaTool is a command line utility (it can be invoked from Maven/Ant in a similar way to how the Enhancer is invoked). The first thing that you need is to update the src/java/META-INF/persistence.xml file with your database details. Here we have a sample file (for HSQLDB) that contains

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">

  <!-- Tutorial "unit" -->
  <persistence-unit name="Tutorial">
    <class>org.datanucleus.samples.jpa.tutorial.Inventory</class>
    <class>org.datanucleus.samples.jpa.tutorial.Product</class>
    <class>org.datanucleus.samples.jpa.tutorial.Book</class>
    <exclude-unlisted-classes/>
    <properties>
      <property name="javax.persistence.jdbc.url" value="mongodb:/nucleus1"/>
      <property name="datanucleus.autoCreateSchema" value="true"/>
      <property name="datanucleus.validateTables" value="false"/>
      <property name="datanucleus.validateConstraints" value="false"/>
    </properties>
  </persistence-unit>
</persistence>
```

Now we need to run DataNucleus SchemaTool. For our case above you would do something like this
Using Ant:
ant createschema

Using Maven:
mvn datanucleus:schema-create

Manually on Linux/Unix:
java -cp target/classes:lib/jdo-api.jar:lib/persistence-api.jar:lib/datanucleus-core.jar:
  lib/datanucleus-mongodb.jar:lib/datanucleus-api-jpa.jar:lib/{mongo-java.jar}
  org.datanucleus.store.schema.SchemaTool
  -create -api JPA -pu Tutorial

Manually on Windows:
java -cp target\classes;lib\jdo-api.jar;lib\persistence-api.jar;lib\datanucleus-core.jar;
  lib\datanucleus-mongodb.jar;lib\datanucleus-api-jpa.jar;lib\{mongo-java.jar}
  org.datanucleus.store.schema.SchemaTool
  -create -api JPA -pu Tutorial

[Command shown on many lines to aid reading. Should be on single line]

This will generate the required tables, indexes, and foreign keys for the classes defined in the
annotations and orm.xml Meta-Data file.

184.2.3 Any questions?

If you have any questions about this tutorial and how to develop applications for use with
DataNucleus please read the online documentation since answers are to be found there. If you don't
find what you're looking for go to our Forums.

The DataNucleus Team
185 Tutorial with HBase

185.1 DataNucleus - Tutorial for JPA for HBase

185.1.1 Background

An application can be JPA-enabled via many routes depending on the development process of the project in question. For example the project could use Eclipse as the IDE for developing classes. In that case the project would typically use the Dali Eclipse plugin coupled with the DataNucleus Eclipse plugin. Alternatively the project could use Ant, Maven2 or some other build tool. In this case this tutorial should be used as a guiding way for using DataNucleus in the application. The JPA process is quite straightforward.

1. Prerequisite: Download DataNucleus AccessPlatform
2. Step 1: Define their persistence definition using Meta-Data.
3. Step 2: Define the "persistence-unit"
4. Step 3: Compile your classes, and instrument them (using the DataNucleus enhancer).
5. Step 4: Write your code to persist your objects within the DAO layer.
6. Step 5: Run your application.

The tutorial guides you through this. You can obtain the code referenced in this tutorial from SourceForge (one of the files entitled "datanucleus-samples-jpa-tutorial-*").

185.1.2 Step 0: Download DataNucleus AccessPlatform

You can download DataNucleus in many ways, but the simplest is to download the distribution zip appropriate to your datastore (in this case HBase). You can do this from SourceForge DataNucleus download page. When you open the zip you will find DataNucleus jars in the lib directory, and dependency jars in the deps directory.

185.1.3 Step 1: Take your model classes and mark which are persistable

For our tutorial, say we have the following classes representing a store of products for sale.
package org.datanucleus.samples.jpa.tutorial;

public class Inventory
{
    String name = null;
    Set<Product> products = new HashSet();

    public Inventory(String name)
    {
        this.name = name;
    }

    public Set<Product> getProducts() {return products;}
}

package org.datanucleus.samples.jpa.tutorial;

public class Product
{
    long id;
    String name = null;
    String description = null;
    double price = 0.0;

    public Product(String name, String desc, double price)
    {
        this.name = name;
        this.description = desc;
        this.price = price;
    }
}

package org.datanucleus.samples.jpa.tutorial;

public class Book extends Product
{
    String author=null;
    String isbn=null;
    String publisher=null;

    public Book(String name, String desc, double price, String author,
            String isbn, String publisher)
    {
        super(name,desc,price);
        this.author = author;
        this.isbn = isbn;
        this.publisher = publisher;
    }
}
So we have a relationship (Inventory having a set of Products), and inheritance (Product-Book). Now we need to be able to persist objects of all of these types, so we need to **define persistence for them**. There are many things that you can define when deciding how to persist objects of a type but the essential parts are

- Mark the class as an *Entity* so it is visible to the persistence mechanism
- Identify which field(s) represent the identity of the object.

So this is what we do now. Note that we could define persistence using XML metadata, annotations. In this tutorial we will use annotations.

```java
package org.datanucleus.samples.jpa.tutorial;

@Entity
public class Inventory {
    @Id
    String name = null;

    @OneToMany(cascade={CascadeType.PERSIST, CascadeType.MERGE, CascadeType.DETACH})
    Set<Product> products = new HashSet();
    ...
}

package org.datanucleus.samples.jpa.tutorial;

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Product {
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE)
    long id;
    ...
}

package org.datanucleus.samples.jpa.tutorial;

@Entity
public class Book extends Product {
    ...
}
```

Note that we mark each class that can be persisted with `@Entity` and their primary key field(s) with `@Id`. In addition we defined a `valueStrategy` for Product field `id` so that it will have its values generated automatically. In this tutorial we are using application identity which means that all objects of these classes will have their identity defined by the primary key field(s). You can read more in application identity when designing your systems persistence.
185.1.4 Step 2 : Define the 'persistence-unit'

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted. You do this via a file META-INF/persistence.xml at the root of the CLASSPATH. Like this

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">

    <!-- JPA tutorial "unit" -->
    <persistence-unit name="Tutorial">
        <class>org.datanucleus.samples.jpa.tutorial.Inventory</class>
        <class>org.datanucleus.samples.jpa.tutorial.Product</class>
        <class>org.datanucleus.samples.jpa.tutorial.Book</class>
        <exclude-unlisted-classes/>
        <properties>
            <property name="javax.persistence.jdbc.url" value="hbase:"/>
        </properties>
    </persistence-unit>
</persistence>
```

185.1.5 Step 3 : Enhance your classes

DataNucleus relies on the classes that you want to persist be enhanced to implement the interface PersistenceCapable. You could write your classes manually to do this but this would be laborious. Alternatively you can use a post-processing step to compilation that "enhances" your compiled classes, adding on the necessary extra methods to make them PersistenceCapable. There are several ways to do this, most notably at post-compile, or at runtime. We use the post-compile step in this tutorial. DataNucleus JPA provides its own byte-code enhancer for instrumenting/enhancing your classes (in datanucleus-core) and this is included in the DataNucleus AccessPlatform zip file prerequisite.

To understand on how to invoke the enhancer you need to visualise where the various source and jdo files are stored
The first thing to do is compile your domain/model classes. You can do this in any way you wish, but the downloadable JAR provides an Ant task, and a Maven2 project to do this for you.

Using Ant:

tool compile

Using Maven:

tool compile

To enhance classes using the DataNucleus Enhancer, you need to invoke a command something like this from the root of your project.

