JDO Mapping Guide (v5.0)
# Table of Contents

Classes ................................................................. 4  
  Persistence Capable Classes ................................. 4  
  Persistence-Aware Classes ................................. 4  
  Read-Only Classes ........................................ 5  

Inheritance ............................................................ 6  
  Discriminator ................................................ 7  
  New Table ....................................................... 8  
  Subclass table ............................................... 10  
  Superclass table ........................................... 11  
  Complete table ............................................... 14  
  Retrieval of inherited objects ............................ 15  

Auditing ............................................................... 17  

Fields/Properties .................................................... 18  
  Persistent Fields ........................................... 18  
  Persistent Properties ....................................... 18  
  Overriding Superclass Field/Property MetaData ....... 19  
  Field/Property positioning ................................ 20  
  Making a field/property read-only ...................... 20  

Field Types .......................................................... 22  
  Primitive and java.lang Types .......................... 23  
  java.math types ............................................ 24  
  Temporal Types (java.util, java.sql, java.time, Jodatime) .......... 24  
  Collection/Map types ..................................... 26  
  Enums .......................................................... 27  
  Geospatial Types ........................................... 29  
  Other Types .................................................. 34  
  Arrays ........................................................ 35  
  Generic Type Variables .................................... 36  
  JDO Attribute Converters .................................. 37  
  Types extending Collection/Map ......................... 39  
  TypeConverters ............................................. 41  

Identity .............................................................. 42  
  datastore Identity .......................................... 42  
  Application Identity ....................................... 45  
  Nondurable Identity ....................................... 50  
  Compound Identity Relationships ....................... 51  

Versioning .......................................................... 63  
  Versioning using a surrogate column ................... 63
Value Generation .......................................................... 65

1-N Relations 

1-1 Relations 

Unidirectional with JoinTable 

Unidirectional with ForeignKey 

Map<PC,Simple> Unidirectional FK (value stored in key) 

Map<Simple, Simple> using AttributeConverter via column 

Map<Simple,PC> using Join Table 

Map<PC,PC> using Join Table 

Collection<PC> via Shared FK 

Collection<Simple> using AttributeConverter via column 

Collection<Simple> via JoinTable 

Using a List 

Collection<PC> Bidirectional FK 

Collection<PC> Bidirectional JoinTable 

Collection<PC> Unidirectional FK 

Collection<PC> Unidirectional JoinTable 

equals() and hashCode(). .............................................. 84

Collection<PC> Unidirectional JoinTable .................................. 84

Collection<PC> Unidirectional FK ........................................ 86

Collection<PC> Bidirectional JoinTable .................................. 87

Collection<PC> Bidirectional FK .......................................... 89

Using a List ........................................................................ 91

Collection<Simple> via JoinTable ........................................ 91

Collection<Simple> using AttributeConverter via column .............. 92

Collection<PC> via Shared JoinTable ..................................... 93

Collection<PC> via Shared FK ........................................... 96

Map<PC,PC> using Join Table ............................................. 97

Map<Simple,PC> using Join Table ....................................... 100

Map<PC,Simple> using Join Table ....................................... 101

Map<Simple, Simple> using Join Table ................................ 101

Map<Simple, Simple> using AttributeConverter via column ........... 103

Map<Simple,PC> Unidirectional FK (key stored in value) ............... 104

Map<Simple,PC> Unidirectional FK (key stored in value) ............... 106

Map<PC,Simple> Unidirectional FK (value stored in key) ............... 107

N-1 Relations ................................................................... 109

Unidirectional with ForeignKey ............................................ 109

Unidirectional with JoinTable ............................................ 110

Versioning using a field/property of the class ................................. 63

native ............................................................................ 66

sequence ........................................................................ 66

identity ......................................................................... 67

increment ...................................................................... 69

uuid-string .................................................................. 70

uuid-hex ....................................................................... 71

datastore-uuid-hex .......................................................... 72

max ................................................................................ 72

uuid .............................................................................. 73

uuid-object .................................................................... 74

auid .............................................................................. 75

timestamp ..................................................................... 75

timestamp-value .............................................................. 76

Standalone ID generation ....................................................... 76

1-1 Relations ................................................................... 79

Unidirectional .................................................................. 79

Bidirectional .................................................................... 80

1-N Relations .................................................................. 83

equals() and hashCode(). .............................................. 84

Collection<PC> Unidirectional JoinTable .................................. 84

Collection<PC> Unidirectional FK ........................................ 86

Collection<PC> Bidirectional JoinTable .................................. 87

Collection<PC> Bidirectional FK .......................................... 89

Using a List ........................................................................ 91

Collection<Simple> via JoinTable ........................................ 91

Collection<Simple> using AttributeConverter via column .............. 92

Collection<PC> via Shared JoinTable ..................................... 93

Collection<PC> via Shared FK ........................................... 96

Map<PC,PC> using Join Table ............................................. 97

Map<Simple,PC> using Join Table ....................................... 100

Map<PC,Simple> using Join Table ....................................... 101

Map<Simple, Simple> using Join Table ................................ 101

Map<Simple, Simple> using AttributeConverter via column ........... 103

Map<Simple,PC> Unidirectional FK (key stored in value) ............... 104

Map<Simple,PC> Unidirectional FK (key stored in value) ............... 106

Map<PC,Simple> Unidirectional FK (value stored in key) ............... 107

N-1 Relations ................................................................... 109

Unidirectional with ForeignKey ............................................ 109

Unidirectional with JoinTable ............................................ 110
To implement a persistence layer with JDO you firstly need to map the classes and fields/properties that are involved in the persistence process. This can be as simple as marking the classes as an @PersistenceCapable, or you can configure down to the fine detail of precisely what schema it maps on to. The following sections deal with the many options available. This guide takes you through the many metadata options.

When mapping a class for JDO you can make use of metadata. This metadata can be Java annotations, or can be XML metadata, or a mixture of both, or you could even define it using a dynamic API. This is very much down to your own personal preference but we try to present both ways here.

- We advise trying to keep schema information out of annotations, so that you avoid tying compiled code to a specific datastore. That way you retain datastore-independence. This may not be a concern for your project however.

- Whilst the JDO spec only allows you to specify your mapping information using JDO metadata (annotations, or JDO/ORM XML metadata, or JDO Metadata API), it also allows you the option of using JPA metadata (annotations or JPA XML metadata). This is provided as a way of easily migrating across to JDO from JPA, for example. Consult the DataNucleus JPA mappings docs for details.

In terms of the relative priority of annotations, JDO XML and ORM XML metadata, the following figure highlights the process
So you can provide the metadata via annotations solely, or via annotations plus ORM XML Metadata overrides, or via JDO XML Metadata solely, or via JDO XML Metadata plus ORM XML Metadata overrides, or finally via a Metadata API.

If you are using XML overrides for ORM, this definition will be merged in to the base definition (annotations or JDO XML Metadata). Note that you can utilise annotations for one class, and then JDO XML Metadata for another class should you so wish.

One further alternative is if you have annotations in your classes, you provide JDO XML Metadata (package.jdo), and also ORM XML Metadata (package-
In this case the annotations are the base representation, applying overrides from JDO XML Metadata, and then overrides from the ORM XML Metadata.

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 schema 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).
Classes

We have the following types of classes in DataNucleus JDO.

- **PersistenceCapable** - persistable class with full control over its persistence.
- **PersistenceAware** - a class that is not itself persisted, but that needs to access internals of persistable classes.

In strict JDO all persistable classes need to have a default constructor. With DataNucleus JDO this is not necessary, since all classes are enhanced before persistence and the enhancer adds on a default constructor if one is not defined.

### Persistence Capable Classes

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. Like this

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

or using XML metadata

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

See also:

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

If any of your other classes **access the fields of these persistable classes directly** then these other classes should be defined as **PersistenceAware**.

### Persistence-Aware Classes

If a class is not itself persistable but it interacts with *fields* of persistable classes then it should be marked as **Persistence Aware**. You do this as follows
@PersistenceAware
class MyClass {
    ...
}

or using XML metadata
<class name="MyClass" persistence-modifier="persistence-aware">
    ...
</class>

See also :-
- Annotations reference for @PersistenceAware

**Read-Only Classes**

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

@PersistenceCapable
@Extension(vendorName="datanucleus", key="read-only", value="true")
class MyClass {
    ...
}

or using XML metadata
<class name="MyClass">
    ...
    <extension vendor-name="datanucleus" key="read-only" value="true"/>
</class>
Inheritance

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.

- 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.
- 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.
- 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.
- The final way is for all classes in an inheritance tree to have 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 the persistence 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.

At runtime, when you start up your PersistenceManagerFactory, JDO will only know about the classes that the persistence API has been introduced to via method calls. To alleviate this, particularly for subclasses of classes in an inheritance relationship, you should make use of one of the many available Auto Start Mechanisms.

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>` element
- Annotations reference for `@Inheritance`
- Annotations reference for `@Discriminator`

**Discriminator**

Applicable to RDBMS, HBase, MongoDB, Cassandra

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

```
<class name="Product">
  <inheritance>
    <discriminator strategy="class-name"/>
  </inheritance>
  ...
</class>
```

or with annotations
@PersistenceCapable
@Discriminator(strategy=DiscriminatorStrategy.CLASS_NAME)

public class Product {...}

Alternatively if using value-map strategy then you need to provide the value map for all classes in the inheritance tree that will be persisted in their own right.

@PersistenceCapable
@Discriminator(strategy=DiscriminatorStrategy.VALUE_MAP, value="PRODUCT")

public class Product {...}

@PersistenceCapable
@Discriminator(value="BOOK")

public class Book {...}

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
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.

**Subclass table**

Applicable to RDBMS

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

```xml
<class name="AbstractProduct">  
  <inheritance strategy="subclass-table"/>  
  <field name="id" primary-key="true">  
    <column name="PRODUCT_ID"/>  
  </field>  
  ...
</class>
```

or with annotations
@PersistenceCapable
@Inheritance(strategy=InheritanceStrategy.SUBCLASS_TABLE)
public class AbstractProduct {...}

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 fully 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.

**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...
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.

```
<table>
<thead>
<tr>
<th>PRODUCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>+PRODUCT_ID</td>
</tr>
<tr>
<td>PRICE</td>
</tr>
<tr>
<td>NAME</td>
</tr>
<tr>
<td>DESCRIPTION</td>
</tr>
<tr>
<td>AUTHOR</td>
</tr>
<tr>
<td>TITLE</td>
</tr>
<tr>
<td>COUNTRY</td>
</tr>
<tr>
<td>ARTIST</td>
</tr>
<tr>
<td>DISCTITLE</td>
</tr>
<tr>
<td>PRODUCT_TYPE</td>
</tr>
</tbody>
</table>
```

In the above example, when we insert a TravelGuide object into the datastore, a row will be inserted into the PRODUCT table only.

JDO allows two types of discriminators. The example above used a discriminator strategy of class-name. 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 :-

```java
// MetaData for Book
Book
  discriminator strategy: class-name

// MetaData for CompactDisc
CompactDisc
  discriminator strategy: value-map
```
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.

**Complete table**

Applicable to RDBMS, Neo4j, NeoDatis, Excel, OOXML, ODF, HBase, Cassandra, 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
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

![Datastore Schema](image)

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.**

### Retrieval of inherited objects

JDO provides particular mechanisms for retrieving inheritance trees. These are accessed via the Extent/Query interface. Taking our example above, we can then do
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.
Auditing

Applicable to RDBMS

With standard JDO you have no annotations available to automatically add timestamps into the datastore against each record when it is persisted or updated. Whilst you can do this manually, setting the field(s) in prePersist callbacks etc, DataNucleus provides some simple annotations to make it simpler still.

```java
import org.datanucleus.api.jdo.annotations.CreateTimestamp;
import org.datanucleus.api.jdo.annotations.UpdateTimestamp;

@PersistenceCapable
public class Hotel {
    @CreateTimestamp
    Timestamp createTimestamp;

    @UpdateTimestamp
    Timestamp updateTimestamp;

    ...
}
```

In the above example we have 2 fields in the class that will have columns in the datastore. The field `createTimestamp` will be persisted at INSERT with the Timestamp of the insert. The field `updateTimestamp` will be persisted whenever any update is made to the object in the datastore, with the Timestamp of the update.
Fields/Properties

Once we have defined a class to be persistable, we need to define how to persist the different fields/properties that are to be persisted. There are two distinct modes of persistence definition; the most common uses fields, whereas an alternative uses properties.

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.

Almost all Java field types are default persistent (if DataNucleus knows how to persist a type then it defaults to persistent) so there is no real need to specify @Persistable to make the field persistent.

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 @Persistence 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.

