



JPA Mapping Guide (v6.0)

Table of Contents

Classes	2
Entity Class	2
MappedSuperclass	3
Embeddable Class	3
Persistence Aware Class	4
Read-Only Class	4
SoftDelete	5
Inheritance	7
Discriminator	8
Strategy : SINGLE_TABLE	8
Strategy : JOINED	10
Strategy : TABLE_PER_CLASS	12
Mapped Superclasses	13
Identity	15
Application Identity	15
Datastore Identity	21
Nondurable Identity	23
Derived Identity Relationships	24
Versioning	37
Version Field/Property	37
Surrogate Version for Class	37
Auditing	39
Defining the Current User	40
Full Traceability Auditing	41
Multitenancy	42
Multitenancy via Discriminator in Table	42
Fields/Properties	44
Persistent Fields	44
Persistent Properties	44
Making a field/property non-persistent	45
Making a field/property read-only	46
Field Types	47
Primitive and java.lang Types	48
java.math types	49
Temporal Types (java.util, java.sql, java.time, Jodatetime)	50
Collection/Map types	52
Enums	53
Geospatial Types	55

Other Types	62
Arrays	62
Generic Type Variables	63
JPA Attribute Converters	65
Types extending Collection/Map	67
Column Adapters	68
Value Generation	70
ValueGeneration Strategy AUTO	70
ValueGeneration Strategy SEQUENCE	71
ValueGeneration Strategy IDENTITY	72
ValueGeneration Strategy TABLE	74
ValueGeneration Strategy "UUID"	75
ValueGeneration Strategy "Custom"	76
1-1 Relations	78
Unidirectional (ForeignKey)	78
Unidirectional (JoinTable)	80
Bidirectional (ForeignKey)	81
Bidirectional (JoinTable)	83
1-N Relations	85
equals() and hashCode()	85
Ordering of elements	86
Collection<Entity> Unidirectional (JoinTable)	87
Collection<Entity> Unidirectional (ForeignKey)	89
Collection<Entity> Bidirectional (JoinTable)	90
Collection<Entity> Bidirectional (ForeignKey)	92
Collection<Entity> (Shared JoinTable)	94
Collection<Entity> (Shared ForeignKey)	97
Collection<Simple> (JoinTable)	98
Collection<Simple> using AttributeConverter (Column)	99
Map<Simple, Entity> (JoinTable)	100
Map<Simple,Entity> Unidirectional (FK key stored in value)	102
Map<Simple,Entity> Bidirectional (FK key stored in value)	104
Map<Simple, Simple> (JoinTable)	106
Map<Simple, Simple> using AttributeConverter (Column)	107
Map<Entity, Entity> (JoinTable)	108
Map<Entity, Simple> (JoinTable)	110
N-1 Relations	111
Unidirectional (ForeignKey)	111
Unidirectional (JoinTable)	112
Bidirectional (ForeignKey)	114
Bidirectional (JoinTable)	114

M-N Relations	115
equals() and hashCode()	116
Using Set	116
Using Ordered Lists	118
Arrays	121
Single Column Arrays (serialised)	121
Simple array stored in join table	122
Entity array persisted into Join Tables	123
Entity array persisted using Foreign-Keys	124
Interfaces	126
1-1 Interface Relation	127
1-N Interface Relation	129
Dynamic Schema Updates (RDBMS)	130
java.lang.Object	131
1-1/N-1 Object Relation	131
1-N Object Relation	133
Serialised Objects	133
Embedded Fields	134
Embedded entity structure	134
Embedding Entities (1-1)	135
Embedding Nested Entities	140
Embedding Collection Elements	142
Embedding Map Keys/Values	145
Serialised Fields	148
Serialised Fields	148
Serialise to File	149
Datastore Schema	151
Tables and Column names	151
Column nullability and default values	153
Column types	154
Field/Column Positioning in a Table	160
Index Constraints	161
Unique Constraints	163
Foreign Key Constraints	164
Primary Key Constraints	166
RDBMS : Views	166
Secondary Tables	169
Datastore Identifiers	171

To implement a persistence layer with JPA you firstly need to map the classes and fields/properties that are involved in the persistence process to how they are represented in the datastore. This can be as simple as marking the classes as `@Entity` and defaulting the datastore definition, 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 for using metadata to map your entities.

When mapping a class for JPA you make use of *metadata*, and this *metadata* can be Java annotations, or can be XML metadata, or a mixture of both. 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 JPA spec only allows you to specify your mapping information using JPA metadata (annotations and/or `orm.xml`), DataNucleus JPA also allows you the option of using JDO metadata (annotations, and/or XML metadata). This is provided as a way of easily migrating across to JPA from JDO, for example. Consult the [DataNucleus JDO mappings docs](#) for details.



You cannot use a Java (14+) "record" with JPA, since it does not meet the requirements for persistence, but you can use it as a DTO type object that is populated from a query.

Classes

We have the following types of classes in DataNucleus JPA.

- **Entity** - persistable class with full control over its persistence.
- **MappedSuperclass** - persistable class that will not be persisted into its own table simply providing some fields to be persisted. Consequently an inheritance tree cannot just have a mapped superclass on its own.
- **Embeddable** - persistable class that is only persistable embedded into an entity class.
- **PersistenceAware** - a class that is not itself persisted, but that needs to access internals of

persistable classes. 

DataNucleus JPA imposes very little on classes used within the persistence process so, to a very large degree, you should design your classes as you would normally do and not design them to fit DataNucleus JPA. In fact, while JPA imposes that you must have a default constructor, the DataNucleus enhancer automatically adds one if not provided so you don't even need to provide that.



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



If defining a method *toString* in a JPA entity class, be aware that use of a persistable field will cause the load of that field if the object is managed and is not yet loaded.



If a JPA entity class is an element of a Java collection in another entity, you are advised to define *hashCode* and *equals* methods for reliable handling by Java collections.

Entity Class

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. Using annotations

```
@Entity
public class Hotel
{
    ...
}
```

or using XML metadata

```
<entity class="mydomain.Hotel">
    ...
</entity>
```

MappedSuperclass

Say we have an abstract base class *Building* with concrete subclass *Hotel* (as above). We want to persist some fields of *Building*, but it is abstract so will not have any objects of that type. So we make the class a *MappedSuperclass*, like this

```
@MappedSuperclass
public abstract class Building
{
    ...
}
```

or using XML metadata

```
<mapped-superclass class="mydomain.Building">
    ...
</mapped-superclass>
```

This is of particular relevance when considering [inheritance](#).

Embeddable Class

Here we have a class *ConstructionDetails* that we never need to persist individually, and it will only ever be persisted as part of an owner object (in this case *Building*). Since information from objects of this class will be persisted, we need to mark the class as *Embeddable*, like this

```
@Embeddable
public class ConstructionDetails
{
    ...
}
```

or using XML metadata

```
<embeddable class="mydomain.ConstructionDetails">
    ...
</embeddable>
```

and hereafter we can persist fields of type *ConstructionDetails*, as per [the Embedded Object guide](#).

Persistence Aware Class



With JPA you cannot access *public* fields of classes. DataNucleus allows an extension to permit this, but such classes need special enhancement. To allow this you need to annotate the class that will access these public fields (assuming it isn't an Entity) with the DataNucleus extension annotation `@PersistenceAware`, as follows

```
import org.datanucleus.api.jpa.annotations.PersistenceAware;

@PersistenceAware
public class MyClassThatAccessesPublicFields
{
    ...
}
```

See also :-

- [Annotations reference for @PersistenceAware](#)

Read-Only Class



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

```
import org.datanucleus.api.jpa.annotations.ReadOnly;

@Entity
@ReadOnly
public class MyClass
{
    ...
}
```

In practical terms this means that at runtime, if you try to persist an object of this type then an exception will be thrown. You can read objects of this type from the datastore just as you would for any persistable class

See also :-

- [Annotations reference for @ReadOnly](#)

SoftDelete



Applicable to RDBMS, MongoDB, Cassandra, *HBase*, *Neo4j*

With standard JPA when you delete an object from persistence it is deleted from the datastore. DataNucleus provides a useful ability to *soft delete* objects from persistence. In simple terms, any persistable types marked for soft deletion handling will have an extra column added to their datastore table to represent whether the record is soft-deleted. If it is soft deleted then it will not be visible at runtime thereafter, but will be present in the datastore.