Using Ant:

tool enhance

Using Maven: (this is usually done automatically after the "compile" goal)

tool datanucleus:enhance

Manually on Linux/Unix:

java -cp target/classes:lib/datanucleus-core.jar:
    lib/datanucleus-api-jpa.jar:lib/persistence-api.jar:lib/jdo-api.jar
    org.datanucleus.enhancer.DataNucleusEnhancer
    -api JPA -pu Tutorial

Manually on Windows:

java -cp target\classes;lib\datanucleus-core.jar;
    lib\datanucleus-api-jpa.jar;lib\persistence-api.jar;lib\jdo-api.jar
    org.datanucleus.enhancer.DataNucleusEnhancer
    -api JPA -pu Tutorial

[Command shown on many lines to aid reading - should be on single line]

This command enhances all class files specified in the persistence-unit "Tutorial". If you accidentally omitted this step, at the point of running your application and trying to persist an object, you would get a ClassNotPersistenceCapableException thrown. The use of the enhancer is documented in more
detail in the Enhancer Guide. The output of this step are a set of class files that represent persistable classes.

185.1.6 Step 4: Write the code to persist objects of your classes

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted, and when. Interaction with the persistence framework of JPA is performed via an EntityManager. This provides methods for persisting of objects, removal of objects, querying for persisted objects, etc. This section gives examples of typical scenarios encountered in an application.

The initial step is to obtain access to an EntityManager, which you do as follows

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("Tutorial");
EntityManager em = emf.createEntityManager();
```

So we created an `EntityManagerFactory` for our "persistence-unit" called "Tutorial". Now that the application has an EntityManager it can persist objects. This is performed as follows

```java
Transaction tx = em.getTransaction();
try {
    tx.begin();
    Inventory inv = new Inventory("My Inventory");
    Product product = new Product("Sony Discman", "A standard discman from Sony", 49.99);
    inv.getProducts().add(product);
    em.persist(inv);
    tx.commit();
} finally {
    if (tx.isActive()) {
        tx.rollback();
    }
    em.close();
}
```

Please note that the `finally` step is important in that it tidies up connections to the datastore and the EntityManager.

Now we want to retrieve some objects from persistent storage, so we will use a "Query". In our case we want access to all Product objects that have a price below 150.00 and ordering them in ascending order.
Transaction tx = em.getTransaction();
try {
    tx.begin();
    Query q = pm.createQuery("SELECT p FROM Product p WHERE p.price < 150.00");
    List results = q.getResultList();
    Iterator iter = results.iterator();
    while (iter.hasNext())
    {
        Product p = (Product)iter.next();
        ... (use the retrieved object)
    }
    tx.commit();
} finally {
    if (tx.isActive())
    {
        tx.rollback();
    }
    em.close();
}

If you want to delete an object from persistence, you would perform an operation something like

Transaction tx = em.getTransaction();
try {
    tx.begin();
    // Find and delete all objects whose last name is 'Jones'
    Query q = em.createQuery("DELETE FROM Person p WHERE p.lastName = 'Jones'");
    int numberInstancesDeleted = q.executeUpdate();
    tx.commit();
} finally {
    if (tx.isActive())
    {
        tx.rollback();
    }
    em.close();
}

Clearly you can perform a large range of operations on objects. We can’t hope to show all of these here. Any good JPA book will provide many examples.
185.1.7 Step 5 : Run your application

To run your JPA-enabled application will require a few things to be available in the Java
CLASSPATH, these being

- The "persistence.xml" file (stored under META-INF/)
- Any ORM MetaData files for your persistable classes
- HBase/Hadoop jars needed for accessing your datastore
- The JDO API JAR (defining the JDO bytecode enhancement contract)
- The JPA API JAR (defining the JPA interface)
- The DataNucleus Core, DataNucleus JPA API and DataNucleus HBase JARs

After that it is simply a question of starting your application and all should be taken care of. You
can access the DataNucleus Log file by specifying the logging configuration properties, and any
messages from DataNucleus will be output in the normal way. The DataNucleus log is a very
powerful way of finding problems since it can list all SQL actually sent to the datastore as well as
many other parts of the persistence process.

Using Ant (you need the included persistence.xml to specify your database)
ant run

Using Maven:
mvn exec:java

Manually on Linux/Unix :
java -cp lib/persistence-api.jar;lib/jdo-api.jar;lib/datanucleus-core.jar;lib/datanucleus-hbase.jar;
lbin/datanucleus-api-jpa.jar;lib/{hbase_jars}:target/classes/;.
org.datanucleus.samples.jpa.tutorial.Main

Manually on Windows :
java -cp lib\persistence-api.jar;lib\jdo-api.jar;lib\datanucleus-core.jar;lib\datanucleus-hbase.jar;
lbin\datanucleus-api-jpa.jar;lib\{hbase_jars}\target\classes\;.
org.datanucleus.samples.jpa.tutorial.Main

Output :

DataNucleus Tutorial with JPA
================================
Persisting products
Product and Book have been persisted

Executing Query for Products with price below 150.00
Book : JRR Tolkien - Lord of the Rings by Tolkien

Deleting all products from persistence

End of Tutorial
185.2 Part 2 : Next steps

In the above simple tutorial we showed how to employ JPA and persist objects to a HBase database. Obviously this just scratches the surface of what you can do, and to use JPA requires minimal work from the user. In this second part we show some further things that you are likely to want to do.

1. Step 6 : Controlling the schema.
2. Step 7 : Generate the database tables where your classes are to be persisted using SchemaTool.

185.2.1 Step 6 : Controlling the schema

In the above simple tutorial we didn't look at controlling the schema generated for these classes. Now let's pay more attention to this part by defining XML Metadata for the schema. We define this in XML to separate schema information from persistence information. So we define a file `orm.xml`
This file should be placed at the root of the CLASSPATH under `META-INF`.
185.2.2 Step 7: Generate any schema required for your domain classes

This step is optional, depending on whether you have an existing database schema. If you haven’t, at this point you can use the DataNucleus SchemaTool to generate the tables where these domain objects will be persisted. DataNucleus SchemaTool is a command line utility (it can be invoked from Maven/Ant in a similar way to how the Enhancer is invoked). The first thing that you need is to update the src/java/META-INF/persistence.xml file with your database details. Here we have a sample file (for HSQLDB) that contains

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">
    <!-- Tutorial "unit" -->
    <persistence-unit name="Tutorial">
        <class>org.datanucleus.samples.jpa.tutorial.Inventory</class>
        <class>org.datanucleus.samples.jpa.tutorial.Product</class>
        <class>org.datanucleus.samples.jpa.tutorial.Book</class>
        <exclude-unlisted-classes/>
        <properties>
            <property name="datanucleus.ConnectionURL" value="hbase:">
            <property name="datanucleus.autoCreateSchema" value="true"/>
            <property name="datanucleus.validateTables" value="false"/>
            <property name="datanucleus.validateConstraints" value="false"/>
        </properties>
    </persistence-unit>
</persistence>
```

Now we need to run DataNucleus SchemaTool. For our case above you would do something like this
Using Ant:

ant createschema

Using Maven:

mvn datanucleus:schema-create

Manually on Linux/Unix:

java -cp target/classes:lib/jdo-api.jar:lib/persistence-api.jar:lib/datanucleus-core.jar:
    lib/datanucleus-hbase.jar:lib/datanucleus-api-jpa.jar:lib/{hbase-jars}
    org.datanucleus.store.schema.SchemaTool
    -create -api JPA -pu Tutorial

Manually on Windows:

java -cp target\classes;lib\jdo-api.jar;lib\persistence-api.jar;lib\datanucleus-core.jar;
    lib\datanucleus-hbase.jar;lib\datanucleus-api-jpa.jar;lib\{hbase-jars}
    org.datanucleus.store.schema.SchemaTool
    -create -api JPA -pu Tutorial

[Command shown on many lines to aid reading. Should be on single line]

This will generate the required tables, indexes, and foreign keys for the classes defined in the
annotations and orm.xml Meta-Data file.

185.2.3 Any questions?

If you have any questions about this tutorial and how to develop applications for use with
DataNucleus please read the online documentation since answers are to be found there. If you don't
find what you're looking for go to our Forums.