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>
```

### 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.
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:

```xml
<jdo>
  <package name="mydomain">
    <class name="Person" detachable="true" table="People">
      <field name="personNum">
        <column position="0"/>
      </field>
      <field name="firstName">
        <column position="1"/>
      </field>
      <field name="lastName">
        <column position="2"/>
      </field>
    </class>
  </package>
</jdo>
```

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;
}
```

Making a field/property read-only

If you want to make a member read-only you can do it like this.
<jdo>
  <package name="mydomain">
    <class name="MyClass">
      <field name="myField">
        <extension vendor-name="datanucleus" key="insertable" value="false"/>
        <extension vendor-name="datanucleus" key="updateable" value="false"/>
      </field>
    </class>
  </package>
</jdo>

and with Annotations

```java
@PersistenceCapable
public class MyClass {
    @Extension(vendorName="datanucleus", key="insertable", value="false")
    @Extension(vendorName="datanucleus", key="updateable", value="false")
    String myField;
}
```

alternatively using a DataNucleus convenience annotation

```java
import org.datanucleus.api.jdo.annotations.ReadOnly;

@PersistenceCapable
public class MyClass {
    @ReadOnly
    String myField;
}
```
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

- **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 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

- **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 sections below shows the currently supported SCO java types in DataNucleus. The tables in these sections show
  - **default-fetch-group (DFG)**: whether the field is retrieved by default when retrieving the object itself
  - **proxy**: 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

With DataNucleus, all types that we have a way of persisting (i.e listed below) are default persistent (meaning that you don't need to annotate them in any way to persist them). The only field types where this is not always true is for java.lang.Object, some Serializable types, array of persistables, and java.io.File so always safer to mark those as persistent.

Where you have a secondary type that can be persisted in multiple possible ways you select which column type(s) by using the **jdbc-type** for the field, or alternatively you find the name of the internal DataNucleus **TypeConverter** and use that via the metadata extension "type-converter-name".

**Plugin**

If you have support for any additional types and would either like to contribute them, or have them listed here, let us know. Supporting a new type is easy, typically involving a **JDO AttributeConverter** if you can easily convert the type into a String or Long. See also the **Java Types plugin-point**. You can also define more specific support for it with RDBMS datastores - 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.

**Primitive and java.lang Types**

All primitive types and wrappers are supported and will be persisted into a single database "column". Arrays of these are also supported, and can either be serialised into a single column, or persisted into a join table (dependent on datastore).

<table>
<thead>
<tr>
<th>Java Type</th>
<th>DFG?</th>
<th>Proxy?</th>
<th>PK?</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>Persisted as <strong>BOOLEAN</strong>, Integer (i.e 1,0), String (i.e 'Y','N').</td>
</tr>
<tr>
<td>byte</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>short</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>java.lang.Boolean</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>Persisted as <strong>BOOLEAN</strong>, Integer (i.e 1,0), String (i.e 'Y','N').</td>
</tr>
<tr>
<td>java.lang.Byte</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>java.lang.Character</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>java.lang.Double</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>java.lang.Float</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>java.lang.Integer</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>java.lang.Long</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>java.lang.Short</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>java.lang.Number</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>Persisted 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.</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Java Type</td>
<td>DFG?</td>
<td>Proxy?</td>
<td>PK?</td>
<td>Comments</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------</td>
<td>--------</td>
<td>-----</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>java.lang.StringBuffer</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>Persisted as String. The dirty check mechanism for this type is limited to immutable mode, which 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.</td>
</tr>
<tr>
<td>java.lang.StringBuilder</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>Persisted as String. The dirty check mechanism for this type is limited to immutable mode, which 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.</td>
</tr>
<tr>
<td>java.lang.Class</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>Persisted as String.</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>Persisted as DOUBLE or String. String can be used to retain precision.</td>
</tr>
<tr>
<td>java.math.BigInteger</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>Persisted as INTEGER or String. String can be used to retain precision.</td>
</tr>
</tbody>
</table>

**java.math types**

BigInteger and BigDecimal are supported and persisted into a single numeric column by default.

<table>
<thead>
<tr>
<th>Java Type</th>
<th>DFG?</th>
<th>Proxy?</th>
<th>PK?</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.math.BigDecimal</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>Persisted as DOUBLE or String. String can be used to retain precision.</td>
</tr>
<tr>
<td>java.math.BigInteger</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>Persisted as INTEGER or String. String can be used to retain precision.</td>
</tr>
</tbody>
</table>

**Temporal Types (java.util, java.sql, java.time, Jodatime)**

DataNucleus supports a very wide range of temporal types, with flexibility in how they are persisted.

<table>
<thead>
<tr>
<th>Java Type</th>
<th>DFG?</th>
<th>Proxy?</th>
<th>PK?</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.sql.Date</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>Persisted as DATE, String, DATETIME or Long.</td>
</tr>
<tr>
<td>java.sql.Time</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>Persisted as TIME, String, DATETIME or Long.</td>
</tr>
<tr>
<td>java.sql.Timestamp</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>Persisted as TIMESTAMP, String or Long.</td>
</tr>
<tr>
<td>Java Type</td>
<td>DFG?</td>
<td>Proxy?</td>
<td>PK?</td>
<td>Comments</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------</td>
<td>--------</td>
<td>-----</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>java.util.Calendar</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>Persisted as <strong>TIMESTAMP (inc Timezone)</strong>, DATETIME, String, or as (Long, String) storing millis + timezone respectively</td>
</tr>
<tr>
<td>java.util.GregorianCalendar</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>Persisted as <strong>TIMESTAMP (inc Timezone)</strong>, DATETIME, String, or as (Long, String) storing millis + timezone respectively</td>
</tr>
<tr>
<td>java.util.Date</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Persisted as <strong>DATETIME</strong>, String or Long.</td>
</tr>
<tr>
<td>java.util.TimeZone</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>Persisted as String.</td>
</tr>
<tr>
<td>java.time.LocalDateTime</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Persisted as <strong>Timestamp</strong>, String, or DATETIME.</td>
</tr>
<tr>
<td>java.time.LocalTime</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Persisted as <strong>TIME</strong>, String, or Long.</td>
</tr>
<tr>
<td>java.time.LocalDate</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Persisted as <strong>DATE</strong>, String or DATETIME.</td>
</tr>
<tr>
<td>java.time.OffsetDateTime</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Persisted as <strong>Timestamp</strong>, String, or DATETIME.</td>
</tr>
<tr>
<td>java.time.OffsetTime</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Persisted as <strong>TIME</strong>, String, or Long.</td>
</tr>
<tr>
<td>java.time.MonthDay</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Persisted as <strong>String</strong>, DATE, or as (Integer,Integer) with the latter being month+day respectively.</td>
</tr>
<tr>
<td>java.time.YearMonth</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Persisted as <strong>String</strong>, DATE, or as (Integer,Integer) with the latter being year+month respectively.</td>
</tr>
<tr>
<td>java.time.Year</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Persisted as <strong>Integer</strong>, or String.</td>
</tr>
<tr>
<td>java.time.Period</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Persisted as <strong>String</strong>.</td>
</tr>
<tr>
<td>java.time.Instant</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Persisted as <strong>TIMESTAMP</strong>, String, Long, or DATETIME.</td>
</tr>
<tr>
<td>java.time.Duration</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Persisted as <strong>String</strong>, Double (secs.nanos), or Long (secs).</td>
</tr>
<tr>
<td>java.time.ZoneId</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Persisted as <strong>String</strong>.</td>
</tr>
<tr>
<td>java.time.ZoneOffset</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Persisted as <strong>String</strong>.</td>
</tr>
<tr>
<td>java.time.ZonedDateTime</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Persisted as <strong>Timestamp</strong>, or String.</td>
</tr>
<tr>
<td>org.joda.time.DateTime</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Requires <strong>datanucleus-jodatime</strong> plugin. Persisted as <strong>TIMESTAMP</strong> or String.</td>
</tr>
<tr>
<td>org.joda.time.LocalTime</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Requires <strong>datanucleus-jodatime</strong> plugin. Persisted as <strong>TIME</strong> or String.</td>
</tr>
<tr>
<td>org.joda.time.LocalDate</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Requires <strong>datanucleus-jodatime</strong> plugin. Persisted as <strong>DATE</strong> or String.</td>
</tr>
<tr>
<td>Java Type</td>
<td>DFG?</td>
<td>Proxy</td>
<td>PK?</td>
<td>Comments</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>------</td>
<td>-------</td>
<td>-----</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>org.joda.time.LocalDateTime</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>Requires dataNucleus-jodatime plugin. Persisted as TIMESTAMP, or String.</td>
</tr>
<tr>
<td>org.joda.time.Duration</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>Requires dataNucleus-jodatime plugin. Persisted as String or Long.</td>
</tr>
<tr>
<td>org.joda.time.Interval</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>Requires dataNucleus-jodatime plugin. Persisted as String or (TIMESTAMP, TIMESTAMP).</td>
</tr>
</tbody>
</table>

**Collection/Map types**

DataNucleus supports a very wide range of collection, list and map types.

<table>
<thead>
<tr>
<th>Java Type</th>
<th>DFG?</th>
<th>Proxy</th>
<th>PK?</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.util.ArrayList</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>See the 1-N Lists Guide</td>
</tr>
<tr>
<td>java.util.BitSet</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>Persisted as collection by default, but will be stored as String when the datastore doesn’t provide for collection storage</td>
</tr>
<tr>
<td>java.util.Collection</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>See the 1-N Collections Guide</td>
</tr>
<tr>
<td>java.util.HashMap</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>See the 1-N Maps Guide</td>
</tr>
<tr>
<td>java.util.HashSet</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>See the 1-N Collections Guide</td>
</tr>
<tr>
<td>java.util.Hashtable</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>See the 1-N Maps Guide</td>
</tr>
<tr>
<td>java.util.LinkedHashMap</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>Persisted as a Map currently. No List-ordering is supported. See the 1-N Maps Guide</td>
</tr>
<tr>
<td>java.util.LinkedHashSet</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>Persisted as a Set currently. No List-ordering is supported. See the 1-N Collections Guide</td>
</tr>
<tr>
<td>java.util.LinkedList</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>See the 1-N Lists Guide</td>
</tr>
<tr>
<td>java.util.List</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>See the 1-N Lists Guide</td>
</tr>
<tr>
<td>java.util.Map</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>See the 1-N Maps Guide</td>
</tr>
<tr>
<td>java.util.Properties</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>See the 1-N Maps Guide</td>
</tr>
<tr>
<td>java.util.PriorityQueue</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>The comparator is specifiable via the metadata extension comparator-name (see below). See the 1-N Lists Guide</td>
</tr>
<tr>
<td>Java Type</td>
<td>DFG?</td>
<td>Proxy?</td>
<td>PK?</td>
<td>Comments</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------</td>
<td>--------</td>
<td>-----</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>java.util.Queue</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>The comparator is specifiable via the metadata extension comparator-name (see below). See the 1-N Lists Guide</td>
</tr>
<tr>
<td>java.util.Set</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>See the 1-N Collections Guide</td>
</tr>
<tr>
<td>java.util.SortedMap</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>The comparator is specifiable via the metadata extension comparator-name (see below). See the 1-N Maps Guide</td>
</tr>
<tr>
<td>java.util.SortedSet</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>The comparator is specifiable via the metadata extension comparator-name (see below). See the 1-N Collections Guide</td>
</tr>
<tr>
<td>java.util.Stack</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>See the 1-N Lists Guide</td>
</tr>
<tr>
<td>java.util.TreeMap</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>The comparator is specifiable via the metadata extension comparator-name (see below). See the 1-N Maps Guide</td>
</tr>
<tr>
<td>java.util.TreeSet</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>The comparator is specifiable via the metadata extension comparator-name (see below). See the 1-N Collections Guide</td>
</tr>
<tr>
<td>java.util.Vector</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>See the 1-N Lists Guide</td>
</tr>
<tr>
<td>com.google.common.collect.Multiset</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>Requires datanucleus-guava plugin. See the 1-N Collections Guide</td>
</tr>
</tbody>
</table>

**Comparators**

Containers that support a Comparator to order the elements of the set can specify it in metadata like this.

```java
@Element
@Extension(vendorName="datanucleus", key="comparator-name", value="mydomain.model.MyComparator")
SortedSet<MyElementType> elements;
```

When instantiating the SortedSet field it will create it with a comparator of the specified class (which must have a default constructor).

**Enums**

DataNucleus supports persisting Enums, and they can be stored as either the ordinal (numeric column) or name (String column).
A DataNucleus extension to this is where you have an Enum that defines its own "value"s for the different enum options.

A DataNucleus extension is available for RDBMS datastores where you are storing the name of the enum, and to put a CHECK constraint on the column. You specify it like this

```java
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;
    }
}
```

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
@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. You can use this method to persist "int", "short", or "String" types.
Geospatial Types

DataNucleus has extensive support for Geospatial types. The datanucleus-geospatial plugin allows using geospatial and traditional types simultaneously in persistent objects making DataNucleus a single interface to read and manipulate any business data. The implementation of many of these spatial types follows the OGC Simple Feature specification, but adds further types where the datastores support them.