You mark a persistable type for soft deletion handling like this

```
import org.datanucleus.api.jpa.annotations.SoftDelete;

@Entity
@SoftDelete
public class Hotel
{
    ...
}
```

You could optionally specify the *column* attribute of the `@SoftDelete` annotation to define the column name where this flag is stored.

Alternatively, using `orm.xml`

```
<entity class="mydomain.Hotel">
    <extension key="softdelete" value="true"/>
    <extension key="softdelete-column-name" value="DELETE_FLAG"/>
    ...
</entity>
```

Whenever any objects of type `Hotel` are deleted, like this

```
em.remove(myHotel);
```

the *myHotel* object will be updated to set the *soft-delete* flag to *true*.

Any call to *em.find* or query will not return the object since it is effectively deleted (though still present in the datastore).

If you want to view the object, you can specify the query hint `datanucleus.query.includeSoftDeletes` as *true* and the soft-deleted records will be visible.

This feature is still undergoing development, so not all aspects are feature complete.

See also :-

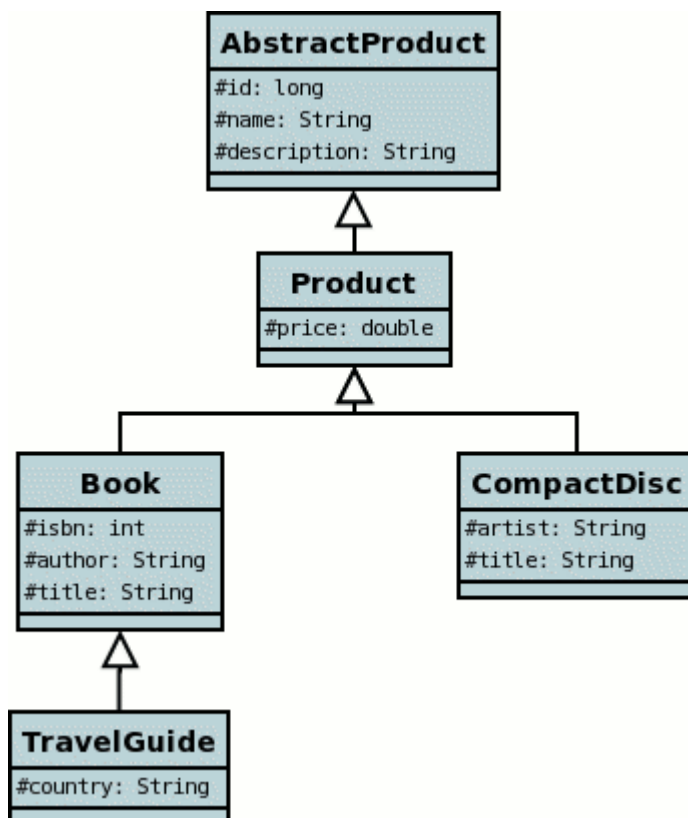
- [Annotations reference for @SoftDelete](#)

Inheritance

In Java it is a normal situation to have inheritance between classes. With JPA you have choices to make as to how you want to persist your classes for the inheritance tree. For each inheritance tree (for the root class) you select how you want to persist those classes information. You have the following choices.

- The *default strategy* is to select a class to have its fields persisted in the table of the base class. There is only one table per inheritance hierarchy. In JPA this is known as [SINGLE_TABLE](#)
- The next way is to have a table for each class in the inheritance hierarchy, and for each table to only hold columns for the fields of that class. Fields of superclasses are persisted into the table of the superclass. Consequently to get all field values for a subclass object a join is made of all tables of superclasses. In JPA this is referred to as [JOINED](#)
- The third way is like *JOINED* except that each table will also contain columns for all inherited fields. In JPA this is referred to as [TABLE_PER_CLASS](#)

In order to demonstrate the various inheritance strategies we need an example. Here are a few simple classes representing products in a (online) store. We have an abstract base class, extending this to provide something that we can represent any product by. We then provide a few specialisations for typical products. We will use these classes later when defining how to persistent these objects in the different inheritance strategies



As mentioned, the default JPA strategy is "SINGLE_TABLE", namely that the base class will have a table and all subclasses will be persisted into that same table. So if you don't specify an "inheritance strategy" in your root class this is what you will get.



You must specify the identity of objects in the root persistable class of the inheritance hierarchy. You cannot redefine it down the inheritance tree

See also:-

- [Metadata reference for <inheritance> element](#)
- [Metadata reference for <discriminator-column> element](#)
- [Annotations reference for @Inheritance](#)
- [Annotations reference for @DiscriminatorColumn](#)

Discriminator



Applicable to RDBMS, HBase, MongoDB

A *discriminator* is an extra "column" stored alongside data to identify the class of which that information is part. It is useful when storing objects which have inheritance to provide a quick way of determining the object type on retrieval. A discriminator in JPA will store the specified value (or the class name if you provide no value). You specify a discriminator as follows

```
<entity name="mydomain.Product">
  <discriminator-column name="OBJECT" discriminator-type="STRING"/>
  <discriminator-value>MyClass</discriminator-value>
  ...
</entity>
```

or with annotations

```
@Entity
@DiscriminatorColumn(name="OBJECT_TYPE", discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("MyClass")
public class Product {...}
```

Strategy : SINGLE_TABLE



Applicable to RDBMS

"SINGLE_TABLE" strategy is where the root class has a table and all subclasses are also persisted into that table. This corresponds to JDOs "new-table" for the root class and "superclass-table" for all subclasses. This has the advantage that retrieval of an object is a single DB call to a single table. It also has the disadvantage that the single table can have a very large number of columns, and database readability and performance can suffer, and additionally that a discriminator column is required.



When using SINGLE-TABLE DataNucleus will always use a discriminator (default column name is DTYPE) whether or not you define metadata for it.

In our example, let's ignore the **AbstractProduct** class for a moment and assume that **Product** is the base class (with the "id"). We have no real interest in having separate tables for the **Book** and **CompactDisc** classes and want everything stored in a single table **PRODUCT**. We change our MetaData as follows

```
<entity name="Product">
  <inheritance strategy="SINGLE_TABLE"/>
  <discriminator-value>PRODUCT</discriminator-value>
  <discriminator-column name="PRODUCT_TYPE" discriminator-type="STRING"/>
  <attributes>
    <id name="id">
      <column name="PRODUCT_ID"/>
    </id>
    ...
  </attributes>
</entity>
<entity name="Book">
  <discriminator-value>BOOK</discriminator-value>
  ...
</entity>
<entity name="TravelGuide">
  <discriminator-value>TRAVELGUIDE</discriminator-value>
  ...
</entity>
<entity name="CompactDisc">
  <discriminator-value>COMPACTDISC</discriminator-value>
  ...
</entity>
```

or using annotations

```

@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorValue("PRODUCT")
@DiscriminatorColumn(name="PRODUCT_TYPE", discriminatorType=DiscriminatorType.STRING)
public class Product {...}

@Entity
@DiscriminatorValue("BOOK")
public class Book {...}

@Entity
@DiscriminatorValue("TRAVELGUIDE")
public class TravelGuide {...}

@Entity
@DiscriminatorValue("COMPACTDISC")
public class CompactDisc {...}

```

This change of use of the **inheritance** element has the effect of using the **PRODUCT** table for all classes, containing the fields of **Product**, **Book**, **CompactDisc**, and **TravelGuide**. You will also note that we used a /discriminator-column_ element for the **Product** class. The specification above will result in an extra column (called **PRODUCT_TYPE**) being added to the **PRODUCT** table, and containing the "discriminator-value" of the object stored. So for a **Book** it will have "BOOK" in that column for example. This column is used in discriminating which row in the database is of which type. The final thing to note is that in our classes **Book** and **CompactDisc** we have a field that is identically named. With **CompactDisc** we have defined that its column will be called **DISCTITLE** since both of these fields will be persisted into the same table and would have had identical names otherwise - this gets around the problem.