The DataNucleus Team
186 Tutorial with Neo4j

186.1 DataNucleus - Tutorial for JPA for Neo4j

186.1.1 Background
An application can be JPA-enabled via many routes depending on the development process of the project in question. For example the project could use Eclipse as the IDE for developing classes. In that case the project would typically use the Dali Eclipse plugin coupled with the DataNucleus Eclipse plugin. Alternatively the project could use Ant, Maven2 or some other build tool. In this case this tutorial should be used as a guiding way for using DataNucleus in the application. The JPA process is quite straightforward.

1. Prerequisite : Download DataNucleus AccessPlatform
2. Step 1 : Define their persistence definition using Meta-Data.
3. Step 2 : Define the "persistence-unit"
4. Step 3 : Compile your classes, and instrument them (using the DataNucleus enhancer).
5. Step 4 : Write your code to persist your objects within the DAO layer.
6. Step 5 : Run your application.

The tutorial guides you through this. You can obtain the code referenced in this tutorial from SourceForge (one of the files entitled "datanucleus-samples-jpa-tutorial-*").

186.1.2 Step 0 : Download DataNucleus AccessPlatform

You can download DataNucleus in many ways, but the simplest is to download the distribution zip appropriate to your datastore (in this case Neo4j, so get the full download). You can do this from SourceForge DataNucleus download page. When you open the zip you will find DataNucleus jars in the lib directory, and dependency jars in the deps directory.

186.1.3 Step 1 : Take your model classes and mark which are persistable

For our tutorial, say we have the following classes representing a store of products for sale.
package org.datanucleus.samples.jpa.tutorial;

class Inventory
{
    String name = null;
    Set<Product> products = new HashSet();

    public Inventory(String name)
    {
        this.name = name;
    }

    public Set<Product> getProducts() { return products; }
}

class Product
{
    long id;
    String name = null;
    String description = null;
    double price = 0.0;

    public Product(String name, String desc, double price)
    {
        this.name = name;
        this.description = desc;
        this.price = price;
    }
}

class Book extends Product
{
    String author=null;
    String isbn=null;
    String publisher=null;

    public Book(String name, String desc, double price, String author,
    String isbn, String publisher)
    {
        super(name,desc,price);
        this.author = author;
        this.isbn = isbn;
        this.publisher = publisher;
    }
}
So we have a relationship (Inventory having a set of Products), and inheritance (Product-Book). Now we need to be able to persist objects of all of these types, so we need to define persistence for them. There are many things that you can define when deciding how to persist objects of a type but the essential parts are

- Mark the class as an **Entity** so it is visible to the persistence mechanism
- Identify which field(s) represent the identity of the object.

So this is what we do now. Note that we could define persistence using XML metadata, annotations. In this tutorial we will use annotations.

```java
package org.datanucleus.samples.jpa.tutorial;

@Entity
public class Inventory
{
    @Id
    String name = null;

    @OneToMany(cascade={CascadeType.PERSIST, CascadeType.MERGE, CascadeType.DETACH})
    Set<Product> products = new HashSet();
    ...
}
```

```java
package org.datanucleus.samples.jpa.tutorial;

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Product
{
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE)
    long id;
    ...
}
```

```java
package org.datanucleus.samples.jpa.tutorial;

@Entity
public class Book extends Product
{
    ...
}
```

Note that we mark each class that can be persisted with `@Entity` and their primary key field(s) with `@Id` In addition we defined a `valueStrategy` for Product field `id` so that it will have its values generated automatically. In this tutorial we are using application identity which means that all objects of these classes will have their identity defined by the primary key field(s). You can read more in application identity when designing your systems persistence.
186.1.4 Step 2 : Define the 'persistence-unit'

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted. You do this via a file META-INF/persistence.xml at the root of the CLASSPATH. Like this

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
 http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">
  <!-- JPA tutorial "unit" -->
  <persistence-unit name="Tutorial">
    <class>org.datanucleus.samples.jpa.tutorial.Inventory</class>
    <class>org.datanucleus.samples.jpa.tutorial.Product</class>
    <class>org.datanucleus.samples.jpa.tutorial.Book</class>
    <exclude-unlisted-classes/>
    <properties>
      <property name="javax.persistence.jdbc.url" value="neo4j:testDB"/>
    </properties>
  </persistence-unit>
</persistence>
```

186.1.5 Step 3 : Enhance your classes

DataNucleus relies on the classes that you want to persist be enhanced to implement the interface PersistenceCapable. You could write your classes manually to do this but this would be laborious. Alternatively you can use a post-processing step to compilation that "enhances" your compiled classes, adding on the necessary extra methods to make them PersistenceCapable. There are several ways to do this, most notably at post-compile, or at runtime. We use the post-compile step in this tutorial. DataNucleus JPA provides its own byte-code enhancer for instrumenting/enhancing your classes (in datanucleus-core) and this is included in the DataNucleus AccessPlatform zip file prerequisite.

To understand on how to invoke the enhancer you need to visualise where the various source and jdo files are stored
The first thing to do is compile your domain/model classes. You can do this in any way you wish, but the downloadable JAR provides an Ant task, and a Maven2 project to do this for you.

To enhance classes using the DataNucleus Enhancer, you need to invoke a command something like this from the root of your project.

This command enhances all class files specified in the persistence-unit "Tutorial". If you accidentally omitted this step, at the point of running your application and trying to persist an object, you would get a `ClassNotPersistenceCapableException` thrown. The use of the enhancer is documented in more
detail in the Enhancer Guide. The output of this step are a set of class files that represent persistable classes.

186.1.6 Step 4: Write the code to persist objects of your classes

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted, and when. Interaction with the persistence framework of JPA is performed via an EntityManager. This provides methods for persisting of objects, removal of objects, querying for persisted objects, etc. This section gives examples of typical scenarios encountered in an application.

The initial step is to obtain access to an EntityManager, which you do as follows

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("Tutorial");
EntityManager em = emf.createEntityManager();
```

So we created an EntityManagerFactory for our "persistence-unit" called "Tutorial", and an EntityManager. Now that the application has an EntityManager it can persist objects. This is performed as follows

```java
Transaction tx = em.getTransaction();
try {
    tx.begin();

    Inventory inv = new Inventory("My Inventory");
    Product product = new Product("Sony Discman", "A standard discman from Sony", 49.99);
    inv.getProducts().add(product);
    em.persist(inv);

    tx.commit();
} finally {
    if (tx.isActive()) {
        tx.rollback();
    }
    em.close();
}
```

Please note that the finally step is important in that it tidies up connections to the datastore and the EntityManager.

Now we want to retrieve some objects from persistent storage, so we will use a "Query". In our case we want access to all Product objects that have a price below 150.00 and ordering them in ascending order.
Transaction tx = em.getTransaction();
try {
    tx.begin();

    Query q = pm.createQuery("SELECT p FROM Product p WHERE p.price < 150.00");
    List results = q.getResultList();
    Iterator iter = results.iterator();
    while (iter.hasNext())
    {
        Product p = (Product) iter.next();
        ... (use the retrieved object)
    }
    tx.commit();
}
finally {
    if (tx.isActive())
    {
        tx.rollback();
    }
    em.close();
}

If you want to delete an object from persistence, you would perform an operation something like

Transaction tx = em.getTransaction();
try {
    tx.begin();

    // Find and delete all objects whose last name is 'Jones'
    Query q = em.createQuery("DELETE FROM Person p WHERE p.lastName = 'Jones'");
    int numberInstancesDeleted = q.executeUpdate();
    tx.commit();
}
finally {
    if (tx.isActive())
    {
        tx.rollback();
    }
    em.close();
}

Clearly you can perform a large range of operations on objects. We can't hope to show all of these here. Any good JPA book will provide many examples.
186.1.7 Step 5: Run your application

To run your JPA-enabled application will require a few things to be available in the Java CLASSPATH, these being:

- The "persistence.xml" file (stored under META-INF/)
- Any ORM MetaData files for your persistable classes
- Neo4J jar(s) needed for accessing your datastore
- The JDO API JAR (for the bytecode enhancement contract)
- The JPA API JAR (defining the JPA interface)
- The DataNucleus Core, DataNucleus JPA API and DataNucleus Neo4J JARs

After that it is simply a question of starting your application and all should be taken care of. You can access the DataNucleus Log file by specifying the logging configuration properties, and any messages from DataNucleus will be output in the normal way. The DataNucleus log is a very powerful way of finding problems since it can list all SQL actually sent to the datastore as well as many other parts of the persistence process.