<table>
<thead>
<tr>
<th>Java Type</th>
<th>DFG?</th>
<th>Proxy?</th>
<th>PK?</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.awt.Point</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>Requires datanucleus-geospatial plugin. Persisted as (int, int) on RDBMS, or as String elsewhere.</td>
</tr>
<tr>
<td>java.awt.Rectangle</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>Requires datanucleus-geospatial plugin. Persisted as (int, int, int, int) on RDBMS, or as String elsewhere.</td>
</tr>
<tr>
<td>java.awt.Polygon</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>Requires datanucleus-geospatial plugin. Persisted as (int[], int[], int) on RDBMS, or as String elsewhere.</td>
</tr>
<tr>
<td>java.awt.geom.Line2D</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>Requires datanucleus-geospatial plugin. Persisted as (double, double, double, double) or (float, float, float, float) on RDBMS, or as String elsewhere.</td>
</tr>
<tr>
<td>java.awt.geom.Point2D</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>Requires datanucleus-geospatial plugin. Persisted as (double, double) or (float, float) on RDBMS, or as String elsewhere.</td>
</tr>
<tr>
<td>java.awt.geom.Rectangle2D</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>Requires datanucleus-geospatial plugin. Persisted as (double, double, double, double) or (float, float, float, float) on RDBMS, or as String elsewhere.</td>
</tr>
<tr>
<td>java.awt.geom.Arc2D</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>Requires datanucleus-geospatial plugin. Persisted as (double, double, double, double, double, double, int) or (float, float, float, float, float, float, int) on RDBMS, or as String elsewhere.</td>
</tr>
<tr>
<td>java.awt.geom.CubicCurve2D</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>Requires datanucleus-geospatial plugin. Persisted as (double, double, double, double, double, double, double) or (float, float, float, float, float, float, float) on RDBMS, or as String elsewhere.</td>
</tr>
<tr>
<td>Java Type</td>
<td>DFG?</td>
<td>Proxy?</td>
<td>PK?</td>
<td>Comments</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------</td>
<td>--------</td>
<td>-----</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>java.awt.geom.Ellipse2D</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>Requires <code>datanucleus-geospatial</code> plugin. Persisted as (double, double, double, double) or (float, float, float, float) on RDBMS, or as String elsewhere.</td>
</tr>
<tr>
<td>java.awt.geom.QuadCurve2D</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>Requires <code>datanucleus-geospatial</code> plugin. Persisted as (double, double, double, double, double, double) or (float, float, float, float, float, float) on RDBMS, or as String elsewhere.</td>
</tr>
<tr>
<td>java.awt.geom.RoundRectangle2D</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>Requires <code>datanucleus-geospatial</code> plugin. Persisted as (double, double, double, double, double, double) or (float, float, float, float, float, float) on RDBMS, or as String elsewhere.</td>
</tr>
<tr>
<td>oracle.spatial.geometry.JGeometry</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>Requires <code>datanucleus-geospatial</code> plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on Oracle(SDO_GEOMETRY), MySQL(geometry).</td>
</tr>
<tr>
<td>com.vividsolutions.jts.geom.Geometry</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>Requires <code>datanucleus-geospatial</code> plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on Oracle(SDO_GEOMETRY), MySQL(geometry), PostGIS(geometry).</td>
</tr>
<tr>
<td>com.vividsolutions.jts.geom.GeometryCollection</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>Requires <code>datanucleus-geospatial</code> plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on Oracle(SDO_GEOMETRY), MySQL(geometry), PostGIS(geometry).</td>
</tr>
<tr>
<td>com.vividsolutions.jts.geom.LinearRing</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>Requires <code>datanucleus-geospatial</code> plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on Oracle(SDO_GEOMETRY), MySQL(geometry), PostGIS(geometry).</td>
</tr>
<tr>
<td>com.vividsolutions.jts.geom.LineString</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>Requires <code>datanucleus-geospatial</code> plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on Oracle(SDO_GEOMETRY), MySQL(geometry), PostGIS(geometry).</td>
</tr>
<tr>
<td>com.vividsolutions.jts.geom.MultiLineString</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>Requires <code>datanucleus-geospatial</code> plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on Oracle(SDO_GEOMETRY), MySQL(geometry), PostGIS(geometry).</td>
</tr>
<tr>
<td>Java Type</td>
<td>DFG?</td>
<td>Proxy?</td>
<td>PK?</td>
<td>Comments</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------</td>
<td>--------</td>
<td>-----</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>com.vividsolutions.jts.geom.MultiPoint</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>Requires datanucleus-geospatial plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on Oracle(SDO_GEOMETRY), MySQL(geometry), PostGIS(geometry).</td>
</tr>
<tr>
<td>com.vividsolutions.jts.geom.MultiPolygon</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>Requires datanucleus-geospatial plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on Oracle(SDO_GEOMETRY), MySQL(geometry), PostGIS(geometry).</td>
</tr>
<tr>
<td>com.vividsolutions.jts.geom.Point</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>Requires datanucleus-geospatial plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on Oracle(SDO_GEOMETRY), MySQL(geometry), PostGIS(geometry).</td>
</tr>
<tr>
<td>com.vividsolutions.jts.geom.Polygon</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>Requires datanucleus-geospatial plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on Oracle(SDO_GEOMETRY), MySQL(geometry), PostGIS(geometry).</td>
</tr>
<tr>
<td>org.postgis.Geometry</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>Requires datanucleus-geospatial plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on MySQL(geometry), PostGIS(geometry).</td>
</tr>
<tr>
<td>org.postgis.GeometryCollection</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>Requires datanucleus-geospatial plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on MySQL(geometry), PostGIS(geometry).</td>
</tr>
<tr>
<td>org.postgis.LinearRing</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>Requires datanucleus-geospatial plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on MySQL(geometry), PostGIS(geometry).</td>
</tr>
<tr>
<td>org.postgis.LineString</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>Requires datanucleus-geospatial plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on MySQL(geometry), PostGIS(geometry).</td>
</tr>
</tbody>
</table>

31
### Java Type

<table>
<thead>
<tr>
<th>Java Type</th>
<th>DFG?</th>
<th>Proxy?</th>
<th>PK?</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.postgis.MultiLineString</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Requires <strong>datanucleus-geospatial</strong> plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on MySQL(geography), PostGIS(geography).</td>
</tr>
<tr>
<td>org.postgis.MultiPoint</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Requires <strong>datanucleus-geospatial</strong> plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on MySQL(geography), PostGIS(geography).</td>
</tr>
<tr>
<td>org.postgis.MultiPolygon</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Requires <strong>datanucleus-geospatial</strong> plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on MySQL(geography), PostGIS(geography).</td>
</tr>
<tr>
<td>org.postgis.Point</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Requires <strong>datanucleus-geospatial</strong> plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on MySQL(geography), PostGIS(geography).</td>
</tr>
<tr>
<td>org.postgis.Polygon</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Requires <strong>datanucleus-geospatial</strong> plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on MySQL(geography), PostGIS(geography).</td>
</tr>
<tr>
<td>org.postgis.PGbox2d</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Requires <strong>datanucleus-geospatial</strong> plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on PostGIS(geography).</td>
</tr>
<tr>
<td>org.postgis.PGbox3d</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Requires <strong>datanucleus-geospatial</strong> plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on PostGIS(geography).</td>
</tr>
</tbody>
</table>

Some extra notes for implementation of JTS, JGeometry and PostGIS types support :-

- **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.

- **Oracle** 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 Oracle, will result in failure!

- **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
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-geospatial 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.

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

datanucleus-geospatial 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.
<package name="org.datanucleus.samples.jtsgeometry">
    <extension vendor-name="datanucleus" key="spatial-dimension" value="2"/>
    <extension vendor-name="datanucleus" key="spatial-srid" value="4326"/>
    <class name="SampleGeometry" detachable="true">
        <field name="id"/>
        <field name="name"/>
        <field name="geom" persistence-modifier="persistent">
            <extension vendor-name="datanucleus" key="mapping" value="no-userdata"/>
        </field>
    </class>
    <class name="SampleGeometryCollectionM" table="samplejtsgeometrycollectionm"
detachable="true">
        <extension vendor-name="datanucleus" key="postgis-hasMeasure" value="true"/>
        <field name="id"/>
        <field name="name"/>
        <field name="geom" persistence-modifier="persistent"/>
    </class>
    <class name="SampleGeometryCollection3D" table="samplejtsgeometrycollection3d"
detachable="true">
        <extension vendor-name="datanucleus" key="spatial-srid" value="-1"/>
        <extension vendor-name="datanucleus" key="spatial-dimension" value="3"/>
        <field name="id"/>
        <field name="name"/>
        <field name="geom" persistence-modifier="persistent"/>
    </class>
</package>

- [1] - The srid & dimension values are used in various places. One of them is schema creation, when using PostGIS, another is when you query the SpatialHelper.

- [2] - Every JTS geometry object can have a user data object attached to it. The default behaviour is to serialize that object and store it in a separate column in the database. If for some reason this isn’t desired, the mapping extension can be used with value “no-mapping” and DataNucleus-Geospatial will ignore the user data objects.

- [3] - If you want to use measure types in PostGIS you have to define that using the postgis-hasMeasure extension.

Other Types

Many other types are supported.
<table>
<thead>
<tr>
<th>Java Type</th>
<th>DFG?</th>
<th>Proxy?</th>
<th>PK?</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Object</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>Either persisted serialized, or represents multiple possible types</td>
</tr>
<tr>
<td>java.util.Currency</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>Persisted as String.</td>
</tr>
<tr>
<td>java.util.Locale</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>Persisted as String.</td>
</tr>
<tr>
<td>java.util.UUID</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>Persisted as String, or alternatively as native uuid on PostgreSQL when specifying sql-type=&quot;uuid&quot;.</td>
</tr>
<tr>
<td>java.util.Optional&amp;type&gt;</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Persisted as the type of the generic type that optional represents.</td>
</tr>
<tr>
<td>java.awt.Color</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Persisted as String or as (Integer,Integer,Integer,Integer) storing red,green,blue,alpha respectively.</td>
</tr>
<tr>
<td>java.awt.image.BufferedImage</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>Persisted as serialized.</td>
</tr>
<tr>
<td>java.net.URI</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>Persisted as String.</td>
</tr>
<tr>
<td>java.net.URL</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>Persisted as String.</td>
</tr>
<tr>
<td>java.io.Serializable</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>Persisted as serialized.</td>
</tr>
<tr>
<td>java.io.File</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>Only for RDBMS, persisted to LONGVARBINARY, and retrieved as streamable so as not to adversely affect memory utilisation, hence suitable for large files.</td>
</tr>
</tbody>
</table>

**Arrays**

The vast majority of the SCO types can also be persisted as arrays of that type as well. Here we list a few of the combinations definitely supported as arrays, but others likely will work fine.

<table>
<thead>
<tr>
<th>Java Type</th>
<th>DFG?</th>
<th>Proxy?</th>
<th>PK?</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean[]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>See the Arrays Guide</td>
</tr>
<tr>
<td>byte[]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>See the Arrays Guide</td>
</tr>
<tr>
<td>char[]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>See the Arrays Guide</td>
</tr>
<tr>
<td>double[]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>See the Arrays Guide</td>
</tr>
<tr>
<td>float[]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>See the Arrays Guide</td>
</tr>
<tr>
<td>int[]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>See the Arrays Guide</td>
</tr>
<tr>
<td>long[]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>See the Arrays Guide</td>
</tr>
</tbody>
</table>
### Generic Type Variables

JDO does not explicitly require support for generic type variables. DataNucleus does support some situations with generic type variables.

The first example that is largely supported is where you have an abstract base class with a generic Type Variable and then you specify the type in the (concrete) subclass(es).
public abstract class Base<T> {
    private T id;
}

public class Sub1 extends Base<Long> {

}

public class Sub2 extends Base<Integer> {

}

Similarly you use TypeVariables to form relations, like this

public abstract class Ownable<T extends Serializable> implements Serializable {
    @ManyToOne(optional = false)
    private T owner;
}

public class Document extends Ownable<Person> {

}

Clearly there are many combinations of where TypeVariables can be used and DataNucleus supports a subset of these currently. Let us know if your generics usage isn’t supported.

**JDO Attribute Converters**

JDO3.2 introduces an API for conversion of an attribute of a PersistenceCapable object to its datastore value. You can define a "converter" that will convert to the datastore value and back from it, implementing this interface.

public interface AttributeConverter<X,Y> {
    public Y convertToDatastore(X attributeValue);
    public X convertToAttribute(Y datastoreValue);
}

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 convertToAttribute(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 convertToDatastore(URL url) {
        return url != null ? url.toString() : null;
    }
}

and now in our PersistenceCapable class we mark any URL field as being converted using this converter

@PersistenceCapable
public class MyClass {
    @PrimaryKey
    long id;

    @Convert(URLStringConverter.class)
    URL url;

    ...
}

Note that you can register converters as the default handler for a java type when constructing the PMF via persistence properties. These properties should be of the form javax.jdo.option.typeconverter.{javatype} and the value is the class name of the AttributeConverter.