PRODUCT
+PRODUCT_ID
PRICE
NAME
DESCRIPTION
AUTHOR
TITLE
COUNTRY
ARTIST
DISCTITLE
PRODUCT_TYPE

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

Strategy : JOINED



Applicable to RDBMS

"JOINED" strategy means that each entity in the inheritance hierarchy has its own table and that the

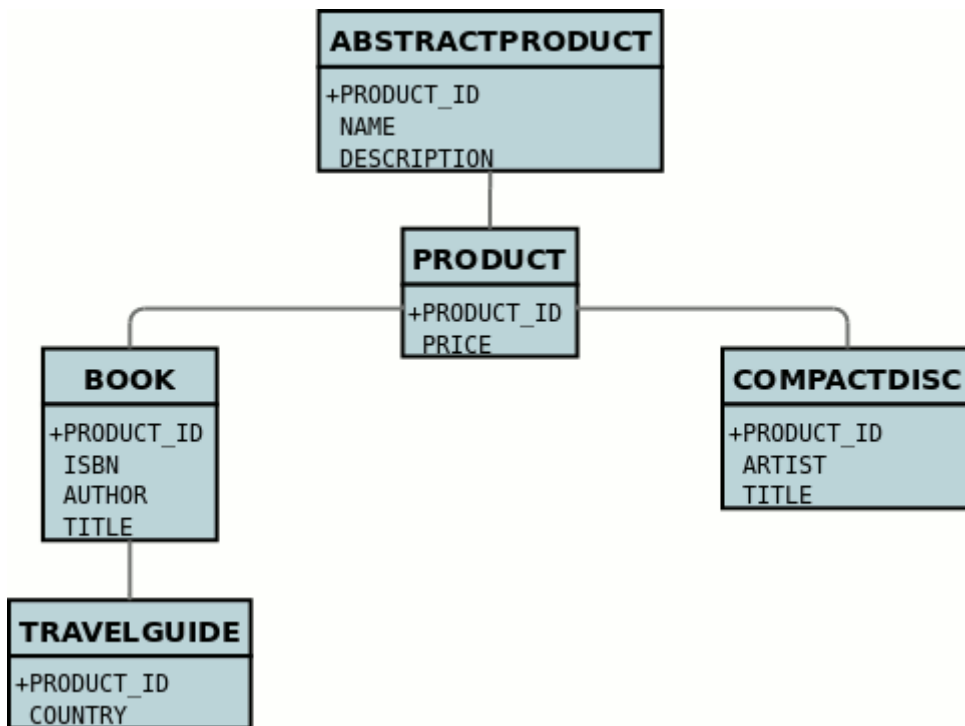
table of each class only contains columns for that class. Inherited fields are persisted into the tables of the superclass(es). This corresponds to JDOs "new-table" (for all classes in the inheritance hierarchy). This has the advantage of being the most normalised data definition. It also has the disadvantage of being slower in performance since multiple tables will need to be accessed to retrieve an object of a sub-type. Let's try an example using the simplest to understand strategy **JOINED**. We have the classes defined above, and we want to persist our classes each in their own table. We define the Meta-Data for our classes like this

```
<entity class="AbstractProduct">
  <inheritance strategy="JOINED"/>
  <attributes>
    <id name="id">
      <column name="PRODUCT_ID"/>
    </id>
    ...
  </attributes>
</entity>
<entity class="Product">
  ...
</entity>
<entity class="Book">
  ...
</entity>
<entity class="TravelGuide">
  ...
</entity>
<entity class="CompactDisc">
  ...
</entity>
```

or using annotations

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Product {...}
```

So we will have 5 tables - **ABSTRACTPRODUCT**, **PRODUCT**, **BOOK**, **COMPACTDISC**, and **TRAVELGUIDE**. They each contain just the fields for that class (and not any inherited fields, except the identity to join with).



In the above example, when we insert a TravelGuide object into the datastore, a row will be inserted into **ABSTRACTPRODUCT**, **PRODUCT**, **BOOK**, and **TRAVELGUIDE**.

Strategy : TABLE_PER_CLASS



Applicable to all datastores

This strategy is like "JOINED" except that in addition to each class having its own table, the table also holds columns for all inherited fields. So taking the same classes as used above


```

<entity class="AbstractProduct">
  <inheritance strategy="TABLE_PER_CLASS"/>
  <attributes>
    <id name="id">
      <column name="PRODUCT_ID"/>
    </id>
    ...
  </attributes>
</entity>
<entity class="Product">
  ...
</entity>
<entity class="Book">
  ...
</entity>
<entity class="TravelGuide">
  ...
</entity>
<entity class="CompactDisc">
  ...
</entity>

```

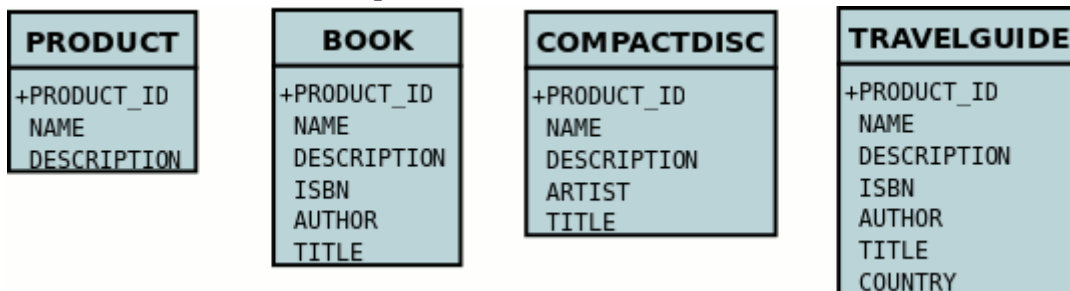
or using annotations

```

@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Product {...}

```

This then implies a datastore schema as follows



So any object of explicit type **Book** is persisted into the table **BOOK**. Similarly any **TravelGuide** is persisted into the table **TRAVELGUIDE**, etc. In addition if any class in the inheritance tree is abstract then it won't have a table since there cannot be any instances of that type. **DataNucleus currently has limitations when using a class using this inheritance as the element of a collection.**

Mapped Superclasses

JPA defines entities called "mapped superclasses" for the situation where you don't persist an actual object of a superclass type but that all subclasses of that type that are entities will also persist the values for the fields of the "mapped superclass". That is a "mapped superclass" has no table to store

its objects in a datastore. Instead its fields are stored in the tables of its subclasses. Let's take an example

```
<mapped-superclass class="AbstractProduct">
  <attributes>
    <id name="id">
      <column name="PRODUCT_ID"/>
    </id>
    ...
  </attributes>
</mapped-superclass>

<entity class="Product">
  ...
</entity>
```

In this case we will have a table for **Product** and the fields of **AbstractProduct** will be stored in this table. If the mapping information (column names etc) for these fields need setting then you should use `<attribute-override>` in the MetaData for **Product**.

Identity

All JPA-enabled persistable classes need to have an "identity" to be able to identify an object for retrieval and relationships. In strict JPA there is only 1 type of identity - *application identity*, where you have a field or field(s) of the entity that are used to define the identity. With DataNucleus JPA we allow 2 additional types of identity. So your options are

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

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 **Derived 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 `@MappedSuperclass` without any identity fields/properties as superclass, and then the base instantiable class is the first persistable class which has the identity field(s).



Your "identity" will be used for any joins to related objects, so consider carefully what field(s) you want to use in this respect.

Application Identity



Applicable to all datastores.

With **application identity** you are taking control of the specification of identity and define which field(s) in your class will define that identity. To specify that a class is to use **application identity**, you add the following to the MetaData for the class.