Using Ant (you need the included persistence.xml to specify your database)

ant run

Using Maven:

mvn exec:java

Manually on Linux/Unix:

```
java -cp lib/persistence-api.jar:lib/jdo-api.jar:lib/datanucleus-core.jar:lib/datanucleus-neo4j.jar:
    lib/datanucleus-api-jpa.jar:lib/neo4j-jars:target/classes/:
    org.datanucleus.samples.jpa.tutorial.Main
```

Manually on Windows:

```
java -cp lib\persistence-api.jar;lib\jdo-api.jar;lib\datanucleus-core.jar;lib\datanucleus-neo4j.jar;
    lib\datanucleus-api-jpa.jar;lib\neo4j_jars;target\classes\;
    org.datanucleus.samples.jpa.tutorial.Main
```

Output:

DataNucleus Tutorial with JPA

Persisting products

Product and Book have been persisted

Executing Query for Products with price below 150.00

> Book: JRR Tolkien - Lord of the Rings by Tolkien

Deleting all products from persistence

End of Tutorial
186.2 Part 2 : Next steps

In the above simple tutorial we showed how to employ JPA and persist objects to a Neo4J database. Obviously this just scratches the surface of what you can do, and to use JPA requires minimal work from the user. In this second part we show some further things that you are likely to want to do.

1. Step 6 : Controlling the schema.

186.2.1 Step 6 : Controlling the schema

In the above simple tutorial we didn't look at controlling the schema generated for these classes. Now let's pay more attention to this part by defining XML Metadata for the schema. We define this in XML to separate schema information from persistence information. So we define a file `orm.xml`
<?xml version="1.0" encoding="UTF-8" ?>
<entity-mappings>
  <description>DataNucleus JPA tutorial</description>
  <package>org.datanucleus.samples.jpa.tutorial</package>
  <entity class="org.datanucleus.samples.jpa.tutorial.Product" name="Product">
    <attributes>
      <id name="id">
        <generated-value strategy="AUTO"/>
      </id>
      <basic name="name">
        <column name="PRODUCT_NAME"/>
      </basic>
      <basic name="description">
        <column name="Desc"/>
      </basic>
    </attributes>
  </entity>
  <entity class="org.datanucleus.samples.jpa.tutorial.Book" name="Book">
    <attributes>
      <basic name="isbn">
        <column name="ISBN"/>
      </basic>
      <basic name="author">
        <column name="AUTHOR"/>
      </basic>
      <basic name="publisher">
        <column name="PUBLISHER"/>
      </basic>
    </attributes>
  </entity>
  <entity class="org.datanucleus.samples.jpa.tutorial.Inventory" name="Inventory">
    <attributes>
      <id name="name">
        <column name="NAME"/>
      </id>
      <one-to-many name="products"/>
    </attributes>
  </entity>
</entity-mappings>

This file should be placed at the root of the CLASSPATH under META-INF.

186.2.2 Any questions?

If you have any questions about this tutorial and how to develop applications for use with DataNucleus please read the online documentation since answers are to be found there. If you don't find what you're looking for go to our Forums.

The DataNucleus Team
187 Eclipse Dali

187.1 DataNucleus, Eclipse Dali, JPA

The Eclipse Dali project provides a powerful development environment for Java Persistence. DataNucleus does not stay behind, and permits the powerful DataNucleus persistence engine to be combined with Eclipse Dali for development.

In this (5 mins) tutorial, we use Eclipse Dali to reverse engineer a database table (ACCOUNT) and generate a persistent class (Account). The DataNucleus Eclipse plug-in is used to enhance the persistent class before running the application.

187.1.1 Requirements

For using the IDE, you must install Eclipse 3.2, Eclipse Dali and the DataNucleus Eclipse plug-in. For using the DataNucleus runtime, see JPA annotations.

187.1.2 Demo

187.1.3 Source Code

The source code for org.jpox.demo.Account class.
package org.jpox.demo;

import java.io.Serializable;
import java.math.BigDecimal;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Account implements Serializable {
    @Id
    @Column(name="ACCOUNT_ID")
    private BigDecimal accountId;

    private String username;
    private BigDecimal enabled;
    private static final long serialVersionUID = 1L;

    public Account() {
        super();
    }

    public BigDecimal getAccountId() {
        return this.accountId;
    }

    public void setAccountId(BigDecimal accountId) {
        this.accountId = accountId;
    }

    public String getUsername() {
        return this.username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public BigDecimal getEnabled() {
        return this.enabled;
    }

    public void setEnabled(BigDecimal enabled) {
        this.enabled = enabled;
    }
}

The source code for org.jpox.demo.Main class.
package org.jpox.demo;

import java.math.BigDecimal;
import java.util.Random;

import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;

public class Main
{
    public static void main(String[] args)
    {
        java.io.InputStream is = Main.class.getClassLoader().getResourceAsStream("PMFProperties.properties");
        PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(is);
        PersistenceManager pm = pmf.getPersistenceManager();
        try
            {
                pm.currentTransaction().begin();
                Account acc = new Account();
                BigDecimal dec = new BigDecimal(new Random().nextInt());
                acc.setAccountId(dec);
                acc.setEnabled(BigDecimal.ONE);
                pm.makePersistent(acc);
                pm.currentTransaction().commit();
                System.out.println("Account "+dec+" was persisted.");
            }
        finally
            {
                if( pm.currentTransaction().isActive() )
                    { pm.currentTransaction().rollback();}
                pm.close();
            }
    }
}

The source code for PMFProperties.properties file.
The database schema model.

```
CREATE TABLE Account (
    ACCOUNT_ID NUMBER NOT NULL,
    username VARCHAR2(255),
    enabled NUMBER(1, 0) NOT NULL
);

ALTER TABLE Account ADD CONSTRAINT Account_PK PRIMARY KEY (ACCOUNT_ID);
```
188 REST API

188.1 REST API
The DataNucleus REST API provides a RESTful interface to persist JSON objects to the datastore. All entities are accessed, queried and stored as resources via well defined HTTP methods. This API consists of a servlet that internally handles the persistence of objects (using JDO). Your POJO classes need to be accessible from this servlet, and can use either JDO or JPA metadata (annotations or XML). The REST API automatically exposes the persistent class in RESTful style, and requires minimum configuration as detailed in the sections linked below.

188.2 Servlet Configuration

The configuration of the REST API consists in the deployment of jar libraries to the CLASSPATH and the configuration of the servlet in the /WEB-INF/web.xml. After it’s configured, all persistent classes are automatically exposed via RESTful HTTP interface. You need to have enhanced versions of the model classes in the CLASSPATH.

188.2.1 Libraries
DataNucleus REST API requires the libraries: datanucleus-core, datanucleus-api-rest, datanucleus-api-jdo, jdo-api, as well as datanucleus-rdbms (or whichever datastore you wish to persist to if not RDBMS). You would also require JPA API jar if using JPA metadata (XML/annotations) in your model classes. In WAR files, these libraries are deployed under the folder /WEB-INF/lib/.