A further use of AttributeConverter is where you want to apply type conversion to the key/value of
a Map field, or to the element of a Collection field. The Collection element case is simple, you just specify the @Convert against the field and it will be applied to the element. If you want to apply type conversion to a key/value of a map do this.

```java
@Key(converter=URLStringConverter.class)
Map<URL, OtherEntity> myMap;
```

You CANNOT use an AttributeConverter for a PersistenceCapable type. This is because a PersistenceCapable type requires special treatment, such as attaching aStateManager etc.

**Types extending Collection/Map**

Say you have your own type that extends Collection/Map. By default DataNucleus will not know how to persist this. You could declare the type in your class as Collection/Map, but often you want to refer to your own type. By default if a type extends Collection/Map then it will be marked as second class, and that DataNucleus will treat it as Collection that should be wrapped (see above for Collection/Map). If you have your type and want to just persist it into a single column then you should do as follows

```java
public class MyCollectionType extends Collection
{
    ...
}

@PersistenceCapable
public class MyClass
{
    MyCollectionType myField;
    ...
}
```

We now define a simple AttributeConverter to allow us to persist fields of this type in String form in the datastore, like this
public class MyCollectionTypeStringConverter implements AttributeConverter<MyCollectionType, String>
{
    public MyCollectionType convertToAttribute(String str)
    {
        if (str == null)
        {
            return null;
        }

        ...

        return myType;
    }

    public String convertToDatastore(MyCollectionType myType)
    {
        return myType != null ? myType.toString() : null;
    }
}

and now in our PersistenceCapable class we mark the myField as being converted using this converter

@PersistenceCapable
public class MyClass
{
    @Convert(MyCollectionTypeStringConverter.class)
    MyCollectionType myField;

    ...
}

This is all well and good, it will map this type on to a single String-based column now. There remains one problem, it is still thinking that this type should be wrapped (to intercept calls to the Collection methods). Since our type hides all of that, we want to disable this. So we update the mapping to be

@PersistenceCapable
public class MyClass
{
    @Convert(MyCollectionTypeStringConverter.class)
    @Extension(vendorName="datanucleus", key="is-second-class", value="false")
    MyCollectionType myField;

    ...
}
so now it is not attempted to be wrapped and will persist and retrieve fine.

If you want your extension of Collection/Map to be managed as a *mutable_second
class type then you will need to provide a _wrapper class for it. Please refer to the
java_type extension for how to support this. In which case you could omit this is-
second-class metadata.

## 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

```
@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.

You CANNOT use a TypeConverter for a PersistenceCapable type. This is because a
PersistenceCapable type requires special treatment, such as attaching a
StateManager etc.
Identity

All JDO-enabled persistable classes need to have an "identity" to be able to identify an object for retrieval and relationships. There are three types of identity defineable using JDO. These are

- **Datastore Identity**: a surrogate column is added to the persistence of the persistable type, and objects of this type are identified by the class plus the value in this surrogate column.

- **Application Identity**: a field, or several fields of the persistable type are assigned as being (part of) the primary key.

- **Nondurable Identity**: the persistable type has no identity as such, so the only way to lookup objects of this type would be via query for values of specific fields. This is useful for storing things like log messages etc.

A further complication is where you use application identity but one of the fields forming the primary key is a relation field. This is known as **Compound Identity**.

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) 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 defined.

**Datastore Identity**

- Applicable to RDBMS, ODF, Excel, OOXML, HBase, Neo4j, MongoDB, XML, Cassandra, JSON

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.

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

or using JDO annotations

```
@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'.

**Datastore Identity : Generating identities**

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 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
@PersistenceCapable
@DatastoreIdentity(strategy="sequence", sequence="MY_SEQUENCE")
public class MyClass
{
    ... 
}
```

or using XML metadata

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

Some of the datastore identity strategies require additional attributes, but the specification is straightforward.

See also :-

- **Value Generation** - strategies for generating ids
- **MetaData reference** for `<datastore-identity>` element
- **Annotations reference** for `@DatastoreIdentity`

**Datastore Identity : 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.
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**

### Datastore Identity : 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.

**Plugin**

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.

### Datastore Identity : 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);
```

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

```
Object obj = pm.getObjectById(MyClass.class, mykey);
```
Application Identity

Applicable to all datastores.

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="myPrimaryFieldName" 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 myPrimaryFieldName;
}
```

See also :-

- MetaData reference for `<field>` element
- Annotations reference for `@Persistent`

**Application Identity : 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
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

**persistable**

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.

**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;
    ...
}
```
<class name="MyClass" identity-type="application">
  <field name="id" primary-key="true"/>
...
</class>

or using annotations

```java
@PersistenceCapable
public class MyClass
{
  @PrimaryKey
  long id;
  ...
}
```

Note that 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

```java
// For a "long" type :
javax.jdo.identity.LongIdentity id = new javax.jdo.identity.LongIdentity(myClass, 101);

// For a "String" type :
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

**PrimaryKey : Rules for User-Defined 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.
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.

Since there are many possible combinations of primary-key fields it is impossible for JDO 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.

**Primary Key Example - Multiple Field**

Here's an example of a composite (multiple field) primary key class

```java
@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;
    }
}
Application Identity : 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 :-

- **Value Generation** - strategies for generating ids

Application Identity : 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

```
// Using the PersistenceManager
Object id = pm.getObjectId(obj);

// Using JDOHelper
Object id = JDOHelper.getObjectId(obj);
```

Application Identity : 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.**

Application Identity : 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.getInstanceById(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.

**Nondurable Identity**

Applicable to RDBMS, ODF, Excel, OXXML, HBase, Neo4j, MongoDB.
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
{
  ...
}
```

What this means for something like RDBMS is that the table (or view) of the class will not have a primary-key.

### Compound Identity Relationships

A JDO "compound identity relationship" is a relationship between two classes in which the child object must coexist with the parent object and where the primary key of the child includes the persistable object of the parent. The key aspect of this type of relationship is that the primary key of one of the classes includes a persistable 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)**

In the identity class of the compound persistable class you should define the **object-idclass** of the persistable type being contained and use that type in the identity class of the compound persistable type.

The persistable class that is contained cannot be using **datastore identity**, and must be using **application identity** with an objectid-class.

When using compound identity, it is best practice to define an **object-idclass** for any persistable classes that are part of the primary key, and **not** rely on the built-in identity types.
1-1 Relationship

Lets 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".

```java
public class User {
  long id;

  ...
}

public class Account {
  User user;

  ...
}
```

In addition we need to define primary key classes for our User and Account classes

```java
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

```
USER
+USER_ID
LOGIN

ACCOUNT
+USER_ID
FIRSTNAME
LASTNAME
```

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

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.

```java
public class Account
{
    long id;

    Set<Address> addresses;

    ...
}
public class Address
{
    long id;

    Account account;

    ...
}
```

In addition we need to define primary key classes for our Account and Address classes

```java
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

        public PK()
        {
        }
    }
public PK(String s)
{
    StringTokenizer token = new StringTokenizer(s, "::");
    this.id = Long.valueOf(token.nextToken()).longValue();
    this.account = new Account.PK(token.nextToken());
}

public String toString()
{
    return "" + id + "::" + this.account.toString();
}

public int hashCode()
{
    return (int)id ^ account.hashCode();
}

public boolean equals(Object other)
{
    if (other != null && (other instanceof PK))
    {
        PK otherPK = (PK)other;
        return otherPK.id == this.id && this.account.equals(otherPK.account);
    }
    return false;
}
So now we have the following datastore schema

```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>
```

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.
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.

```java
public class Account
{
    long id;

    Map<String, Address> addresses;

    ...
}

public class Address
{
    long id;

    String alias;

    Account account;

    ...
}
```

In addition we need to define primary key classes for our Account and Address classes

```java
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;
}

/**
 * Inner class representing Primary Key
 */
public static class PK implements Serializable
{
    public String alias; // Same name as real field above
    public Account.PK account; // Same name as the real field above

    public PK()
    {
    }

    public PK(String s)
    {
        StringTokenizer token = new StringTokenizer(s, "::");
        this.alias = Long.valueOf(token.nextToken()).longValue();
        this.account = new Account.PK(token.nextToken());
    }
}
public String toString()
{
    return alias + "::" + this.account.toString();
}

public int hashCode()
{
    return alias.hashCode() ^ account.hashCode();
}

public boolean equals(Object other)
{
    if (other != null && (other instanceof PK))
    {
        PK otherPK = (PK)other;
        return otherPK.alias.equals(this.alias) && this.account.equals(otherPK.account);
    }
    return false;
}

To achieve what we want with the datastore schema we define the MetaData like this
<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

![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
- 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.
Versioning

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 doesn't currently support this option**

Versioning using a surrogate column

The default JDO 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.

Versioning using a field/property of the class

**Extension**

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 default versioning process whereby you have to have a field present. To do this lets take a class

```java
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.
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 String utilising the JRE UUID class (DataNucleus extension)
- **uuid-object** - provides a pure UUID object utilising the JRE UUID class (DataNucleus extension)
- **auid** - provides a pure UUID following the OpenGroup standard (DataNucleus extension)
- **timestamp** - creates a java.sql.Timestamp of the current time (DataNucleus extension)
- **timestamp-value** - creates a long (millisecs) of the current time (DataNucleus extension)
- **max** - uses a max(column)+1 method (only in RDBMS) (DataNucleus extension)
- **datastore-uuid-hex** - UUID in hexadecimal form using datastore capabilities (only in RDBMS) (DataNucleus extension)
- **user-supplied value generators** - allows you to hook in your own identity generator (DataNucleus extension)

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

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
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.

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:

- Applicable to Oracle, PostgreSQL, SAPDB, DB2, Firebird, HSQLDB, H2, Derby, SQLServer, NuoDB

To configure a class to use either of these generation methods with datastore identity you simply add this to the class' Meta-Data

```
<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=IdGeneratorStrategy.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
<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

@PersistenceCapable
@Sequence(name="yourseq", datastore-sequence="YOUR_SEQUENCE_NAME" strategy =NONCONTIGUOUS/>
public class MyClass
{
  @Persistent(valueStrategy=IdGeneratorStrategy.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-min-value</td>
<td>determines the minimum value a sequence can generate</td>
<td>No</td>
</tr>
<tr>
<td>key-max-value</td>
<td>determines the maximum value a sequence can generate</td>
<td>No</td>
</tr>
<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

**identity**

Applicable to DB2 (IDENTITY), MySQL (AUTOINCREMENT), SQLServer (IDENTITY), Sybase (IDENTITY), HSQLDB (IDENTITY), H2(IDENTITY), PostgreSQL (SERIAL), Derby (IDENTITY), MongoDB (String), Neo4j (long), NuoDB (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.

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 **datstore 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>
```

or using annotations

```java
@PersistenceCapable
@DatastoreIdentity(strategy=IdGeneratorStrategy.IDENTITY)
public class MyClass {
  ...
}
```

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
<class name="myclass" identity-type="application">
  <field name="myfield" primary-key="true" value-strategy="identity"/>
  ...
</class>
```

or using annotations

```java
@PersistenceCapable
public class MyClass {
  @Persistent(valueStrategy=IdGeneratorStrategy.IDENTITY, primaryKey="true")
  long myfield;
}
```

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).
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

### 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.

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

or using annotations

```java
@PersistenceCapable
@DatastoreIdentity(strategy=IdGeneratorStrategyINCREMENT)
public class MyClass {...}
```

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
<class name="MyClass" ... >
   <field name="myfield" primary-key="true" value-strategy="increment"/>
   ...
</class>
```

or using annotations

```java
@PersistenceCapable
public class MyClass
{
   @Persistent(valueStrategy=IdGeneratorStrategyINCREMENT, primaryKey="true")
   long myfield;
   ...
}
```

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>
<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. Refer to persistence property datanucleus.valuegeneration.increment.allocationSize</td>
<td>No. Defaults to 10</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 &quot;base table name&quot; when the root class has &quot;subclass-table&quot; this should be set to &quot;class&quot; when the root class has &quot;subclass-table&quot; 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>
</tbody>
</table>

This value generator will generate values unique across different JVMs

**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:

```xml
<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:

```xml
<class name="myclass">
  <field name="myfield" primary-key="true" value-strategy="uuid-string"/>
  ...
</class>
```

### 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:

```xml
<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.
**datastore-uuid-hex**

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>
```

**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
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="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.

**uuid**

This generator uses the JRE UUID class to generate String 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>
```

or using annotations

```java
@PersistenceCapable
@DatastoreIdentity(customStrategy="uuid")
public class MyClass {...}
```

To configure an application identity class to use this generation method you simply add this to the class' JDO Meta-Data.
This value generator will generate values unique across different JVMs

**uuid-object**

This generator uses the JRE UUID class to generate UUID 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.

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-object"/>
  </class>
```

Or using annotations

```java
public class MyClass {
    @Persistent(customValueStrategy="uuid-object")
    UUID myField;
}
```

This value generator will generate values unique across different JVMs
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-MMHHH-CCCG-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

**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.
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>
```

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-value"/>
    ...
</class>
```

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
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);
            }
            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;
            }
        }
    }
Some ValueGenerators are specific to RDBMS datastores, and some are generic, so bear this in mind when selecting and adding your own.
1-1 Relations

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.