```
<entity class="mydomain.MyClass">
  <attributes>
    <id name="myPrimaryKeyField"/>
  </attributes>
</entity>
```

or, if we are using annotations

```
@Entity
public class MyClass
{
    @Id
    private long myPrimaryKeyField;
}
```

Application identity requires a primary key class when you have multiple identity fields. Each persistable class may define a different class for its primary key, and different persistable classes can use the same primary key class, as appropriate. To provide an example, when we have **multiple identity fields** we also require an **id-class**, so using XML

```
<entity class="mydomain.MyClass">
    <id-class class="mydomain.MyIdClass"/>
    <attributes>
        <id name="myPrimaryKeyField1"/>
        <id name="myPrimaryKeyField2"/>
    </attributes>
</entity>
```

or, if we are using annotations

```
@Entity
@IdClass(class=MyIdClass.class)
public class MyClass
{
    @Id
    private long myPrimaryKeyField1;

    @Id
    private long myPrimaryKeyField2;
}
```

With **application identity** the field(s) of the primary key (specified by `@IdClass` or `<id-class>`) will be present as field(s) of the class itself. The definition of the `@IdClass` is described further below.

See also:-

- [MetaData reference for <id> element](#)
- [Annotations reference for @Id](#)

Application Identity : Generating identities

For any field that is part of the "identity" you are responsible for defining how its value is set. Such field(s) can be set by the user, or can be generated. JPA defines [many ways of generating these](#)

[identities](#) and DataNucleus supports all of these and provides some more of its own besides.

Application Identity : Changing Identities

JPA doesn't define what happens if you change the identity (an identity field) of an object once persistent. **DataNucleus doesn't support changes to identities.**

Application Identity : Accessing objects by Identity

You access an object from its object class name and identity "value" as follows

```
MyClass myObj = em.find(MyClass.class, mykey);
```

If your class has a single identity field and no "IdClass" then the *mykey* is simply the value of that field. If you have defined your own "IdClass" then the *mykey* is the toString() form of the identity of your PK class.

Primary Key

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.



If you are thinking of using multiple primary key fields in a class we would urge you to consider using a single (maybe surrogate) primary key field instead for reasons of simplicity and performance. This also means that you can avoid the need to define your own primary key class.

Where you have more than 1 primary key field, you would define the PK class like this

```
<entity class="MyClass">
  <id-class class="MyIdClass"/>
  ...
</entity>
```

or using annotations

```
@Entity
@IdClass(class=MyIdClass.class)
public class MyClass
{
    ...
}
```

You now need to define the PK class to use. This is simplified for you because **if you have only one PK field then you don't 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*

Note that the types in **bold** are JPA standard types. Any others are DataNucleus extensions and, as always, [check the specific datastore docs](#) to see what is supported for your datastore.

Single PrimaryKey field

The simplest way of using **application identity** is where you have a single PK field, and in this case you use an inbuilt primary key class that DataNucleus provides, so you don't need to specify the *id-class*. Let's take an example

```
public class MyClass
{
    long id;
    ...
}
```

```
<entity class="MyClass">
  <attributes>
    <id name="id"/>
    ...
  </attributes>
</entity>
```

or using annotations

```
@Entity
public class MyClass
{
    @Id
    long id;
    ...
}
```

Note that we didn't specify the JPA "id-class". You will, of course, have to give the field a value before persisting the object, either by setting it yourself, or by using a [value-strategy](#) on that field.

PrimaryKey : Rules for User-Defined classes

If you wish to use **application identity** and don't want to use the "SingleFieldIdentity" built-in PK classes then you must define a Primary Key class of your own. You can't use classes like `java.lang.String`, or `java.lang.Long` directly. You must follow these rules when defining your primary key class.

- The Primary Key class must be public
- The Primary Key class must implement `Serializable`
- The Primary Key class must have a public no-arg constructor, which might be the default constructor
- The PrimaryKey class can have a constructor taking the primary key fields, or can use Java bean setters/getters
- The field types of all non-static fields in the Primary Key class must be serializable, and are recommended to be primitive, `String`, `Date`, or `Number` types
- All serializable non-static fields in the Primary Key class can be public, but package/protected/private should also be fine
- The names of the non-static fields in the Primary Key class must include the names of the primary key fields in the Entity, and the types of the common fields must be identical
- The `equals()` and `hashCode()` methods of the Primary Key class must use the value(s) of all the fields corresponding to the primary key fields in the JPA entity
- 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 an entity then you have [Derived 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 DataNucleus 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 above 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.



Your "id" class can store the target class name of the persistable object that it represents. This is useful where you want to avoid lookups of a class in an inheritance tree. To do this, add a field to your id-class called *targetClassName* and make sure that it is part of the *toString()* and *String constructor* code.

PrimaryKey Example - Multiple Field



Again, if you are thinking of using multiple primary key fields in a class we would urge you to consider using a single (maybe surrogate) primary key field instead for reasons of simplicity and performance. This also means that you can avoid the need to define your own primary key class.

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

```
@Entity
@IdClass(ComposedIdKey.class)
public class MyClass
{
    @Id
    String field1;

    @Id
    String field2;
    ...
}

public class ComposedIdKey implements Serializable
{
    public String targetClassName; // DataNucleus extension, storing the class name of
the persistable object
    public String field1;
    public String field2;

    /**
     * Default constructor.
     */
    public ComposedIdKey ()
```



```

{
}

/**
 * Constructor accepting same input as generated by toString().
 */
public ComposedIdKey(String value)
{
    StringTokenizer token = new StringTokenizer (value, "::");
    this.targetClassName = token.nextToken();
    this.field1 = token.nextToken ();
    this.field2 = token.nextToken ();
}

public boolean equals(Object obj)
{
    if (obj == this)
    {
        return true;
    }
    if (!(obj instanceof ComposedIdKey))
    {
        return false;
    }
    ComposedIdKey c = (ComposedIdKey)obj;

    return field1.equals(c.field1) && field2.equals(c.field2);
}

public int hashCode ()
{
    return this.field1.hashCode() ^ this.field2.hashCode();
}

public String toString ()
{
    // Give output expected by String constructor
    return this.targetClassName + this.field1 + "::" + this.field2;
}
}

```

Datastore Identity



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

While JPA defines support for [application identity](#) only, DataNucleus also provides support for

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

```
<entity class="mydomain.MyClass">
  <datastore-id/>
  ...
</entity>
```

or using annotations, for example

```
@Entity
@org.datanucleus.api.jpa.annotations.DatastoreIdentity
public class MyClass
{
    ...
}
```

Please note that since the JPA XML metadata is poorly designed it is not possible to specify datastore identity using XML, you have to use the annotations.

Datastore Identity : Generating identities

By choosing **datastore identity** you are handing the process of identity generation to the DataNucleus. This does not mean that you haven't got any control over how it does this. JPA defines many ways of generating these identities and DataNucleus supports all of these and provides some more of its own besides.

Defining which one to use is a simple matter of adding a MetaData element to your classes definition, like this

```
@Entity
@org.datanucleus.api.jpa.annotations.DatastoreIdentity(generationType=GenerationType.T
ABLE)
public class MyClass
{
    ...
}
```

See also:-

- [Identity Generation Guide](#) - strategies for generating ids
- [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.