188.2.2 web.xml
The DataNucleus REST Servlet class implementation is org.datanucleus.api.rest.RestServlet. It has to be configured in the /WEB-INF/web.xml file, and it takes one initialisation parameter.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>persistence-context</td>
<td>Name of a PMF (if using jdoconfig.xml), or the name of a persistence-unit (if using persistence.xml) accessible to the servlet</td>
</tr>
</tbody>
</table>
changing `myPMFName` to the name of your PMF, or the name of your persistence-unit, and changing `/dn/*` to the URL pattern where you want DataNucleus REST API calls to be answered.

### 188.3 HTTP Methods

The persistence to the datastore in your application is performed via HTTP methods as following:

<table>
<thead>
<tr>
<th>Method</th>
<th>Operation</th>
<th>URL format</th>
<th>Return</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>Insert object</td>
<td>/{full-class-name}</td>
<td>The JSON Object is returned.</td>
<td>The JSON Object is passed in the HTTP Content.</td>
</tr>
<tr>
<td>PUT</td>
<td>Update object</td>
<td>/{full-class-name}/[primary key]</td>
<td>The JSON Object is returned.</td>
<td>The JSON Object is passed in the HTTP Content. The primary-key is specified in the URL if the PK is application-identity single field or if it is datastore-identity.</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete object</td>
<td>/{full-class-name}/[primary key]</td>
<td></td>
<td>The primary key fields are passed in the HTTP Content (JSONObject) if the PK uses multiple PK fields, otherwise in the URL.</td>
</tr>
</tbody>
</table>
### DELETE
Delete all objects of type

<table>
<thead>
<tr>
<th>Request Method</th>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE</td>
<td>/{full-class-name}</td>
<td>JSON Array of JSON objects</td>
</tr>
</tbody>
</table>

### GET
Fetch all objects of type

<table>
<thead>
<tr>
<th>Request Method</th>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/{full-class-name}?fetch={fetchGroupName}</td>
<td>JSON Array of JSON objects</td>
</tr>
</tbody>
</table>

### GET
Fetch a single object

<table>
<thead>
<tr>
<th>Request Method</th>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/{full-class-name}/ {primary key}?fetch={fetchGroupName}</td>
<td>A JSON object</td>
</tr>
</tbody>
</table>

### GET
Query objects via a filter. Returns a JSON Array of objects

<table>
<thead>
<tr>
<th>Request Method</th>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/{full-class-name}?{filter} [&amp;fetch={fetchGroupName}]</td>
<td>JSON Array of JSON objects</td>
</tr>
</tbody>
</table>

### GET
Query objects via JDOQL. Returns a JSON Array of objects

<table>
<thead>
<tr>
<th>Request Method</th>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/jdoql?{JDOQL-single-string-query} [&amp;fetch={fetchGroupName}]</td>
<td>JSON Array of JSON objects</td>
</tr>
</tbody>
</table>

### GET
Query objects via JPQL. Returns a JSON Array of objects

<table>
<thead>
<tr>
<th>Request Method</th>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/jpql?{JPQL-single-string-query} [&amp;fetch={fetchGroupName}]</td>
<td>JSON Array of JSON objects</td>
</tr>
</tbody>
</table>

### HEAD
Validates if an object exists

<table>
<thead>
<tr>
<th>Request Method</th>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEAD</td>
<td>/{full-class-name}/ {primary key}</td>
<td>The primary key fields are passed in the HTTP Content (JSONObject) if the PK uses multiple PK fields, otherwise in the URL</td>
</tr>
</tbody>
</table>

## 188.4 Example REST Usages

Note that the URL in all of these examples assumes you have "/dn/*" is your web.xml configuration.

### 188.4.1 Insert a new object of class using application identity

This inserts a Greeting object. The returned object will have the "id" field set.

```
POST http://localhost/dn/mydomain.Greeting
{"author":null,
 "content":"test insert",
 "date":1239213923232}
```

Response:
188.4.2 Insert a new object of class using datastore identity
This inserts a Person object. The returned object will have the "]id" property set.

```
POST http://localhost/dn/mydomain.Person
{
  "firstName": "Joe",
  "lastName": "User",
  "age": 15
}
```

Response:
```
{
  "firstName": "Joe",
  "lastName": "User",
  "age": 15,
  "]id": 2
}
```

188.4.3 Update an object of class using application identity
This updates a Greeting object with id=1, updating the "content" field only.

```
PUT http://localhost/dn/mydomain.Greeting/1
{
  "content": "test update"
}
```

188.4.4 Update an object using datastore identity
This updates a Person object with identity of 2, updating the "age" field only.

```
PUT http://localhost/dn/mydomain.Person/2
{
  "age": 23
}
```

188.4.5 Fetch all objects of class using application identity
This gets the Extent of Greeting objects.

```
GET http://localhost/dn/mydomain.Greeting
```

Response:
188.4.6 Fetch object with id 2 using datastore identity

GET http://localhost/dn/mydomain.Person/2

Response:

```json
{"firstName":"Joe",
 "lastName":"User",
 "age":23,
 "_id":2}
```

Note that it replies with a JSONObject that has "_id" property representing the datastore id.

188.4.7 Query object of class using application identity

This performs the JDOQL query

```
SELECT FROM mydomain.Greeting WHERE content == 'test'
```

GET http://localhost/dn/mydomain.Greeting?content=='test'

Response:

```json
[
{"author":null,
 "content":"test",
 "date":1239213624216,
 "id":1},
{"author":null,
 "content":"test2",
 "date":1239213632286,
 "id":2}]
```

188.4.8 Fetch object using Application PrimaryKey Class (JSON)


Response:
{"class":"google.maps.Markers",
"key":{"class":"com.google.appengine.api.datastore.Key",
"id":1001,
"kind":"Markers"
},
"markers":[
{"class":"google.maps.Marker",
"html":"Paris",
"key":{"class":"com.google.appengine.api.datastore.Key",
"id":1,
"kind":"Marker",
"parent":{"class":"com.google.appengine.api.datastore.Key",
"id":1001,
"kind":"Markers"
}
},
"lat":48.862222,
"lng":2.351111
}]
}
189 Guides

189.1 DataNucleus Access Platform Guides

This section provides a series of worked examples of persistence. If you have any guide that you would like to contribute to be included here please contribute them via the forum.
190 Use with IDEA

190.1 DataNucleus and IntelliJ IDEA

IntelliJ IDEA is one of the most powerful development environments and DataNucleus provides its own plugin for transparently embedding the class enhancer into IDEA's build cycle.

- Features and notes
- Installation
- Upgrade
- Plugin configuration

190.1.1 Features and notes

What this plugin provides:

- Transparent integration of the DataNucleus enhancer into IDEA's build cycle (classes will be enhanced automatically after build)
- Automatic detection of DataNucleus project modules by scanning the classpath for DataNucleus enhancer
- Enabling/Disabling class enhancement per project module
- Enhancement in test output directory (can be deactivated)
- Supports annotation- and file-based metadata
- JDO and JPA support

Notes:

- A project module will only be detected if DataNucleus enhancer is in its classpath
- Enhancement will only work if DataNucleus enhancer -and- its dependencies are in the module-classpath
- Detected Project modules will not be activated automatically, an initial activation via the plugins configuration dialog is necessary
- Configuration dialog: classes subject to enhancement will only be shown after the first build.

190.1.2 Plugin Installation

The DataNucleus plugin is part of the official IntelliJ IDEA plugin repository, hence installing is a trivial task. Select File -> Settings to open the settings dialog and choose Plugins Activate tab Available and scroll or search to DataNucleus Enhancer integration Right-click and select Download and install

After restart go to the settings dialog again, chose DataNucleus Enhancer and activate the relevant project modules. See Plugin configuration.
If necessary, the plugin can also be downloaded from the official JetBrains plugin site for manual installation. Please refer to IDEA’s documentation on how to do that.