For RDBMS a 1-1 relation is stored as a foreign-key column(s). For non-RDBMS it is stored as a String "column" storing the 'id' (possibly with the class-name included in the string) of the related object.

Unidirectional

For this case you could have 2 classes, User and Account, as below.

```java
public class Account
{
    User user;
}

public class User
{
    ...
}
```

so the Account class knows about the User class, but not vice-versa. If you define the annotations for these classes as follows

```java
public class Account
{
    ...

    @Column(name="USER_ID")
    User user;
}

public class User
{
    ...
}
```

or using XML metadata
<package name="mydomain">
   <class name="User" table="USER">
      ...
   </class>

   <class name="Account" table="ACCOUNT">
      ...
      <field name="user">
         <column name="USER_ID"/>
      </field>
   </class>
</package>

This contrasts with JPA mapping where you have to explicitly specify that it is a one-to-one relation!

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 tables USER and ACCOUNT with relationships]

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 datastore 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.

**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.
public class Account
{
    User user;
    ...
}

public class User
{
    Account account;
    ...
}

We create the 1-1 relationship with a single foreign-key. To do this you define the annotations as

```java
public class Account
{
    @Column(name="USER_ID")
    User user;
}

public class User
{
    @Persistent(mappedBy="user")
    Account account;
}
```

or using XML metadata

```xml
<package name="mydomain">
    <class name="User" table="USER">
        ...
        <field name="account" mapped-by="user"/>
    </class>

    <class name="Account" table="ACCOUNT">
        ...
        <field name="user">
            <column name="USER_ID"/>
        </field>
    </class>
</package>
```
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

```
USER
+USER_ID
+LOGIN

ACCOUNT
+ACCOUNT_ID
+FIRSTNAME
+LASTNAME
#USER_ID
```

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.

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.
1-N Relations

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 whilst Sets don't allow duplicates, and so the persistence process reflects this with the choice of primary keys. There are two principal 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.

- Collection<PC> Unidirectional using Join Table
- Collection<PC> Unidirectional using Foreign-Key
- Collection<PC> Bidirectional using Join Table
- Collection<PC> Bidirectional using Foreign-Key
- List<PC>
- Collection<Simple> using Join Table
- Collection<Simple> using AttributeConverter into single column
- Collection<PC> using shared join table (DataNucleus Extension)
- Collection<PC> using shared foreign key (DataNucleus Extension)
- Map<PC, PC> using join table
- Map<Simple, PC> using join table
- Map<Simple, Simple> using join table
- Map<Simple, Simple> using AttributeConverter into single column
- Map<PC, Simple> using join table
- Map<Simple, PC> - Unidirectional using foreign-key (key stored in the value class)
- Map<Simple, PC> - Bidirectional using foreign-key (key stored in the value class)
- Map<PC,Simple> - Unidirectional using foreign-key (value stored in the key class)

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.
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.

Collection<PC> Unidirectional JoinTable

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

```java
public class Account {
    Collection<Address> addresses
    ...
}
public class Address {
    ...
}
```

If you define the annotations of the classes like this

```java
public class Account {
    ...

    @Persistent(table="ACCOUNT_ADDRESSES")
    @Join(column="ACCOUNT_ID_OID")
    @Element(column="ADDRESS_ID_EID")
    Collection<Address> addresses;
}

public class Address {
    ...
}
```

or using XML metadata
<package name="com.mydomain">
  <class name="Account">
    ...
    <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">
    ...
  </class>
</package>

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.

![Diagram of database tables]

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_PKIDX" 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 (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

Collection<PC> Unidirectional FK

We have the same classes Account and Address as above for the join table case, but this time we will store the "relation" as a foreign key in the Address class. So we define the annotations like this

```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.

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.
- 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.

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.

**Collection<PC> Bidirectional JoinTable**

We have our 2 sample classes Account and Address. These are related in such a way as Account contains a Collection of objects of type Address, and now each Address has a reference to the Account object that it relates to. Like this:

```java
public class Account {
    Collection<Address> addresses;

    ...
}

public class Address {
    Account account;

    ...
}
```
If you define the annotations for these classes as follows

```java
public class Account {
    ...

    @Persistent(mappedBy="account")
    @Join
    Collection<Address> addresses;
}

public class Address {
    ...
}
```

or using XML metadata

```xml
<package name="com.mydomain">
    <class name="Account">
        ...
        <field name="addresses" mapped-by="account">
            <collection element-type="com.mydomain.Address"/>
            <join/>
        </field>
    </class>
    <class name="Address">
        ...
        <field name="account"/>
    </class>
</package>
```

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 (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

Collection<PC> Bidirectional FK

We have the same classes Account and Address as above for the join table case, but this time we will store the "relation" as a foreign key in the Address class. If you define the annotations for these classes as follows

```java
public class Account
{
    ...

    @Persistent(mappedBy="account")
    Collection<Address> addresses;
}

class Address
{
    @Column(name="ACCOUNT_ID")
    Account account;
}
```
or using XML metadata

```xml
<package name="com.mydomain">
  <class name="Account">
    ...  
    <field name="addresses" mapped-by="account">
      <collection element-type="com.mydomain.Address"/>
    </field>
  </class>
  ...
</package>

<class name="Address">
  ...
  <field name="account">
    <column name="ACCOUNT_ID"/>
  </field>
</class>
```

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.

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.
Using a List

In the case of the relation field being a List (i.e ordered), you define the relation just like you would for a Collection (above) but then define whether you want the relation to be either ordered or indexed.

By default JDO operates with indexed lists (i.e adds a surrogate column in the element or in the join table), and you simply add the following where required

```java
@Order
```

or using XML

```xml
<order/>
```

If you have defined the field type as a List then this is not required to be added unless you want to configure details of the order column.

DataNucleus also supports ordered lists whereby the elements of the List are ordered according to some field (or fields) of the element.

If you have an element with a field called "city" then this specification will use that field for ordering (and not add a surrogate ordering column).

```java
@Order(extensions=@Extension(vendorName="datanucleus", key="list-ordering", value ="city ASC"))
```

```xml
<order>
    <extension vendor-name="datanucleus" key="list-ordering" value="city ASC"/>
</order>
```

Collection<Simple> via JoinTable

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
public class Account
{
    ...

    @Persistent
    @Join
    @Element(column="ADDRESS")
    Collection<String> addresses;
}

or using XML metadata

```xml
<package name="com.mydomain">
    <class name="Account">
        ...
        <field name="addresses" persistence-modifier="persistent">
            <collection element-type="java.lang.String"/>
            <join/>
            <element column="ADDRESS"/>
        </field>
    </class>
</package>
```

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.

```
ACCOUNT
+ACCOUNT_ID
FIRSTNAME
LASTNAME
```

```
ACCOUNT_ADDRESSES
+ACCOUNT_ID_OID
ADDRESS
+ADPT_PK_IDX
```

The column ADPT_PK_IDX is added by DataNucleus so that duplicates can be stored. You can control the name of this column by adding an <order> element and specifying the column name for the order column (within <field>).

**Collection<Simple> using AttributeConverter via column**

Just like in the above example, here we have a Collection of simple types. In this case we are wanting to store this Collection into a single column in the owning table. We do this by using a JDO AttributeConverter.
public class Account
{
    ...
    @Persistent
    @Convert(CollectionStringToStringConverter.class)
    @Column(name = "ADDRESSES")
    Collection<String> addresses;
}

and then define our converter. You can clearly define your conversion process how you want it. You could, for example, convert the Collection into comma-separated strings, or could use JSON, or XML, or some other format.

public class CollectionStringToStringConverter implements AttributeConverter<Collection<String>, String>
{
    public String convertToDatastore(Collection<String> attribute)
    {
        if (attribute == null)
        {
            return null;
        }

        StringBuilder str = new StringBuilder();
        ... convert Collection to String
        return str.toString();
    }

    public Collection<String> convertToAttribute(String columnValue)
    {
        if (columnValue == null)
        {
            return null;
        }

        Collection<String> coll = new HashSet<String>();
        ... convert String to Collection
        return coll;
    }
}

Collection<PC> via Shared JoinTable

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

```java
public class Account
{
    Collection<Address> workAddresses;
    Collection<Address> homeAddresses;
    ...
}
```

We now change the metadata we had earlier to allow for 2 collections, but sharing the join table

```java
import org.datanucleus.api.jdo.annotations.SharedRelation;

public class Account
{
    ...
    @Persistent
    @Join(table="ACCOUNT_ADDRESSES", columns={@Column(name="ACCOUNT_ID_OID"))
    @Element(columns={@Column(name="ADDRESS_ID_EID"))
    @SharedRelation(column="ADDRESS_TYPE", value="work")
    Collection<Address> workAddresses;

    @Persistent
    @Join(table="ACCOUNT_ADDRESSES", columns={@Column(name="ACCOUNT_ID_OID"))
    @Element(columns={@Column(name="ADDRESS_ID_EID"))
    @SharedRelation(column="ADDRESS_TYPE", value="home")
    Collection<Address> homeAddresses;

    ...
}
```

or using XML metadata
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:
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

```java
public class Account {
    Collection<Address> workAddresses;
    Collection<Address> homeAddresses;
    ...
}
```

We now change the metadata we had earlier to allow for 2 collections, but sharing the join table

```java
import org.datanucleus.api.jdo.annotations.SharedRelation;

public class Account {
    ...
    @Persistent
    @SharedRelation(column="ADDRESS_TYPE", value="work")
    Collection<Address> workAddresses;
    ...
    @Persistent
    @SharedRelation(column="ADDRESS_TYPE", value="home")
    Collection<Address> homeAddresses;
    ...
}
```

or using XML metadata
<package name="com.mydomain">
  <class name="Account">
    ...
    <field name="workAddresses" persistence-modifier="persistent">
      <collection element-type="com.mydomain.Address"/>
      <element column="ACCOUNT_ID_OID"/>
      <extension vendor-name="datanucleus" key="relation-discriminator-column" value="ADDRESS_TYPE"/>
      <extension vendor-name="datanucleus" key="relation-discriminator-value" value="work"/>
    </field>
    <field name="homeAddresses" persistence-modifier="persistent">
      <collection element-type="com.mydomain.Address"/>
      <element column="ACCOUNT_ID_OID"/>
      <extension vendor-name="datanucleus" key="relation-discriminator-column" value="ADDRESS_TYPE"/>
      <extension vendor-name="datanucleus" key="relation-discriminator-value" value="home"/>
    </field>
  </class>
  <class name="Address">
    ...
  </class>
</package>

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 (@SharedRelation annotation). 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

**Account**
+ACCOUNT_ID
FIRSTNAME
LASTNAME

**Address**
+ADDRESS_ID
STREET
CITY
#ACCOUNT_ID_OID
-ADDRESS_TYPE

Map<PC,PC> using Join Table

Here we have a Map field, with key and value as persistable classes.
If we define the annotations like this

```java
@PersistenceCapable
public class Account {
    ...
    Map<Name, Address> addresses;
}
```

or using XML metadata

```xml
<package name="com.mydomain">
    <class name="Account" identity-type="datastore">
        ...
        <field name="addresses" persistence-modifier="persistent">
            <map/>
            <join/>
        </field>
    </class>
    <class name="Address" identity-type="datastore">
        ...
    </class>
    <class name="Name" identity-type="datastore">
        ...
    </class>
</package>
```

we don't need to set the keyType or valueType since we are using generics.

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
<field name="addresses" persistence-modifier="persistent" table="ACCOUNT_ADDRESS">
  <map/>
  <join>
    <column name="ACCOUNT_ID"/>
  </join>
  <key>
    <column name="NAME_ID"/>
  </key>
  <value>
    <column name="ADDRESS_ID"/>
  </value>
</field>
```

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.
Map<Simple,PC> using Join Table

Here our key is a simple type (in this case a String) and the values are persistable. Like this

```java
public class Account {
   Map<String, Address> addresses;

   ...
}

public class Address {...}
```

or using XML metadata

```xml
<package name="com.mydomain">
   <class name="Account" identity-type="datastore">
      ...
      <field name="addresses" persistence-modifier="persistent">
         <map/>
         <join/>
      </field>
   </class>

   <class name="Address" identity-type="datastore">
      ...
   </class>
</package>
```

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.