```
import org.datanucleus.api.jpa.NucleusJPAHelper;  
  
Object idKey = NucleusJPAHelper.getDatastoreIdForEntity(obj);
```

From this you can use the "find" method to retrieve the object

```
MyClass myObj = em.find(MyClass.class, idKey);
```

Nondurable Identity



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

JPA requires that all objects have an identity. DataNucleus provides a vendor extension that allows objects of a class to 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 DataNucleus you would add the following to the MetaData for the class.

```
<entity class="mydomain.MyClass">  
  <nondurable-id/>  
  ...  
</entity>
```

or using annotations, for example

```
@Entity  
@org.datanucleus.api.jpa.annotations.NonDurableId  
public class MyClass  
{  
  ...  
}
```

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

Derived Identity Relationships

An derived identity relationship is a relationship between two objects of two classes in which the child object must coexist with the parent object and where the primary key of the child includes the Entity object of the parent. So effectively the key aspect of this type of relationship is that the primary key of one of the classes includes a Entity field (hence why is is referred to as *Derived 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 \(key stored in value\)](#)



A more efficient relationship is to handle this as a standard 1-1/1-N relation, with individual id field in the related class, and then define a unique constraint on the related class. This then foregoes the "derived identity" relationship idea and you can stick to standard relationships.



In standard JPA, if the entity that is part of the id of the derived entity has a single long field then you can put a *long* field in the identity class of the derived entity. In DataNucleus you **cannot** do this currently, and should define the `@IdClass` of the entity being contained and use that type in the identity class of the derived entity.



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



When using derived identity, it is best practice to define an `@IdClass` for any entity that is part of the primary key, and **not** rely on the built-in identity types.

Derived 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".

```

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

```

@Entity
public class User
{
    @Id
    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;
        }
    }

@Entity
public class Account
{
    @Id
    @OneToOne
    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

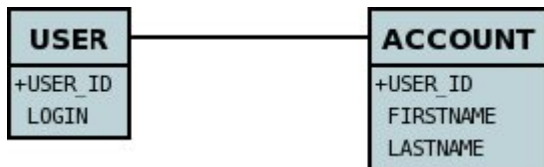
```

<entity-mappings>
  <entity class="mydomain.User">
    <table name="USER"/>
    <id-class class="mydomain.User.PK"/>
    <attributes>
      <id name="id">
        <column name="USER_ID"/>
      </id>
      <basic name="login">
        <column name="LOGIN" length="20"/>
      </basic>
    </attributes>
  </entity>

  <entity class="mydomain.Account">
    <table name="ACCOUNT"/>
    <id-class class="mydomain.Account.PK"/>
    <attributes>
      <id name="user">
        <column name="USER_ID"/>
      </id>
      <basic name="firstName">
        <column name="FIRSTNAME" length="50"/>
      </basic>
      <basic name="secondName">
        <column name="LASTNAME" length="50"/>
      </basic>
      <one-to-one name="user"/>
    </attributes>
  </entity>
</entity-mappings>

```

So now we have the following datastore schema



Things to note:-

- In the child Primary Key class, you must have a field with the same name as the relationship in the child class, and the field in the child Primary Key class must be the same type as the Primary Key class of the parent
- See also the [general instructions for Primary Key classes](#)
- You can only have one "Account" object linked to a particular "User" object since the FK to the "User" is now the primary key of "Account". To remove this restriction you could also add a "long id" to "Account" and make the "Account.PK" a composite primary-key

Derived 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.

```

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

```

@Entity
public class Account
{
    @Id
    long id;
  
```



```

@OneToMany
Set<Address> 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;
    }
}

@Entity
public class Address
{
    @Id
    long id;

    @Id

```

```

@ManyToOne
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;
    }
}
}

```

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

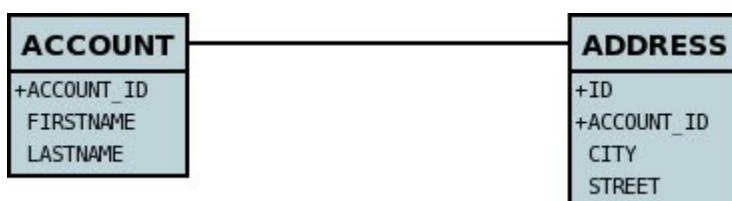
```

<entity-mappings>
  <entity class="mydomain.Account">
    <table name="ACCOUNT"/>
    <id-class class="mydomain.Account.PK"/>
    <attributes>
      <id name="id">
        <column name="ACCOUNT_ID"/>
      </id>
      <basic name="firstName">
        <column name="FIRSTNAME" length="50"/>
      </basic>
      <basic name="secondName">
        <column name="LASTNAME" length="50"/>
      </basic>
      <one-to-many name="addresses" mapped-by="account"/>
    </attributes>
  </entity>

  <entity class="mydomain.Address">
    <table name="ADDRESS"/>
    <id-class class="mydomain.Address.PK"/>
    <attributes>
      <id name="id">
        <column name="ID"/>
      </id>
      <id name="account">
        <column name="ACCOUNT_ID"/>
      </id>
      <basic name="city">
        <column name="CITY"/>
      </basic>
      <basic name="street">
        <column name="STREET"/>
      </basic>
      <many-to-one name="account"/>
    </attributes>
  </entity>
</entity-mappings>

```

So now we have the following datastore schema



Things to note :-

- In the child Primary Key class, you must have a field with the same name as the relationship in

the child class, and the field in the child Primary Key class must be the same type as the Primary Key class of the parent

- See also the [general instructions for Primary Key classes](#)
- If we had omitted the "id" field from "Address" it would have only been possible to have one "Address" in the "Account" "addresses" collection due to PK constraints. For that reason we have the "id" field too.

Derived 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.

```
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

```
@Entity
public class Account
{
    @Id
    long id;

    @OneToMany
    Map<String, Address> addresses;

    ... (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;
    }
}

@Entity
public class Address
{
    @Id
    String alias;

    @Id
    @ManyToOne
    Account account;

    .. (remainder of Address class)
}

```

* 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

```

<entity-mappings>
  <entity class="mydomain.Account">
    <table name="ACCOUNT"/>
    <id-class class="mydomain.Account.PK"/>
    <attributes>
      <id name="id">
        <column name="ACCOUNT_ID"/>
      </id>
      <basic name="firstName">
        <column name="FIRSTNAME" length="50"/>
      </basic>
      <basic name="secondName">
        <column name="LASTNAME" length="50"/>
      </basic>
      <one-to-many name="addresses" mapped-by="account">
        <map-key name="alias"/>
      </one-to-many>
    </attributes>
  </entity>

  <entity class="mydomain.Address">
    <table name="ADDRESS"/>
    <id-class class="mydomain.Address.PK"/>
    <attributes>
      <id name="account">
        <column name="ACCOUNT_ID"/>
      </id>
      <id name="alias">
        <column name="KEY"/>
      </id>
      <basic name="city">
        <column name="CITY"/>
      </basic>
      <basic name="street">
        <column name="STREET"/>
      </basic>
      <many-to-one name="account"/>
    </attributes>
  </entity>
</entity-mappings>

```



So now we have the following datastore schema

Things to note :-

- In the child Primary Key class, you must have a field with the same name as the relationship in

the child class, and the field in the child Primary Key class must be the same type as the Primary Key class of the parent

- See also the [general instructions for Primary Key classes](#)
- If we had omitted the "alias" field from "Address" it would have only been possible to have one "Address" in the "Account" "addresses" collection due to PK constraints. For that reason we have the "alias" field too as part of the PK.

Versioning

JPA allows objects of classes to be versioned. The version is typically used as a way of detecting if the object has been updated by another thread or EntityManager since retrieval using the current EntityManager - for use by [Optimistic Locking](#).

Version Field/Property

The standard JPA mechanism for versioning of objects is to mark a field of the class to store the version.



The field must be of one of the following types : int, long, short, *java.lang.Integer*, *java.lang.Long*, *java.lang.Short*, *java.sql.Timestamp*, *java.sql.Date*, *java.sql.Time*, *java.util.Date*, *java.util.Calendar*, *java.time.Instant*.

With JPA you can specify the details of this **version field** as follows

```
@Entity
public class User
{
    ...