190.1.3 Upgrading to a newer version

Updating older plugin versions to 1.0.4 or later does not work, they must be manually uninstalled before upgrading!
Ignore IDEA’s auto-upgrade request in this case. If you’re reading this too late, just uninstall and reinstall the plugin.

190.1.4 Plugin configuration

The DataNucleus IDEA plugin transparently persists it’s configuration into the IDEA project file, hence opening a project will always restore it’s last state.
The plugin configuration dialog can be found under File -> Settings -> DataNucleus Enhancer
- **Enable Enhancer**: Enable/Disable the enhancer for the whole project. Also possible via
  Build -> DataNucleus Enhancer

- **JDO/JPA**: Select the target api to enhance classes for

- **Include Test classes**: If enabled, also test output will be processed

- **Metadata file extensions**: If using metadata files instead of- or additionally to annotations,
  specify the file extensions to search for (e.g. when using JDO with annotations but having orm
  related configuration in separate xml files like 'ClassName.orm' or 'package.orm', then 'orm' has
  to be added here; for JPA 'xml' would have to be added to find a file named 'orm.xml')

- **Add compiler resource patterns**: if enabled, metadata file extensions will be automatically
  added to
  File -> Settings -> Compiler -> Resource patterns
  In case of removing extensions, resource patterns have to be adapted manually

- **Affected Modules**: List of project modules with datanucleus enhancer in classpath and subject
to enhancement if checkbox under
  Enabled
  is active (Use this to activate/deactivate enhancement for specific project modules). Will become
  available after IDEA completed indexing the project.

- **Metadata and annotated classes for enhancement**: List of detected classes for enhancement.
  Will become available after first build (only build target folders are scanned).

---

### Metadata and annotated classes for enhancement

<table>
<thead>
<tr>
<th>Module</th>
<th>Class</th>
<th>File</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>jpa</td>
<td></td>
<td>stfusjon:jpa:jpa3 ORM</td>
<td>C:/Users/ger/IdeaProjects/jpa/target/test-dr</td>
</tr>
<tr>
<td>jpa</td>
<td></td>
<td>stfusjon:jpa:jpa3 ORM</td>
<td>C:/Users/ger/IdeaProjects/jpa/target/test-dr</td>
</tr>
<tr>
<td>jpa</td>
<td></td>
<td>stfusjon:jpa:jpa3 ORM</td>
<td>C:/Users/ger/IdeaProjects/jpa/target/test-dr</td>
</tr>
<tr>
<td>jpa</td>
<td></td>
<td>stfusjon:jpa:jpa3 ORM</td>
<td>C:/Users/ger/IdeaProjects/jpa/target/test-dr</td>
</tr>
<tr>
<td>jpa</td>
<td></td>
<td>stfusjon:jpa:jpa3 ORM</td>
<td>C:/Users/ger/IdeaProjects/jpa/target/test-dr</td>
</tr>
</tbody>
</table>
191 Use with Netbeans

191.1 DataNucleus and NetBeans 7.0
Perhaps the most important step in developing applications with DataNucleus is the enhancement of compiled classes. NetBeans provides a convenient way of integrating this procedure into the build process without the need for any additional tool or plugin. This is possible because NetBeans has native integration with Maven and Ant. Any DataNucleus project based on Maven will open and run as is with NetBeans. No changes are needed in the project nor in NetBeans. When using Ant Class enhancement thus becomes a simple matter of adding a new task to the existing build.xml generated by NetBeans.

This tutorial shows how to integrate DataNucleus with NetBeans 7.0 to simplify the development of JDO applications. Please contribute any updates to this guide that you have since the developers of DataNucleus don't use NetBeans.

191.1.1 Requirements
The following components are required to complete this tutorial successfully:

- DataNucleus AccessPlatform
- DataNucleus Samples
- NetBeans
- Hsqldb

191.1.2 Working with Maven-based DataNucleus Project
NetBeans has native integration with Maven. If your project builds with Maven then it will also build with NetBeans without any change. Unzip the sample project to $home/DNSamples/datanucleus-samples-jdo-tutorial-3.0

The sample project comes with a pom.xml. When you try to open this project NetBeans will automatically detect it as maven project.

Start NetBeans
Open Maven Project

Clean Build the project
You will see a success message as below

```
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time: 5.406s
Finished at: Fri Aug 05 09:23:53 IST 2011
Final Memory: 14M/108M
------------------------------------------------------------------------
```

**Run Main Class**
The results can be seen in the output window
DataNucleus AccessPlatform with JDO
=======================================================================
Persisting products
Product and Book have been persisted

Retrieving Extent for Products
>> Book : JRR Tolkien - Lord of the Rings by Tolkien
>> Product : Sony Discman [A standard discman from Sony]

Executing Query for Products with price below 150.00
>> Book : JRR Tolkien - Lord of the Rings by Tolkien

Deleting all products from persistence
Deleted 2 products

End of Tutorial
=======================================================================
BUILD SUCCESS
=======================================================================
Total time: 6.221s
Finished at: Fri Aug 05 09:27:43 IST 2011
Final Memory: 7M/106M
=======================================================================

191.1.3 Setting up NetBeans for DataNucleus ANT use

The first thing to do is to register the DataNucleus components in the Library Manager of NetBeans so that these become available to any project created with the IDE. This involves creating a new library and adding the JAR files to it, as shown in the following screenshots

Open Library Manager

Add New Library with Name DN3 - Repeat the process for DN3Deps
Add all the libraries under $home/datanucleus-accessplatform-full-deps-3.0.0-release/lib to DN3.
Add all the libraries under $home/datanucleus-accessplatform-full-deps-3.0.0-release/deps to DN3Deps
Once this is done, the NetBeans 7 will add the JAR files to the classpath whenever the newly-created DN3 and DN3Deps library is selected for a project.

191.1.4 Setting up a new project

Delete pom.xml from the downloaded samples project. NetBeans treats any folder with pom.xml as a maven project. Copy the datanucleus.properties to src/java in the downloaded samples project.

Now create a new project from existing sources.
Remember to Select **Java Project with Existing Source**
Click Next and Add Folder. The source folder should point to `datanucleus-samples-jdo-tutorial-3.0/src/java`
Then click Next and Finish

Now we have the sucessfully created the sample project using Ant

Add the Libraries created in first set to this project. Right-Click on Libraries in the Projects Tab
Add DN3 library to your project. Also add DN3Deps
We also need to add JDBC support to our project. Add this using add jars option. Right-Click on Libraries in the Projects Tab and select Add jars.
Add hsqldb

191.1.5 Enhancing the classes

The enhancement process needs to be defined and integrated into the build process. As stated in the introduction, this requires a simple change to the *nbbuild.xml* file.

**TypeSafe Queries** :- Datanucleus generates additonal code for supporting type safe queries. Ensure that “Enable Annotation processing” check box is selected, which is under Project Properties | build | compiling

Click on the **Files** tab, expand the project tree, then open *nbbuild.xml*
Override the *-post-compile* task/target with the following Ant instructions.

```
<target name="-post-compile" depends="init">
  <path id="module.enhancer.classpath">
    <pathelement path="${javac.classpath}"/>
    <pathelement location="${build.classes.dir}"/>
  </path>
  <taskdef name="datanucleusenhancer" classpathref="module.enhancer.classpath" classname="org.datanucleus.enhancer.EnhancerTask"/>
  <echo message="start datanucleusenhancer"/>
  <datanucleusenhancer classpathref="module.enhancer.classpath" dir="${build.classes.dir}" verbose="true">
    <fileset dir="${build.classes.dir}'">
      <include name="**/*.class"/>
    </fileset>
  </datanucleusenhancer>
  <echo message="end datanucleusenhancer"/>
</target>
```

This target is the most convenient for enhancing classes because it occurs just after all classes have been compiled and is called in any case, whether the project is being built, tested or deployed. This ensures that classes are always enhanced. **Ensure that the Compile on Save option is turned off. Enable Annotation Processing should be turned ON**

### 191.1.6 Building the project

The project can now be built, with the knowledge that the classes will be enhanced in the process.
Output window will show something similar to this.
BUILD SUCCESSFUL (total time: 2 seconds)
Run the main class
The output window will show as below
Product and Book have been persisted
Retrieving Extent for Products
>> Book : JRR Tolkien - Lord of the Rings by Tolkien
>> Product : Sony Discman [A standard discman from Sony]
Executing Query for Products with price below 150.00
>> Book : JRR Tolkien - Lord of the Rings by Tolkien
Deleting all products from persistence
Deleted 2 products
End of Tutorial
BUILD SUCCESSFUL (total time: 2 seconds)

191.1.7 Conclusion
This concludes our tutorial on how to integrate DataNucleus with NetBeans 7. As can be seen, thanks to NetBeans project system based on Ant and Native Maven support, development of JDO applications is largely simplified. This tutorial was provided by a user of this software, Kiran Kumar.
192 Use with Ivy

192.1 DataNucleus with Ant and Ivy

192.1.1 Background

This guide introduces Apache Ant Ivy as tool to manage project dependencies for DataNucleus based applications.