**Map<PC,Simple> using Join Table**

Here our value is a simple type (in this case a String) and the keys are *persistable*. Like this

```java
public class Account {
    Map<Address, String> addresses;
    ...
}

public class Address {...}
```

or using XML metadata

```xml
<package name="com.mydomain">
    <class name="Account" identity-type="datastore">
        ...
        <field name="addresses" persistence-modifier="persistent">
            <map/>
            <join/>
        </field>
    </class>

    <class name="Address" identity-type="datastore">
        ...
    </class>
</package>
```

This operates exactly the same as "Map<Simple, PC>" except that the additional table is for the key instead of the value.

**Map<Simple, Simple> using Join Table**

Here our keys and values are of simple types (in this case a String). Like this
public class Account
{
    Map<String, String> addresses;
    ...
}

If you define the annotations for these classes as follows

@PersistenceCapable
public class Account
{
    @Join
    Map<String, String> addresses;
    ...
}

or using XML metadata

<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.
**Map<Simple, Simple> using AttributeConverter via column**

Just like in the above example, here we have a Map of simple keys/values. In this case we are wanting to store this Map into a single column in the owning table. We do this by using a JDO AttributeConverter.

```java
public class Account
{
    ...
    @Persistent
    @Convert(MapStringToStringConverter.class)
    @Column(name="ADDRESSES")
    Map<String, String> addresses;
}
```

and then define our converter. You can clearly define your conversion process how you want it. You could, for example, convert the Map into comma-separated strings, or could use JSON, or XML, or some other format.
public class MapStringStringToStringConverter implements AttributeConverter<Map<String, String>, String> {
    
    public String convertToDatastore(Map<String, String> attribute) {
        if (attribute == null) {
            return null;
        }

        StringBuilder str = new StringBuilder();
        ... convert Map to String
        return str.toString();
    }

    public Map<String, String> convertToAttribute(String columnValue) {
        if (columnValue == null) {
            return null;
        }

        Map<String, String> map = new HashMap<String, String>();
        ... convert String to Map
        return map;
    }
}

Map<Simple, PC> Unidirectional FK (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. Here we use a field of the value as the key. The classes are like this

```java
public class Account {
    Map<String, Address> addresses;
}

public class Address {
    String alias; // Use as key
}
```

In this relationship, the Account class has a Map of Address objects, yet the Address knows nothing about the Account. We define the annotations like this
public class Account
{
    ...
    @Key(mappedBy="alias")
    Map<String, Address> addresses;
}

<package name="com.mydomain">
    <class name="Account" identity-type="datastore">
        ...
        <field name="addresses" persistence-modifier="persistent">
            <map/>
            <key mapped-by="alias"/>
            <value column="ACCOUNT_ID_OID"/>
        </field>
    </class>

    <class name="Address" identity-type="datastore">
        ...
        <field name="alias" null-value="exception">
            <column name="KEY" length="20" jdbc-type="VARCHAR"/>
        </field>
    </class>
</package>

There will be 2 tables, one for Address, and one for Account. Note that we now have no "join" annotation/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.
Map<Simple,PC> Unidirectional FK (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.

```java
public class Account {
    Map<String, Address> addresses;
}

public class Address {
    String alias; // Use as key
    Account account;
}
```

The only difference to the variant above is the bidirectional link back to the Account from Address.

So we define our metadata in a similar way

```xml
<package name="com.mydomain">
    <class name="Account" identity-type="datastore">
        ...
        <field name="addresses" persistence-modifier="persistent" mapped-by="account">
            <map/>
            <key mapped-by="alias"/>
        </field>
    </class>

    <class name="Address" identity-type="datastore">
        ...
        <field name="account"/>
        <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).
Map<PC,Simple> Unidirectional FK (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.

```java
public class Account {
    Map<Address, String> phoneNumbers;
}

public class Address {
    String businessPhoneNumber; // Use as value
}
```

We define the MetaData like this

```java
public class Account {
    @Value(mappedBy="businessPhoneNumber")
    Map<Address, String> phoneNumbers;
}
```
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.

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.
N-1 Relations

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.

For RDBMS a N-1 relation is stored as a foreign-key column(s). For non-RDBMS it is stored as a String "column" storing the 'id' (possibly with the class-name included in the string) of the related object.

Unidirectional with ForeignKey

For this case you could have 2 classes, User and Account, as below.

```java
public class Account {
    User user;
    ...
}
public class User {
    ...
}
```

so the Account class ("N" side) knows about the User class ("1" side), but not vice-versa. A particular user could be related to several accounts. If you define the annotations for these classes as follows

```java
public class Account {
    ...
    @Column(name="USER_ID")
    User user;
}
public class User {
    ...
}
```

or using XML metadata
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.

This is exactly the same as a 1-1 Unidirectional relation.

**Unidirectional with JoinTable**

For this case we have the same 2 classes, User and Account, as before.

```
public class Account
{
    User user;

    ...
}

public class User
{
    ...
}
```

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 annotations as follows

```java
public class Account
{
    ...

    @Persistent(table="ACCOUNT_USER")
    @Join
    User user;
}
```
or with XML metadata

```xml
<package name="mydomain">
    <class name="User" identity-type="datastore">
        ...
    </class>

    <class name="Account" identity-type="datastore">
        ...
        <field name="user" table="ACCOUNT_USER">
            <join/>
        </field>
    </class>
</package>
```

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.

![Database Table Diagram]

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

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`)

**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.
M-N Relations

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.

```java
public class Product {
    Set<Supplier> suppliers;
    ...
}

public class Supplier {
    Set<Product> products;
    ...
}
```

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.

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.

When adding objects to an M-N relation, you MUST add to the owner side as a minimum, and optionally also add to the non-owner side. Just adding to the non-owner side will not add the relation.

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.

The various possible relationships are described below.

- M-N Set relation
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.

Using Set

If you define the Meta-Data for these classes as follows:

```java
public class Product {
    ... 
    @Persistent(table="PRODUCTS_SUPPLIERS")
    @Join(column="PRODUCT_ID")
    @Element(column="SUPPLIER_ID")
    SetSupplier suppliers;
}

public class Supplier {
    ... 
    @Persistent(mappedBy="suppliers")
    SetProducts products;
}
```

or using XML metadata
<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>

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.

Using Ordered Lists

In this case our fields are of type List instead of Set used above. If you define the annotations for these classes as follows
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
}

or using XML metadata
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 ordered list specification 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 ordering.

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
public class Product {
    ...

    @Join
    List<Supplier> suppliers;
}

public class Supplier {
    ...

    @Join
    List<Products> products;
}

or using XML metadata

<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>

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 (indexed) 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.

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).
Arrays

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 *persistable*
- **Using a Foreign-Key in the element** - only available where the array is of a *persistable* type
- **Simple array stored in JoinTable** - the array is stored in a "join" table, with a column in that table storing each element of the array

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 OR update the array element and then call *JDOHelper.makeDirty(obj, "fieldName");*

**Single Column Arrays**

Let's suppose you have a class something like this

```java
public class Account {
    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 in a single-column into the table of the account (but we don't want them serialised).

In terms of metadata required we simply define the field as "persistent". You could add `<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:

```
ACCOUNT
+ACCOUNT_ID
FIRST_NAME
LAST_NAME
PERMISSIONS
```

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

### Serialised Arrays

Let's suppose you have a class something like the previous example

```java
public class Account {
    byte[] permissions;
    ...
}
```

and now we want to persist the permissions as **serialised** into the table of the account. We define metadata like this

```java
public class Account {
    @Serialized
    byte[] permissions;
    ...
}
```

or using XML metadata

```
<class name="Account" identity-type="datastore">
...
    <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

```
ACCOUNT
+ACCOUNT_ID
FIRST_NAME
LAST_NAME
PERMISSIONS
```

DataNucleus supports persistence of many array types in this way, including :_ boolean[], byte[],
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

```java
public class Account
{
    ...
    Permission[] permissions;
}

public class Permission
{
    ...
}
```

So an Account has an array of Permission*s, and both of these objects are entities. We want to persist the relationship using a join table. We define the MetaData as follows

```java
public class Account
{
    ...
    @Join(column="ACCOUNT_ID")
    @Element(column="PERMISSION_ID")
    @Order(column="PERMISSION_ORDER_IDX")
    Permission[] permissions;
}
```

or using XML metadata
This results in a datastore schema as follows

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`

### 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 `persistable` type. Let's take the same example above. So we have

```java
public class Account
{
    ... 

    Permission[] permissions;
}

public class Permission
{
    ... 
}
and the metadata is

```java
public class Account {
    ...
    @Element(column="ACCOUNT_ID")
    @Order(column="ACCOUNT_PERMISSION_ORDER_IDX")
    Permission[] permissions;
}
```

or using XML metadata

```xml
<class name="Account" table="ACCOUNT">...
    <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

```
<table>
<thead>
<tr>
<th>ACCOUNT</th>
<th>PERMISSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ACCOUNT_ID</td>
<td>+PERMISSION_ID</td>
</tr>
<tr>
<td>FIRST_NAME</td>
<td>NAME</td>
</tr>
<tr>
<td>LAST_NAME</td>
<td>#ACCOUNT_ID</td>
</tr>
<tr>
<td></td>
<td>ACCOUNT_PERMISSION_ORDER_IDX</td>
</tr>
</tbody>
</table>
```

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

**Simple array stored in join table**

If you want an array of non-entity objects be stored in a "join" table, you can follow this example. We have an **Account** that stores a Collection of addresses. These addresses are simply Strings. We define the annotations like this
public class Account {
    ...

    @Join(table="ACCOUNT_ADDRESSES")
    String[] addresses;
}

or using XML metadata

<class name="Account" table="ACCOUNT">
    ...
    <field name="permissions">
        <array/>
        <join table="ACCOUNT_ADDRESSES"/>
        <element column="ACCOUNT_ID"/>
        <order column="ACCOUNT_PERMISSION_ORDER_IDX"/>
    </field>
</class>

In the datastore the following is created

<table>
<thead>
<tr>
<th>ACCOUNT</th>
<th>ACCOUNT_ADDRESSES</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ACCOUNT_ID</td>
<td>+ACCOUNT_ID_OID</td>
</tr>
<tr>
<td>FIRSTNAME</td>
<td>ADDRESS</td>
</tr>
<tr>
<td>LASTNAME</td>
<td>+AUPT_PK_IDX</td>
</tr>
</tbody>
</table>

Use @Column on the field/method to define the column details of the element in the join table.
Interfaces

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

```java
public interface Shape
{
    double getArea();
}

public class Circle implements Shape
{
    double radius;
    ...
}

public class Square implements Shape
{
    double length;
    ...
}

public class Rectangle implements Shape
{
    double width;
    double length;
    ...
}
```

You then have a class that contains an object of this interface type

```java
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.

### 1-1 Interface Relation

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
@Entity
public class Square implements Shape {
    ...
}
@Entity
public class Circle implements Shape {
    ...
}
@Entity
public class Rectangle implements Shape {
    ...
}
```

The global way means that when mapping that field DataNucleus will look at all persistable 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:

```java
@Entity
public class ShapeHolder {

    @Extension(key="implementation-classes", value = "mydomain.Circle,mydomain.Rectangle,mydomain.Square")
    @Extension(key="mapping-strategy", value="identity")
    Shape shape;

    ...}
```

or using XML metadata:

```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:

```
SHAPEHOLDER
+SHAPEHOLDER_ID
#SHAPE_CIRCLE_ID
#SHAPE_RECTANGLE_ID
#SHAPE_SQUARE_ID

CIRCLE
+CIRCLE_ID
RADII

SQUARE
+SQUARE_ID
LENGTH

RECTANGLE
+RECTANGLE_ID
WIDTH
LENGTH
```

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.
and the datastore schema becomes

```
+SHAPEHOLDER_ID
<table>
<thead>
<tr>
<th>SHAPE</th>
</tr>
</thead>
</table>
+CIRCLE_ID | RADIUS |
+SQUARE_ID | LENGTH |
+RECTANGLE_ID | WIDTH | LENGTH |
```

and the column "SHAPE" will contain strings such as `mydomain.Circle:1` allowing retrieval of the related implementation object.

1-N Interface Relation

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
public class ShapeHolder {
    @Join
    @Extension(key="implementation-classes", value="mydomain.Circle,mydomain.Rectangle,mydomain.Square")
    @Extension(key="mapping-strategy", value="identity")
    Collection<Shape> shapes;
    ...
}
```
<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.

**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.
**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 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.

### 1-1/N-1 Object 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
@Persistent(types={mydomain.samples.vehicles.Car.class, mydomain.samples.vehicles.Motorbike.class})
Object occupier;

or using XML metadata

```xml
<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>
```

This will result in the following database schema.

```
+-- PARKING_SPACE
    + PARKING_SPACE_ID
    + LOCATION
    + OCCUPIER_CAR_ID
    + OCCUPIER_MOTORBIKE_ID

    | CAR
    + CAR_ID

    | MOTORBIKE
    + MOTORBIKE_ID
```

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.

```xml
<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.

1-N Object Relation

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.