    @Version
    int version;

    ...
}
```

or using XML metadata

```
<entity name="mydomain.User">
    <attributes>
        ...
        <version name="version"/>
        ...
    </attributes>
</entity>
```

The specification above will use the "version" field for storing the version of the object. DataNucleus will use a "version-number" strategy for populating the value.

Surrogate Version for Class



While the above mechanism should always be used for portability, DataNucleus also supports a surrogate version for objects of a class. With this you don't have a particular field that stores the version and instead DataNucleus persists the version in the datastore with the field values in its own "column". You do this as follows.

```
import org.datanucleus.api.jpa.annotations.SurrogateVersion;

@Entity
@SurrogateVersion
public class User
{
    ...
}
```

or using XML metadata

```
<entity name="mydomain.User">
    <surrogate-version column="version"/>
    ...
</entity>
```

To access the "surrogate" version, you can make use of the following method

```
import org.datanucleus.api.jpa.NucleusJPAHelper;

Object version = NucleusJPAHelper.getSurrogateVersionForEntity(obj);
```

Auditing



Applicable to RDBMS

With standard JPA you have no annotations available to automatically add timestamps and user names 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.

```
import org.datanucleus.api.jpa.annotations.CreateTimestamp;
import org.datanucleus.api.jpa.annotations.CreateUser;
import org.datanucleus.api.jpa.annotations.UpdateTimestamp;
import org.datanucleus.api.jpa.annotations.UpdateUser;

@Entity
public class Hotel
{
    @Id
    long id;

    @CreateTimestamp
    Timestamp createTimestamp;

    @CreateUser
    String createUser;

    @UpdateTimestamp
    Timestamp updateTimestamp;

    @UpdateUser
    String updateUser;

    ...
}
```

In the above example we have 4 fields in the class that will have columns in the datastore. The field `createTimestamp` and `createUser` will be persisted at INSERT with the Timestamp and current user for the insert. The field `updateTimestamp` and `updateUser` will be persisted whenever any update is made to the object in the datastore, with the Timestamp and current user for the update.

If you instead wanted to define this in `orm.xml` then do it like this

```

<entity class="mydomain.Hotel">
  <attributes>
    <basic name="createTimestamp">
      <extension key="create-timestamp" value="true"/>
    </basic>
    <basic name="createUser">
      <extension key="create-user" value="true"/>
    </basic>
    <basic name="updateTimestamp">
      <extension key="update-timestamp" value="true"/>
    </basic>
    <basic name="updateUser">
      <extension key="update-user" value="true"/>
    </basic>
  </attributes>
</entity>

```



Any field marked as `@CreateTimestamp` / `@UpdateTimestamp` needs to be of type `java.sql.Timestamp` or `java.time.Instant`.



You can only annotate a single field/property per class with each of these annotations (e.g a single `@CreateUser`)

Defining the Current User

The timestamp can be automatically generated for population here, but clearly the *current user* is not available as a standard, and so we have to provide a mechanism for setting it. You have 2 ways to do this; choose the one that is most appropriate to your situation

- Specify the persistence property **datanucleus.CurrentUser** on the EMF to be the current user to use. Optionally you can also specify the same persistence property on each PM if you have a particular user for each EM.
- Define an implementation of the DataNucleus interface `org.datanucleus.store.schema.CurrentUserProvider`, and specify it on EMF creation using the property **datanucleus.CurrentUserProvider**. This is defined as follows

```

public interface CurrentUserProvider
{
    /** Return the current user. */
    String currentUser();
}

```

So you could, for example, store the current user in a thread-local and return it via your implementation of this interface.

Full Traceability Auditing

DataNucleus doesn't currently provide a full traceability auditing capability, whereby you can track all changes to every relevant field. This would involve having a mirror table for each persistable class and, for each insert/update of an object, would require 2 SQL statements to be issued. The obvious consequence would be to slow down the persistence process.

Should your organisation require this, we could work with you to provide it. Please contact us if interested.

Multitenancy

On occasion you need to share a data model with other user-groups or other applications and where the model is persisted to the same structure of datastore. There are three ways of handling this with DataNucleus.

- **Separate Database per Tenant** - have a different database per user-group/application. In this case you will have a separate EMF for each database, and manage use of the appropriate EMF yourself.
- **Separate Schema per Tenant** - as the first option, except use different schemas. In this case you will have a separate EMF for each database schema, and manage use of the appropriate EMF yourself.
- **Same Database/Schema but with a Discriminator in affected Table(s)** - this is described below. In this case you will have a single EMF, and DataNucleus will manage selecting appropriate data for the tenant in question. This is described below.

Multitenancy via Discriminator in Table



Applicable to RDBMS, HBase, MongoDB, Neo4j, Cassandra

To define that a class is to be "multi-tenant" and hence have a discriminator column added you need to do as follows.

```
<entity class="mydomain.MyClass">
  <extension key="multitenant" value="true"/>
  <extension key="multitenancy-column-name" value="TENANT"/>
  <extension key="multitenancy-column-length" value="24"/>
  ...
</entity>
```

or using annotations

```
@Entity
@MultiTenant(column="TENANT", columnLength=24)
public class MyClass
{
    ...
}
```

By default (without the specification of column name/length) this will add a column **TENANT_ID** to each primary table, of String-based (255) type.



In all DataNucleus prior to v6.0.0-m3 you enabled multitenancy globally (so the discriminator was always added, and you had to disable on a class-by-class basis). This is now changed so that you **explicitly** define which classes need a multitenancy discriminator.

Fields/Properties

Once we have defined a class to be persistable (as either *Entity*, *MappedSuperclass*, or *Embedded*), 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.

An example of how to define the persistence of a field is shown below

```
@Entity
public class MyClass
{
    @Basic
    Date birthday;

    @Transient
    String someOtherField;
}
```

So, using annotations, we have marked this class as persistent, and the field *birthday* also as persistent, whereas field *someOtherField* is not persisted. Using XML MetaData we would have done

```
<entity name="mydomain.MyClass">
  <attributes>
    <basic name="birthday"/>
    <transient name="someOtherField"/>
  </attributes>
</entity>
```

Please note that the field Java type defines whether it is, by default, persistable. With DataNucleus the majority of types that you will come across in your application are 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.



The JavaBean specification is to have a getter method with signature *{type} getMyField()* and a setter method with signature *void setMyField({type} arg)*, where the property name is then *myField*, and the type is *{type}*.

An example of how to define the persistence of a property is shown below

```
@Entity
public class MyClass
{
    @Basic
    Date getBirthday()
    {
        ...
    }

    void setBirthday(Date date)
    {
        ...
    }
}
```

So, using annotations, we have marked this class as persistent, and the getter is marked as persistent. By default a property is non-persistent, so we have no need in specifying the *someOtherField* as transient. Using XML MetaData we would have done

```
<entity name="mydomain.MyClass">
    <attributes>
        <basic name="birthday"/>
    </attributes>
</entity>
```

Making a field/property non-persistent

If you have a field/property that you don't want to persist, just mark it as *transient*, like this

```
@Transient
String unimportantField;
```

or with XML

```
<entity name="mydomain.MyClass">
  <attributes>
    <transient name="unimportantField"/>
  </attributes>
</entity>
```

Making a field/property read-only



If you want to make a member read-only you can do it like this.