In this quick tutorial you will learn how to use Apache Ant Ivy to compile and enhance persistent classes.

192.1.2 Enhancement

DataNucleus enhances persistent classes during compile time. It's a transparent process when using the JDK 1.6 or upper, otherwise use the DataNucleus Ant Enhancer Task (See Ant Guide).

192.1.3 Build files

There are two ivy files and one traditional ant build file:

- ivy.xml
- ivysettings.xml
- build.xml

The ivy.xml declares the project dependencies to DataNucleus libraries and the ivysettings.xml declares the DataNucleus SNAPSHOT repository.

The ivy.xml file:
The *ivysettings.xml* file:

```xml
<ivysettings>
  <settings defaultResolver="resolver"/>
  <resolvers>
    <chain name="resolver">
      <url name="datanucleus-nightly" m2compatible="true">
        <artifact
          pattern="http://www.datanucleus.org/downloads/maven2-nightly/[organisation]/[module]/[revision]/[artifact]-[revision].[ext]" />
      </url>
      <ibiblio name="ibiblio" m2compatible="true"/>
      <filesystem name="repository">
        <ivy pattern="${ivy.settings.dir}/../repository/[organisation]/[module]/[module]-[revision].xml"
          xmlns="http://xmlns.jdash.org/ivy/ivysettings"
        </ivy>
        <artifact pattern="${ivy.settings.dir}/../repository/[organisation]/[artifact]/[artifact]-[revision].[ext]"/>
      </filesystem>
    </chain>
  </resolvers>
</ivysettings>

The *build.xml* file:
<project basedir="." default="default" name="build" xmlns:ivy="antlib:org.apache.ivy.ant">  
  <property environment="env"/>
  <property name="debuglevel" value="source,lines,vars"/>
  <property name="target" value="1.5"/>
  <property name="source" value="1.5"/>
  <property name="bin" value="bin"/>
  <property name="src" value="src"/>
  <property name="dist" value="target"/>

  <target name="default" depends="retrieve,compile,jar,publish"></target>

  <target name="retrieve" description="retrieve dependencies with Ivy">
    <ivy:retrieve/>
    <ivy:cachepath pathid="build.path" conf="build" />  
    <ivy:cachepath pathid="enhancer.path" conf="enhancer"/>
  </target>

  <target name="publish">
    <ivy:cleancache />
    <ivy:publish deliverivypattern="ivy.xml" resolver="repository" overwrite="true">
      <artifacts pattern="${dist}/${artifact}-${revision}.${ext}"/>
    </ivy:publish>
  </target>

  <target name="compile">
    <delete dir="${bin}" failonerror="true"></delete>
    <mkdir dir="${bin}">
      <copy todir="${bin}">
        <fileset dir="${src}" includes="**/**.jdo"/>
      </copy>
    </mkdir>
    <!-- using JDK 1.6 and datanucleus-core will enhance the classes automatically -->
    <!-- MUST use fork, so datanucleus ensures that classes are enhanced before going ahead with the build -->
    <javac srcdir="${src}" destdir="${bin}"
      source="${source}"
      target="${target}"
      debuglevel="${debuglevel}"
      debug="true"
      fork="true">
      <classpath refid="build.path"/>
      <classpath refid="enhancer.path"/>
    </javac>
  </target>

  <target name="jar">
    <delete dir="${dist}" failonerror="false"></delete>
    <mkdir dir="${dist}">
      <jar destfile="${dist}/${ivy.module}-${ivy.revision}.jar">
        <fileset dir="${src}" includes="**/**.jdo"/>
        <fileset dir="${bin}"/>
      </jar>
    </target>
  </target>
</project>
193 Enhancing with Ant

193.1 Enhancing with Ant

193.1.1 Background

Enhancement is a JDO standard process to inject code into the persistent class. The injected code provides the basis of the solid and transparent object persistence specified by JDO.

In this quick tutorial you will learn how to use Apache Ant to compile and enhance persistent classes.

193.1.2 Enhancement

DataNucleus provides an Ant task to enhance files and all you have to do is set up some parameters, but before enhancing files with the DataNucleus Enhancer for Ant, let's look at a complete Ant build file specially created to compile, enhance and generate archive files (jar, war).

193.1.3 Enhancer Task

The Ant build file provided here as example is divided in many targets as like clean, compile, enhance, distribution and others. The one we are interested is the "enhance".

This first section of the target "enhance" defines the classpath to use for enhancement. In this classpath we put the datanucleus-core.jar + datanucleus-api-jdo.jar datanucleus-api-jpa.jar DataNucleus dependencies + Your Persistent Classes (.class) + Your JDO Classes (.jdo) + Your dependencies.

```xml
<!-- the classpath to enhance -->
<path id="module.enhancer.classpath">
  <pathelement location="${module.classes.dir}"/>
  <path refid="module.lib.classpath"/>
</path>
```

In the second section, we define the DataNucleus Enhancer task.

```xml
<!-- define the task enhancer -->
<taskdef name="nucleusenhancer"
  classpathref="module.enhancer.classpath"
  classname="org.datanucleus.enhancer.EnhancerTask"/>
```
With the DataNucleus Enhancer task defined, we are now able to use it. It's important to correctly define the classpath. See this reference for further information on attributes that can be configured for the enhancer.