Serialised Objects

By default a field of type `java.lang.Object` is stored as an instance of the underlying entity in the table of that object. If either your Object field represents non-entities or you simply wish to serialise the Object into the same table as the owning object, you need to specify it as serialized, like this

```java
class MyClass
{
    ...
    @Serialized
    Object myObject;
}
```

```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 persistable 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
<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.
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.

```java
@PersistenceCapable(embeddedOnly="true")
public class MyClass
{
    ...
}
```

or using XML metadata

```
<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 PCs** - where you have a 1-1 relationship and you want to embed the other persistable into the same table as the your object
- **Embedded Nested PCs** - like the first example except that the other object also has another persistable 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)

With respect to what types of fields you can have in an embedded class, DataNucleus supports all basic types, as well as 1-1/N-1 relations (where the *foreign-key* is at the embedded object side) and some 1-N/M-N relations.
whilst nested embedded members are supported, you cannot use recursive embedded objects since that would require potentially infinite columns in the owner table, or infinite embedded join tables.

Embedding persistable objects (1-1)

Applicable to RDBMS, Excel, OXXML, ODF, HBase, MongoDB, Neo4j, Cassandra, JSON

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 persistable 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;
    }

    ...
}

public 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

<table>
<thead>
<tr>
<th>COMPUTER</th>
<th>COMPUTERCARD</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
public class Computer
{
    @Embedded(nullIndicatorColumn="GRAPHICS_MANUFACTURER", members={
        @Persistent(name="manufacturer", column="GRAPHICS_MANUFACTURER"),
        @Persistent(name="type", column="GRAPHICS_TYPE"))
    private ComputerCard graphicsCard;

    @Embedded(nullIndicatorColumn="SOUND_MANUFACTURER", members={
        @Persistent(name="manufacturer", column="SOUND_MANUFACTURER"),
        @Persistent(name="type", column="SOUND_TYPE"))
    private ComputerCard soundCard;

    ...
}

@PersistenceCapable(embeddedOnly="true")
public class ComputerCard
{
    ...
}

or using XML metadata

</jdo>

<package name="com.mydomain.samples.embedded">
    <class name="Computer" identity-type="datastore" table="COMPUTER">
        ...
        <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" embedded-only="true">
        ...
    </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.

<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>SOUND_MANUFACTURER</td>
</tr>
<tr>
<td>SOUND_TYPE</td>
</tr>
</tbody>
</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.

```
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`

**Embedding Nested persistable objects**

Applicable to RDBMS, Excel, OOXML, ODF, HBase, MongoDB, Neo4j, Cassandra, JSON

In the above example we had an embedded persistable object within a persisted object. What if our embedded persistable object also contain another persistable object. So, using the above example
what if **ComputerCard** contains an object of type **Connector**?

```java
@PersistenceCapable(embeddedOnly="true")
public class ComputerCard
{
    Connector connector;

    public ComputerCard(String manufacturer, int type, Connector conn)
    {
        this.manufacturer = manufacturer;
        this.type = type;
        this.connector = conn;
    }

    ...
}

@PersistenceCapable(embeddedOnly="true")
public class Connector
{
    int type;
}
```

Well we want to store all of these objects into the same record in the COMPUTER table, so we define our XML metadata like this
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 **GRAPHICS_CONNECTOR_TYPE**, and **SOUND_CONNECTOR_TYPE** columns in the **COMPUTER** table.
you cannot specify nested embedded column information using JDO annotations; use XML metadata instead.

**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 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 an entity or of a "reference" type (Interface or Object) it will *always* be embedded, and this functionality here applies to embeddable entity 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 *Device*s. So we define our classes as

```java
import com.datanucleus.IDataManager;
import com.datanucleus.annotations.*;

@Entity
public class Network {
    @EmbeddedCollection
    private List<Device> devices;
}
```
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 XML MetaData like this:

```java
public class Network {
    @Element(inlineMaping={
        @Embedded(members={
            @Persistent(name="name", column="DEVICE_NAME"),
            @Persistent(name="ipAddress", column="DEVICE_IP_ADDR")
        })
    })
    private Collection<Device> devices = new HashSet<>();

    ...
}

@PersistenceCapable(embeddedOnly="true")
public class Device {
    private String name;
    private String ipAddress;

    ...
}
```

or using XML metadata
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.

<table>
<thead>
<tr>
<th>NETWORK</th>
<th>NETWORK_DEVICES</th>
</tr>
</thead>
<tbody>
<tr>
<td>+NETWORK_ID</td>
<td>+NETWORK_ID</td>
</tr>
<tr>
<td>NAME</td>
<td>+ADPT_PK_IDX</td>
</tr>
<tr>
<td></td>
<td>DEVICE_NAME</td>
</tr>
<tr>
<td></td>
<td>DEVICE_IP_ADDR</td>
</tr>
</tbody>
</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 XML metadata like this

```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>
  </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`

### 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 persistable or of a "reference" type (Interface or Object) it will always be embedded, and this functionality here applies to persistable 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 *Film*s, keyed by a String alias. So we define our classes as

```java
public class FilmLibrary
{
    private String owner;
    private Map<String, Film> films = new HashMap<>();

    ...
}

public class Film
{
    private String name;
    private String 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 XML MetaData like this
public class FilmLibrary
{
    @Key(column="FILM_ALIAS")
    @Value(embeddedMapping={
        @Embedded(members={
            @Persistent(name="name", column="FILM_NAME"),
            @Persistent(name="director", column="FILM_DIRECTOR")
        })
    })
    private Map<String, Film> films = new HashMap<>();

    ...
}

@PersistenceCapable(embeddedOnly="true")
public class Film
{
    private String name;
    private String director;

    ...
}
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`
Serialised 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 persistable 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`.

Queries cannot be performed on map keys/values stored as serialised.

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
public class Farm
{
    Collection<Animal> animals;
    ...
}
public class Animal
{
    String name;
    Type type;
}

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

<field name="animals" serialized="true">
    <collection element-type="Animal"/>
    <column name="ANIMALS"/>
</field>

So we make use of the serialized attribute of <field>. This specification results in a table like this

<table>
<thead>
<tr>
<th>FARM</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ID</td>
</tr>
<tr>
<td>NAME</td>
</tr>
<tr>
<td>ANIMALS</td>
</tr>
</tbody>
</table>

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-persistable 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 persistable 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

**Serialised Collection Elements**

Applicable to RDBMS
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

```java
@Persistent(table="FARM_ANIMALS")
@Element(serialised="true")
Collection<Animal> animals;
```

```xml
<field name="animals" table="FARM_ANIMALS">
  <collection element-type="Animal" serialised-element="true"/>
  <join column="FARM_ID_OID"/>
</field>
```

So we make use of the `serialized-element` attribute of `<collection>`. This specification results in tables like this

![Diagram of tables]

See also :-

- MetaData reference for `<collection>` element
- MetaData reference for `<join>` element
- Annotations reference for `@Element`

### Serialised Maps

Applicable to RDBMS, HBase, MongoDB

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
public class ClassRoom
{
    Map<String, Child> children;
...
}
public class Child
{
...
}

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
<field name="children" serialized="true">
    <map key-type="java.lang.String" value-type="Child"/>
    <column name="CHILDREN"/>
</field>
```

So we make use of the serialized attribute of <field>. This specification results in a table like this

<table>
<thead>
<tr>
<th>CLASSROOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>LEVEL</td>
</tr>
<tr>
<td>CHILDREN</td>
</tr>
</tbody>
</table>

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-persistable, non-persistable>, 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-persistable, persistable>, 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

**Serialised Map Keys/Values**

Applicable to RDBMS
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.

```java
@Persistent(table="CLASS_CHILDREN")
@Value(serialised="true")
Map<String, Child> children;
```

or using XML metadata

```xml
<class name="ClassRoom">
...
  <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:

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

**Serialised persistable Fields**

A field that is a persistable 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

```java
class ClassRoom {
    ...
    Teacher teacher;
}
class Teacher {...}
```

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

```java
@Serialized
Teacher teacher;
```

or using XML metadata

```xml
<field name="teacher" serialized="true">
    <column name="TEACHER" />
</field>
```

So we make use of the `serialized` attribute of `<field>`. This specification results in a table like this

<table>
<thead>
<tr>
<th>CLASSROOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ID</td>
</tr>
<tr>
<td>LEVEL</td>
</tr>
<tr>
<td>TEACHER</td>
</tr>
</tbody>
</table>

**Serialised Reference (Interface/Object) Fields**

Applicable to RDBMS

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

```java
public class ClassRoom {
    ...
    Teacher teacher;
}
class Teacher {...}
```
public class ClassRoom
{
    Person teacher;
    ...
}

public interface Person {...}

public class Teacher implements Person {...}

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:

    <field name="teacher" serialized="true">
        <column name="TEACHER"/>
    </field>

So we make use of the serialized attribute of <field>. This specification results in a table like this:

<table>
<thead>
<tr>
<th>CLASSROOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ID</td>
</tr>
<tr>
<td>LEVEL</td>
</tr>
<tr>
<td>TEACHER</td>
</tr>
</tbody>
</table>

See also:
- MetaData reference for <implements> element
- Annotations reference for @Serialized

### 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
@Persistent
@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.
We have shown earlier 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 datastore. Fields of a class are mapped to columns of a table (note that with some datastores it is not called a 'table' or 'column', but the concept is similar and we use ‘table’ and ‘column’ here to represent the mapping). 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
- Position of a column in a table
- RDBMS : Mapping a class to an RDBMS View
- RDBMS : Supported types for a field

### 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
Alternatively, if you really want to embody schema info in your class, you can use annotations

```java
@PersistenceCapable(table="ESTABLISHMENT")
public class Hotel {
    @Column(name="NAME")
    private String name;
    @Column(name="DIRECTION")
    private String address;
    @Column(name="PHONE")
    private String telephoneNumber;
    @Column(name="NUMBER_OF_ROOMS")
    private int numberOfRooms;
}
```

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`

**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

```xml
<class name="MyClass" table="MYCLASS">
    <datastore-identity>
        <column name="MY_PK_COLUMN"/>
    </datastore-identity>
    ...
</class>

<class name="MySubClass" table="MYSUBCLASS">
    <datastore-identity>
        <column name="MYSUB_PK_COLUMN"/>
    </datastore-identity>
    ...
</class>
```

Alternatively, you can specify these using annotations should you so wish.

```java
@PersistenceCapable(table="MYCLASS")
@DatastoreIdentity(column="MY_PK_COLUMN")
public class MyClass
{
    ...
}

@PersistenceCapable(table="MYSUBCLASS")
@DatastoreIdentity(column="MYSUB_PK_COLUMN")
public class MySubClass extends MyClass
{
    ...
}
```

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"
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 let's show an example how to do it.

```java
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" table="MYCLASS">
    <datastore-identity>
        <column name="MY_PK_COLUMN"/>
    </datastore-identity>
    ...
</class>

<class name="MySubClass" table="MYSUBCLASS">
    <inheritance strategy="new-table"/>
    <primary-key>
        <column name="MYSUB_PK_COLUMN"/>
    </primary-key>
    ...
</class>
```

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`

### 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 let's show an example how to do it.

```java
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:
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

See also :-

- Inheritance Guide - defining how to use inheritance between classes
- MetaData reference for <inheritance> element
- MetaData reference for <column> element
- MetaData reference for <primary-key> element
- Annotations reference for @Inheritance
- Annotations reference for @Column
- Annotations reference for @PrimaryKey
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`

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`

### 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).

**Position of column in a table**

With some datastores it is desirable to be able to specify the relative position of a column in the table schema. 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. See fields/properties column positioning docs for details.

**RDBMS : Views**

The standard situation with an RDBMS datastore is to map classes 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 (though not persisting into them). 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 (persistable) 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 represent the values from this View.

```java
package mydomain.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, here using XML but equally possible using annotations
Please note the following

- We've defined our class as using "nondurable" identity. This was mandatory in all versions up to 5.0.7 but after that is optional and you can use application/datastore identity also.

- 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.

- If you have a relation to the class represented by a View, you cannot expect it to create an FK in the View. The View will map on to exactly the members defined in the class it represents. i.e cannot have a 1-N FK uni relation to the class with the View.
We can now utilise this class within normal DataNucleus querying operation.