```
<entity name="mydomain.MyClass">
  <attributes>
    <basic name="myField">
      <column insertable="false" updatable="false"/>
    </basic>
  </attributes>
</entity>
```

or with Annotations

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

@Entity
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 JPA specification define lists of types that are required to be supported by all implementations of those specifications. This support can be conveniently split into two parts

- **Primary Types** : An object that can be *referred to* (object reference, providing a relation) and that has an "identity" is termed a **primary type**. DataNucleus supports the following Java types as primary : any *Entity* that has its own identity, *interface* where it represents an *Entity*, or **java.lang.Object** where it represents an *Entity*.
- **Secondary Types** : An object that does not have an "identity" is termed a **secondary type**. This is something like a String or Date field in a class, or alternatively a Collection (that contains other objects), or an embedded *Entity*. The sections below shows the currently supported secondary java types in DataNucleus. The tables in these sections show
 - **EAGER** : 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 (such as Collection, Map).
 - **PK** : whether the field can be used as part of the primary-key



The JPA spec places restrictions on what you can and can't do with a **@MappedSuperclass** type. DataNucleus largely treats it like any entity, so can be used for querying, or relations.



The JPA spec has a *required list of types* that a provider should support is very basic. DataNucleus provides all of those and many many more, see below.



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.

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 [JPA AttributeConverter](#) if you can easily convert the type into a String or Long.



You can add support for a Java type using the [the Java Types](#) .



You can also define more specific support for it with RDBMS datastores using the [the RDBMS Java Types](#) .

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.



When your field type is a type that is mutable it will be replaced by a "wrapper" when the owning object is managed. By default this wrapper type will be based on the *instantiated* type. You can change this to use the *declared* type by setting the persistence property **datanucleus.type.wrapper.basis** to *declared*.

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).

Java Type	EAGER?	Proxy?	PK?	Comments
boolean	✓	✗	✓	Persisted as BOOLEAN , Integer (i.e 1,0), String (i.e 'Y','N').
byte	✓	✗	✓	
char	✓	✗	✓	
double	✓	✗	✗	
float	✓	✗	✗	
int	✓	✗	✓	
long	✓	✗	✓	
short	✓	✗	✓	
java.lang.Boolean	✓	✗	✓	Persisted as BOOLEAN , Integer (i.e 1,0), String (i.e 'Y','N').
java.lang.Byte	✓	✗	✓	
java.lang.Character	✓	✗	✓	
java.lang.Double	✓	✗	✗	
java.lang.Float	✓	✗	✗	
java.lang.Integer	✓	✗	✓	
java.lang.Long	✓	✗	✓	
java.lang.Short	✓	✗	✓	

Java Type	EAGER?	Proxy?	PK?	Comments
java.lang.Number	✓	✗	✗	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.
java.lang.String	✓	✗	✓	
java.lang.StringBuffer	✓	✗	✓	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.
java.lang.StringBuilder	✓	✗	✓	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.
java.lang.Class	✓	✗	✗	Persisted as String.

java.math types

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

Java Type	EAGER?	Proxy?	PK?	Comments
java.math.BigDecimal	✓	✗	✗	Persisted as DOUBLE or String. String can be used to retain precision.
java.math.BigInteger	✓	✗	✓	Persisted as INTEGER or String. String can be used to retain precision.

CHECK constraints



Supported for RDBMS datastores.

If you want to constraint the column where a standard numeric/character field is stored to only have particular values you can put a CHECK constraint on the column contents in the datastore. You specify it like this

```
@Extension(key="check-constraint-values", value="1,2,4")
int groupNumber;
```

This results in a column defined like

```
GRP_NUMBER INTEGER CHECK (GRP_NUMBER IN (1, 2, 4)),
```

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.

Java Type	EAGER?	Proxy?	PK?	Comments
java.sql.Date	✓	✓ (1)	✓	Persisted as DATE , String, DATETIME or Long.
java.sql.Time	✓	✓ (1)	✓	Persisted as TIME , String, DATETIME or Long.
java.sql.Timestamp	✓	✓ (1)	✓	Persisted as TIMESTAMP , String or Long.
java.util.Calendar	✓	✓	✗	Persisted as TIMESTAMP (inc Timezone) , DATETIME, String, or as (Long, String) storing millis + timezone respectively
java.util.GregorianCalendar	✓	✓	✗	Persisted as TIMESTAMP (inc Timezone) , DATETIME, String, or as (Long, String) storing millis + timezone respectively
java.util.Date	✓	✓ (1)	✓	Persisted as DATETIME , String or Long.
java.util.TimeZone	✓	✗	✓	Persisted as String.
java.time.LocalDateTime	✓	✗	✗	Persisted as DATETIME , String, or Timestamp.
java.time.LocalTime	✓	✗	✗	Persisted as TIME , String, or Long.
java.time.LocalDate	✓	✗	✗	Persisted as DATE , String, or DATETIME.
java.time.OffsetDateTime	✓	✗	✗	Persisted as Timestamp , String, or DATETIME.
java.time.OffsetTime	✓	✗	✗	Persisted as TIME , String, or Long.

Java Type	EAGER?	Proxy?	PK?	Comments
java.time.MonthDay	✓	✗	✗	Persisted as String , DATE, or as (Integer,Integer) with the latter being month+day respectively.
java.time.YearMonth	✓	✗	✗	Persisted as String , DATE, or as (Integer,Integer) with the latter being year+month respectively.
java.time.Year	✓	✗	✗	Persisted as Integer , or String.
java.time.Period	✓	✗	✗	Persisted as String .
java.time.Instant	✓	✗	✗	Persisted as TIMESTAMP , String, Long, or DATETIME.
java.time.Duration	✓	✗	✗	Persisted as String , Double (secs.nanos), or Long (secs).
java.time.ZoneId	✓	✗	✗	Persisted as String .
java.time.ZoneOffset	✓	✗	✗	Persisted as String .
java.time.ZonedDateTime	✓	✗	✗	Persisted as Timestamp , or String.
org.joda.time.DateTime	✓	✗	✗	Requires datanucleus-jodatime plugin. Persisted as TIMESTAMP or String.
org.joda.time.LocalDateTime	✓	✗	✗	Requires datanucleus-jodatime plugin. Persisted as TIME or String.
org.joda.time.LocalDate	✓	✗	✗	Requires datanucleus-jodatime plugin. Persisted as DATE or String.
org.joda.time.LocalDateTime	✓	✗	✗	Requires datanucleus-jodatime plugin. Persisted as TIMESTAMP , or String.
org.joda.time.Duration	✓	✗	✗	Requires datanucleus-jodatime plugin. Persisted as String or Long.
org.joda.time.Interval	✓	✗	✗	Requires datanucleus-jodatime plugin. Persisted as String or (TIMESTAMP, TIMESTAMP).
org.joda.time.Period	✓	✗	✗	Requires datanucleus-jodatime plugin. Persisted as String .



(1) By default the legacy Java temporal types (java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp) are treated as mutable even though most of their mutator methods are deprecated. If you never call the setTime() method on fields of these types you can get an efficiency benefit by setting persistence property **datanucleus.type.treatJavaUtilDateAsMutable** to *false* and they will not be wrapped by a proxy when handed back to you.

Collection/Map types

DataNucleus supports a very wide range of collection, list and map types. It also supports the declared type to be not just interfaces ([Collection](#), [Set](#), [List](#), [Map](#)) but also common implementations (unlike many other JPA providers which only support the interface types).

Java Type	EAGER?	Proxy ?	PK?	Comments
java.util.Collection	✗	✓	✗	See the 1-N Mapping Guide
java.util.List	✗	✓	✗	See the 1-N Mapping Guide
java.util.Map	✗	✓	✗	See the 1-N Mapping Guide
java.util.Queue	✗	✓	✗	The comparator is specifiable via the metadata extension <i>comparator-name</i> (see below). See the 1-N Mapping Guide
java.util.Set	✗	✓	✗	See the 1-N Collections Guide
java.util.SortedMap	✗	✓	✗	The comparator is specifiable via the metadata extension <i>comparator-name</i> (see below). See the 1-N Mapping Guide
java.util.SortedSet	✗	✓	✗	The comparator is specifiable via the metadata extension <i>comparator-name</i> (see below). See the 1-N Collections Guide
java.util.ArrayList	✗	✓	✗	See the 1-N Mapping Guide
java.util.BitSet	✗	✓	✗	Persisted as collection by default, but will be stored as String when the datastore doesn't provide for collection storage
java.util.HashMap	✗	✓	✗	See the 1-N Mapping Guide
java.util.HashSet	✗	✓	✗	See the 1-N Mapping Guide
java.util.Hashtable	✗	✓	✗	See the 1-N Maps Guide
java.util.LinkedHashMap	✗	✓	✗	Persisted as a Map currently. No List-ordering is supported. See the 1-N Maps Guide
java.util.LinkedHashSet	✗	✓	✗	Persisted as a Set currently. No List-ordering is supported. See the 1-N Collections Guide
java.util.LinkedList	✗	✓	✗	See the 1-N Mapping Guide
java.util.Properties	✗	✓	✗	See the 1-N Mapping Guide
java.util.PriorityQueue	✗	✓	✗	The comparator is specifiable via the metadata extension <i>comparator-name</i> (see below). See the 1-N Mapping Guide
java.util.Stack	✗	✓	✗	See the 1-N Mapping Guide

Java Type	EAGER?	Proxy?	PK?	Comments
java.util.TreeMap	✗	✓	✗	The comparator is specifiable via the metadata extension <i>comparator-name</i> (see below). See the 1-N Mapping Guide
java.util.TreeSet	✗	✓	✗	The comparator is specifiable via the metadata extension <i>comparator-name</i> (see below). See the 1-N Mapping Guide
java.util.Vector	✗	✓	✗	See the 1-N Mapping Guide
com.google.common.collect.Multiset	✗	✓	✗	Requires datanucleus-guava plugin. See the 1-N Collections Guide

Collection Comparators



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