```xml
<!-- enhance -->
<nucleusenhancer classpathref="module.enhancer.classpath"/>
```

### 193.1.4 Full example

This example is composed of three files, `build.xml`, `module.properties` and `module.xml`. In the file `build.xml` we load the properties for the module and call the `distribution` target defined in the `module.xml` file, which compiles, enhance and create an archive file. In the `module.properties` file we define the variables to point to the location of files.
### module.properties
#
# Name=my-module-name
# name=${Name}
# version=1.0
#
# build.default.target = distribution
#
# project.build.debug=on
# project.lib.dir=lib
#
# module.dir=./WebContent
# module.lib.dir=${module.dir}/WEB-INF/lib
# module.lib2.dir=/home/user/lib
# module.classes.dir=${module.dir}/WEB-INF/classes
# module.src.java.dir=./src/java
# module.build.debug=on
# module.archive.dist.dir=dist
# module.archive.dist.file=${name}.war
# module.archive.files=**/*
#
# module.lib.dir=./lib
# module.lib2.dir=/home/user/lib
# module.classes.dir=./classes
# module.src.java.dir=./src
# module.build.debug=on
# module.archive.dist.dir=dist
# module.archive.dist.file=${name}.jar
# module.archive.files=**/*
<project name="project" default="default">
  <import file="module.xml"/>

  <!--
  environment
  -->
  <property environment="env"/>
  <property name="project.location" location="."/>

  <!--
  TARGET : default
  -->
  <target name="default">
    <echo message="Welcome to the build."
    <echo message="Welcome to the build."
    <echo message="Welcome to the build."
    <property file="${project.location}/module.properties"/>
    <antcall target="distribution"/>
  </target>
</project>
<!--
===================================================================
module.xml
===================================================================
-->
<project name="module" default="distribution">
  
  <!--
  environment
  -->
  
  <property environment="env"/>
  <property name="project.location" location="."/>
  
  <!--
  TARGET : default
  -->
  
  <target name="default">
    <echo message="Welcome to the build."/>
    <echo message="Welcome to the module."/>
    <antcall target="${build.default.target}"/>
  </target>

  <!--
  CONFIGURATION: MODULE
  -->

  <!--
  libs necessary to build the module -->
  <path id="module.lib.classpath">
    <fileset dir="${module.lib.dir}"/>
    <include name="**/*.jar"/>
  </fileset>

  <fileset dir="${module.lib2.dir}"/>
  <include name="**/*.jar"/>
  
  </path>

  <!--
  the classpath to compile -->
  <path id="module.compile.classpath">
    <pathelement location="${module.classes.dir}"/>
    <path refid="module.lib.classpath"/>
  </path>

  <!--
  TARGET : clean
  -->

  <target name="clean">
    <delete includeEmptyDirs="true" quiet="true">
      <fileset dir="${module.classes.dir}" includes="**/*.class,**/*.properties,**/*.*"/>
    </delete>
  </target>
</project>
194 Jobfilter - JDO/JPA web app

194.1 JobFilter

Jobfilter is a simple Spring-based webapp for managing and filtering job vacancies and application. Vacancy information is obtained from an external data source - currently www.jobserve.com-mailed CSV files; however the implementation is designed to be extensible to other sources of vacancy information.

Vacancies are handled via a simple workflow: a vacancy begins life (having been loaded from the external data source) in a new state. An initial filtering is performed on all new vacancies whereby they are either accepted as being of potential interest or rejected. New vacancies which are accepted become pending. A pending vacancy can be edited repeatedly and, at each stage, have notes attached; it can also be rejected if, on closer inspection, it is deemed to be of no further interest. If a pending vacancy is of interest, an application can be made for the vacancy. Vacancy applications require a CV (Resume) and Jobfilter maintains a set of available CVs for this purpose. An application can be made by email (using an email address provided in the vacancy data) or via an external website (using a URL provided in the vacancy data). In the latter case, Jobfilter merely maintains a record of the vacancy application but leaves the application process to the external website.

Vacancies which are rejected (at either the new or pending stages) are maintained internally within the system for a period of time after the rejection. This allows the system to spot and remove duplicates of rejected vacancies which might appear in subsequent vacancy data loads. A timer within the Jobfilter application automatically removes rejected vacancies after a period of time (when it is assumed duplicates will no longer appear).

The data comprising a vacancy is divided into data specific to the vacancy itself, data pertaining to the agent who posted the vacancy and data about the agency to which that agent belongs. Jobfilter maintains a record of all agencies and their agents and provides the ability to edit this information. Additionally a record is maintained of CVs sent to each agency for reference in the event of a later application for a different vacancy posted by the same agency.

You can download this sample from SourceForge.
194.2 Implementation

194.2.1 Architecture

Jobfilter is implemented as a simple 3-tier web application. The web tier is implemented using Spring web MVC which communicates with a middle tier via a set of service interfaces whose implementations are dependency injected into the web tier using Spring. Spring is also used to provide transactional behaviour on the service interfaces. Service interface implementations communicate with the persistence tier via a set of Data Access Object (DAO) interfaces whose implementations are dependency injected into the middle tier using Spring. The following DAO implementations are provided:

- Hibernate3 (included for completeness)
- JDO (DataNucleus)
- JPA (DataNucleus)

194.2.2 Projects

The Jobfilter implementation is spread across a number of projects as follows:

- jobfilter-toplevel
  Project whose sole purpose is for invoking a Maven build of the entire application. This project does not provide any code nor does it provide any resultant artifact.
- jobfilter-doc
  Project documentation - README's, build and deployment instructions etc.
- jobfilter-common
  Provides domain classes, persistence layer and middle tier (business logic). Note that the jobfilter-common codebase includes the sources for all variants of the persistence layer (JDO, JPA, Hibernate3 etc.) - an appropriate implementation will be selected according to a property passed to the Maven build.
- jobfilter-war
  Web front-end - servlets, JSPs, Spring MVC form controllers, web-related decorator classes etc. Also provides top-level Spring configuration of application, linking together web tier, business logic and DAOs.

194.2.3 Object Model

The Jobfilter object model is given in the UML class diagram below:
The classes in the diagram correspond to those in package com.plus.fcentre.jobfilter.domain (in project jobfilter-common). A description of the classes and their relationships follows:

- **Agency** - An agency employing a number of agents and holding the registrations of a number of CVs.
- **Agent** - An agent employed by an agency and who is handing a number of vacancies.
- **Vacancy** - Abstract base class for a vacancy being handled by an agent.
- **NewVacancy** - A vacancy in the new state. New vacancies have not undergone the initial filtering process.
- **PendingVacancy** - A vacancy in the pending state. Pending vacancies have successfully passed the initial vacancy filtering process and may or may not be associated with a vacancy application. A series of notes may be attached to a pending vacancy, recording events during its lifecycle.
- **VacancyNote** - A note associated with a pending vacancy. Vacancy notes are timestamped and contain a textual description of some event during the lifecycle of its associated vacancy - e.g. "Called agent to check on progress of application - still awaiting response from client."
- **RejectedVacancy** - A vacancy in the rejected state. Rejected vacancies are new or pending vacancies which have ultimately been determined to be of no further interest. Rejected vacancies do not appear in the user interface but are retained by the system for a short time to help prevent the uploading of future, duplicate vacancies.
- **CV** - A Curriculum Vitae (Resume) held by the system. The set of available CVs is presented to the user upon making an application for a vacancy. A CV object contains a binary image of the original CV file uploaded from filestore and a reference to an entity reflecting the type of the contents of the binary image (e.g. "Word document"). Any given CV may be registered with any number of agencies - note that, in the CV domain class, the relationship from a CV to a CV
registration is actually represented by a map of CVRegistration keyed by the agency with which the CV is registered rather than a simple list or set.

- **CVContentType** - The type of the content of a CV binary held by a CV object. CV content types are static data, having a fixed set of values (currently "PDF" and "Word"). These are created in the database when the application starts up for the first time.

- **CVRegistration** - The registration of a given CV with a given agency. A CV registration can be used as the basis of any number of vacancy applications - i.e. upon registering a CV with an agency, it is possible to use that CV to apply for any number of vacancies advertised by that agency.

- **VacancyApplication** - An application for a (pending) vacancy using a given CV registered with a given agency (i.e. a CV registration entity).

### 194.2.4 Database model

The tables of the Jobfilter database model (into which the Object Model is persisted) are described below. Note that, for brevity, this description omits the majority of the value fields of the various tables and, instead, concentrates on the various primary and foreign key fields.

- **agency**
  - id - primary key

- **agent**
  - id - primary key
  - agency_id - id of agency to which agent belongs

- **vacancy**
  - id - primary key
  - agent_id - id of agent posting vacancy
  - subtype - discriminator field indicating vacancy subtype

- **vacancy_note**
  - id - primary key
  - vacancy_id - id of (pending) vacancy to which note applies

- **cv**
  - id - primary key
  - content_type_id - id of CV content type
  - content - BLOB field containing CV file binary image

- **cv_content_type**
  - id - primary key

- **cv_registration**
  - agency_id - id of agency with which CV has been registered
  - cv_id - id of registered CV

- **vacancy_application**
  - id - primary key
  - agency_id - id of agency posting vacancy being applied for
  - cv_id - id of CV used for application
• vacancy_id - id of (pending) vacancy being applied for