```java
Extent<SaleableProduct> e = pm.getExtent(SaleableProduct.class);
Iterator<SaleableProduct> iter = e.iterator();
while (iter.hasNext())
{
    SaleableProduct product = iter.next();
}
```

Hopefully that has given enough detail on how to create and access views from with a DataNucleus-enabled application.

**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.

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 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>Java Type</td>
<td>Number Columns</td>
<td>Queryable</td>
<td>JDBC Type(s)</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------</td>
<td>-----------</td>
<td>--------------------------------------------------</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>long</td>
<td>1</td>
<td>✓</td>
<td>BIGINT, NUMERIC, DOUBLE, DECIMAL, INTEGER</td>
</tr>
<tr>
<td>short</td>
<td>1</td>
<td>✓</td>
<td>SMALLINT, INTEGER, NUMERIC</td>
</tr>
<tr>
<td>boolean[]</td>
<td>1</td>
<td>✓ [5]</td>
<td>LONGVARBINARY, BLOB</td>
</tr>
<tr>
<td>byte[]</td>
<td>1</td>
<td>✓ [5]</td>
<td>LONGVARBINARY, BLOB</td>
</tr>
<tr>
<td>char[]</td>
<td>1</td>
<td>✓ [5]</td>
<td>LONGVARBINARY, BLOB</td>
</tr>
<tr>
<td>double[]</td>
<td>1</td>
<td>✓ [5]</td>
<td>LONGVARBINARY, BLOB</td>
</tr>
<tr>
<td>float[]</td>
<td>1</td>
<td>✓ [5]</td>
<td>LONGVARBINARY, BLOB</td>
</tr>
<tr>
<td>int[]</td>
<td>1</td>
<td>✓ [5]</td>
<td>LONGVARBINARY, BLOB</td>
</tr>
<tr>
<td>long[]</td>
<td>1</td>
<td>✓ [5]</td>
<td>LONGVARBINARY, BLOB</td>
</tr>
<tr>
<td>short[]</td>
<td>1</td>
<td>✓ [5]</td>
<td>LONGVARBINARY, BLOB</td>
</tr>
<tr>
<td>java.lang.Boolean</td>
<td>1</td>
<td>✓</td>
<td>BIT, CHAR('Y','N'), BOOLEAN, TINYINT, SMALLINT</td>
</tr>
<tr>
<td>java.lang.Byte</td>
<td>1</td>
<td>✓</td>
<td>TINYINT, SMALLINT, NUMERIC</td>
</tr>
<tr>
<td>java.lang.Character</td>
<td>1</td>
<td>✓</td>
<td>CHAR, INTEGER, NUMERIC</td>
</tr>
<tr>
<td>java.lang.Double</td>
<td>1</td>
<td>✓</td>
<td>DOUBLE, DECIMAL, FLOAT</td>
</tr>
<tr>
<td>java.lang.Float</td>
<td>1</td>
<td>✓</td>
<td>FLOAT, REAL, DOUBLE, DECIMAL</td>
</tr>
<tr>
<td>java.lang.Integer</td>
<td>1</td>
<td>✓</td>
<td>INTEGER, BIGINT, NUMERIC</td>
</tr>
<tr>
<td>java.lang.Long</td>
<td>1</td>
<td>✓</td>
<td>BIGINT, NUMERIC, DOUBLE, DECIMAL, INTEGER</td>
</tr>
<tr>
<td>java.lang.Short</td>
<td>1</td>
<td>✓</td>
<td>SMALLINT, INTEGER, NUMERIC</td>
</tr>
<tr>
<td>Java Type</td>
<td>Number Columns</td>
<td>Queryable</td>
<td>JDBC Type(s)</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>----------------</td>
<td>-----------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>java.lang.Short[]</td>
<td>1</td>
<td>✓ [5]</td>
<td>LONGVARBINARY, BLOB</td>
</tr>
<tr>
<td>java.lang.Number</td>
<td>1</td>
<td>✓</td>
<td>LONGVARBINARY, BLOB</td>
</tr>
<tr>
<td>java.lang.Object</td>
<td>1</td>
<td>✓</td>
<td>LONGVARBINARY, BLOB</td>
</tr>
<tr>
<td>java.lang.String [8]</td>
<td>1</td>
<td>✓</td>
<td>VARCHAR, CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [6], UNIQUEIDENTIFIER [7], XMLTYPE [9]</td>
</tr>
<tr>
<td>java.lang.StringBuffer [8]</td>
<td>1</td>
<td>✓</td>
<td>VARCHAR, CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [6], UNIQUEIDENTIFIER [7], XMLTYPE [9]</td>
</tr>
<tr>
<td>java.lang.Enum</td>
<td>1</td>
<td>✓</td>
<td>LONGVARBINARY, BLOB, VARCHAR, INTEGER</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>1</td>
<td>✓</td>
<td>DECIMAL, NUMERIC</td>
</tr>
<tr>
<td>java.math.BigInteger</td>
<td>1</td>
<td>✓</td>
<td>NUMERIC, DECIMAL</td>
</tr>
<tr>
<td>java.sql.Date</td>
<td>1</td>
<td>✓</td>
<td>DATE, TIMESTAMP</td>
</tr>
<tr>
<td>java.sql.Time</td>
<td>1</td>
<td>✓</td>
<td>TIME, TIMESTAMP</td>
</tr>
<tr>
<td>java.sql.Timestamp</td>
<td>1</td>
<td>✓</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>java.util.ArrayList</td>
<td>0</td>
<td>✓</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>java.util.BitSet</td>
<td>0</td>
<td>✗</td>
<td>LONGVARBINARY, BLOB</td>
</tr>
<tr>
<td>java.util.Calendar [3]</td>
<td>1 or 2</td>
<td>✗</td>
<td>INTEGER, VARCHAR, CHAR</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>VARCHAR, CHAR</td>
</tr>
<tr>
<td>java.util.Date</td>
<td>1</td>
<td>✓</td>
<td>TIMESTAMP, DATE, CHAR, BIGINT</td>
</tr>
<tr>
<td>java.util.Date[]</td>
<td>1</td>
<td>✓ [5]</td>
<td>LONGVARBINARY, BLOB</td>
</tr>
<tr>
<td>java.util.GregorianCalendar [2]</td>
<td>1 or 2</td>
<td>✗</td>
<td>INTEGER, VARCHAR, CHAR</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>
</tbody>
</table>

170
<table>
<thead>
<tr>
<th>Java Type</th>
<th>Number Columns</th>
<th>Queryable</th>
<th>JDBC Type(s)</th>
</tr>
</thead>
<tbody>
<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 [8]</td>
<td>1</td>
<td>✓</td>
<td>VARCHAR, CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [6], UNIQUEIDENTIFIER [7], XMLTYPE [9]</td>
</tr>
<tr>
<td>java.util.Locale[]</td>
<td>1</td>
<td>✓ [5]</td>
<td>LONGVARBINARY, BLOB</td>
</tr>
<tr>
<td>java.util.Map</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>java.util.Properties</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>java.util.PriorityQueue</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>java.util.Queue</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>java.util.Set</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>java.util.SortedMap</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>java.util.SortedSet</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>java.util.Stack</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>java.util.TimeZone [8]</td>
<td>1</td>
<td>✓</td>
<td>VARCHAR, CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [7], UNIQUEIDENTIFIER [8], XMLTYPE [9]</td>
</tr>
<tr>
<td>java.util.TreeMap</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>java.util.TreeSet</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>java.util.UUID [8]</td>
<td>1</td>
<td>✓</td>
<td>VARCHAR, CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [7], UNIQUEIDENTIFIER [8], XMLTYPE [9]</td>
</tr>
<tr>
<td>java.util.Vector</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>java.awt.Color [1]</td>
<td>4</td>
<td>✗</td>
<td>INTEGER</td>
</tr>
<tr>
<td>java.awt.Point [2]</td>
<td>2</td>
<td>✗</td>
<td>INTEGER</td>
</tr>
<tr>
<td>java.net.URI [8]</td>
<td>1</td>
<td>✓</td>
<td>VARCHAR, CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [7], UNIQUEIDENTIFIER [8], XMLTYPE [9]</td>
</tr>
<tr>
<td>Java Type</td>
<td>Number Columns</td>
<td>Queryable</td>
<td>JDBC Type(s)</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------</td>
<td>-----------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>java.net.URL [8]</td>
<td>1</td>
<td>✔</td>
<td>VARCHAR, CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [7], UNIQUEIDENTIFIER [8], XMLTYPE [9]</td>
</tr>
<tr>
<td>java.io.Serializable</td>
<td>1</td>
<td>✗</td>
<td>LONGVARBINARY, BLOB</td>
</tr>
<tr>
<td>Persistable</td>
<td>1</td>
<td>✔</td>
<td>[embedded]</td>
</tr>
<tr>
<td>Persistable[]</td>
<td>1</td>
<td>✔ [5]</td>
<td></td>
</tr>
</tbody>
</table>

- **1** - `java.awt.Color` - stored in 4 columns (red, green, blue, alpha). ColorSpace is not persisted.
- **2** - `java.awt.Point` - stored in 2 columns (x and y).
- **3** - `java.util.Calendar` - stored in 2 columns (milliseconds and timezone).
- **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.

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

DataNucleus provides support for the majority of the JDBC types with RDBMS. 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>✔</td>
<td>Only for PostgreSQL array type</td>
</tr>
<tr>
<td>BIGINT</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>BINARY</td>
<td>✔</td>
<td>Only for geospatial 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>JDBC Type</td>
<td>Supported</td>
<td>Restrictions</td>
</tr>
<tr>
<td>---------------------</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>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>NCHAR</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>NVARCHAR</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>Only for geospatial types on Oracle</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>

**Secondary Tables**

{info-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
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

```
package com.mydomain.samples.secondarytable;

public class Printer
{
    long id;
    String make;
    String model;

    String tonerModel;
    int 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

```
<class name="Printer" table="PRINTER">
    <join table="PRINTER_TONER" column="PRINTER_REFID"/>

    <field name="id" primary-key="true" column="PRINTER_ID"/>
    <field name="make" column="MAKE"/>
    <field name="model" column="MODEL"/>
    <field name="tonerModel" table="PRINTER_TONER" column="MODEL"/>
    <field name="tonerLifetime" table="PRINTER_TONER" column="LIFETIME"/>
</class>
```

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
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.

**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>

  ...  

</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
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.

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>
  ...  
</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)`

**Unique constraints**

![Applicable to RDBMS, NeoDatis, MongoDB.](image)

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. Let’s take the previous class, and show how to specify this
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">
  <field name="bookingType">
    <unique name="BOOKING_TYPE_CONSTRAINT"/>
  </field>
</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)

### Foreign Keys

![Applicable to RDBMS](https://example.com/applicable-to-rdbms.png)

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? Lets take an example
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

<class name="Hotel">
    <field name="rooms">
        <collection element-type="com.mydomain.samples.hotel.Room"/>
        <foreign-key name="HOTEL_ROOMS_FK" delete-action="cascade"/>
    </field>
</class>

So we now have given the datastore control over the cascade deletion strategy for objects stored in these tables. Please be aware that JDO 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 datanucleus.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
See also:-

- MetaData reference for <foreignkey> element
- Annotations reference for @ForeignKey
- Deletion of related objects using FK constraints

**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

```xml
<class name="Booking">
    <primary-key name="BOOKING_PK"/>
    ...
</class>
```

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

```xml
<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)

**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 its 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 for RDBMS is controlled by an IdentifierFactory, and DataNucleus provides a default implementation. You can provide your own RDBMS IdentifierFactory plugin to give your own preferred naming if so desired. You set the `RDBMS IdentifierFactory` by setting the persistence 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** RDBMS IdentifierFactory (default for JDO persistence)
- **jpa** RDBMS IdentifierFactory (default for JPA persistence)
- **datanucleus1** RDBMS IdentifierFactory (used in DataNucleus v1)
- **jpox** RDBMS IdentifierFactory (compatible with JPOX)

Generation of identifier names for non-RDBMS datastores is controlled by an NamingFactory, and DataNucleus provides a default implementation. You can provide your own NamingFactory plugin to give your own preferred naming if so desired. You set the NamingFactory by setting the persistence property `datanucleus.identifier.namingFactory`. 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** NamingFactory (default for JDO persistence for non-RDBMS)
- **jpa** NamingFactory (default for JPA persistence for non-RDBMS)

In describing the different possible naming conventions available out of the box with DataNucleus we’ll use the following example
**NamingFactory 'datanucleus2'**

This is default for JDO persistence to non-RDBMS datastores. Using the example above, the rules in this NamingFactory 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_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**

**NamingFactory 'jpa'**

The NamingFactory "jpa" aims at providing a naming policy consistent with the "JPA" specification. Using the same example above, the rules in this NamingFactory 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_MCLASS_ID* (FK to owner table) and *ELEMENTS1_ELEMENT_ID* (FK to element table)

- *MyClass.elements2* will be persisted into a column *ELEMENTS2_MCLASS_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*

**RDBMS 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 *RDBMS 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 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_MCLASS_ID_OWN* or *ELEMENTS2_MCLASS_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*

**RDBMS 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 *RDBMS 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_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`

**RDBMS IdentifierFactory 'jpa'**

The *RDBMS IdentifierFactory* "jpa" aims at providing a naming policy consistent with the "JPA" specification.

Using the same example above, the rules in this *RDBMS 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_OID` (FK to owner table) and `ELEMENTS1_ELEMENT_ID_OID` (FK to element table)
- `MyClass.elements2` will be persisted into a column `ELEMENTS2_MYCLASS_ID_OID` (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`
RDBMS IdentifierFactory 'jpx'

This RDBMS 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.

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
- **MixedCase**: No case changes are made to the name of the identifier provided by the user (class name or metadata).

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.

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)