```
@OneToMany
@Extension(key="comparator-name", value="mydomain.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

By default an Enum is persisted as either a String form (the name), or as an integer form (the ordinal). You control which form by specifying the **@Enumerated** annotation (or equivalent XML).

Java Type	EAGER?	Proxy?	PK?	Comments
java.lang.Enum	✓	✗	✓	Persisted as String (name) or int (ordinal). Specified via @Enumerated annotation or equivalent XML.

Enum custom values



A DataNucleus extension to this is where you have an Enum that defines its own "value"s for the

different enum options.



Applicable to RDBMS, MongoDB, Cassandra, Neo4j, HBase, Excel, ODF and JSON currently.

```
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 `@Enumerated` as `ORDINAL` it would persist 0, 1, 2, 3 being the ordinal values. If you define the metadata as

```
@Extension(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.

Enum CHECK constraints



Supported for RDBMS datastores.

If you want to constraint the column where the Enum is stored to only have the values for that enum you can put a CHECK constraint on the column contents in the datastore. You specify it like this

```
@Extension(vendorName="datanucleus", key="enum-check-constraint", value="true")
MyColour colour;
```

This results in a column defined like

```
MY_COL VARCHAR(10) CHECK (MY_COL IN ('RED', 'GREEN', 'BLUE', 'YELLOW')),
```



This is the recommended way of constraining enum values in the datastore since it uses ANSI SQL, and it is a better more portable solution than using such as PostgreSQL enum type.

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. This plugin supports types from all of the most used geospatial libraries, see below. The implementation of many of these spatial types follows the [OGC Simple Feature specification](#), but adds further types where the datastores support them.

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 mapping scenario!



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!



We do not currently support persisting to the PostGIS "geography" type, only the (most used) "geometry" type.

java.awt Java types

The JRE contains very limited support for some geometric types, largely under the *java.awt* package.

Java Type	EAGER?	Proxy?	PK?	Comments
java.awt.Point	✓	✓	✗	Requires datanucleus-geospatial plugin. Persisted as (int, int) on RDBMS, or as String elsewhere.
java.awt.Rectangle	✓	✓	✗	Requires datanucleus-geospatial plugin. Persisted as (int, int, int, int) on RDBMS, or as String elsewhere.
java.awt.Polygon	✓	✓	✗	Requires datanucleus-geospatial plugin. Persisted as (int[], int[], int) on RDBMS, or as String elsewhere.
java.awt.geom.Line2D	✓	✓	✗	Requires datanucleus-geospatial plugin. Persisted as (double, double, double, double) or (float, float, float, float) on RDBMS, or as String elsewhere.
java.awt.geom.Point2D	✓	✓	✗	Requires datanucleus-geospatial plugin. Persisted as (double, double) or (float, float) on RDBMS, or as String elsewhere.
java.awt.geom.Rectangle2D	✓	✓	✗	Requires datanucleus-geospatial plugin. Persisted as (double, double, double, double) or (float, float, float, float) on RDBMS, or as String elsewhere.
java.awt.geom.Arc2D	✓	✓	✗	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.
java.awt.geom.CubicCurve2D	✓	✓	✗	Requires datanucleus-geospatial plugin. Persisted as (double, double, double, double, double, double, double, double) or (float, float, float, float, float, float, float, float) on RDBMS, or as String elsewhere.
java.awt.geom.Ellipse2D	✓	✓	✗	Requires datanucleus-geospatial plugin. Persisted as (double, double, double, double) or (float, float, float, float) on RDBMS, or as String elsewhere.

Java Type	EAGER?	Proxy?	PK?	Comments
java.awt.geom.QuadCurve2D	✓	✓	✗	Requires datanucleus-geospatial plugin. Persisted as (double, double, double, double, double, double) or (float, float, float, float, float, float) on RDBMS, or as String elsewhere.
java.awt.geom.RoundRectangle2D	✓	✓	✗	Requires datanucleus-geospatial plugin. Persisted as (double, double, double, double, double, double) or (float, float, float, float, float, float) on RDBMS, or as String elsewhere.

JTS Topology Suite Java types

The [JTS Topology Suite](#) is a Java library for creating and manipulating vector geometry.

Java Type	DFG?	Proxy?	PK?	Comments
com.vividsolutions.jts.geom.Geometry	✓	✗	✗	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).
com.vividsolutions.jts.geom.GeometryCollection	✓	✗	✗	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).
com.vividsolutions.jts.geom.LineString	✓	✗	✗	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).
com.vividsolutions.jts.geom.LineString	✓	✗	✗	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).
com.vividsolutions.jts.geom.MultiLineString	✓	✗	✗	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).

Java Type	DFG?	Proxy ?	PK?	Comments
com.vividsolutions.jts.geom.MultiPoint	✓	✗	✗	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).
com.vividsolutions.jts.geom.MultiPolygon	✓	✗	✗	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).
com.vividsolutions.jts.geom.Point	✓	✗	✗	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).
com.vividsolutions.jts.geom.Polygon	✓	✗	✗	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).

PostGIS Java types

[PostGIS](#) provides a series of geometric types for use in Java applications

Java Type	DFG?	Proxy ?	PK?	Comments
org.postgis.Geometry	✓	✗	✗	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).
org.postgis.GeometryCollection	✓	✗	✗	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).
org.postgis.LinearRing	✓	✗	✗	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).

Java Type	DFG?	Proxy ?	PK?	Comments
org.postgis.LineString	✓	✗	✗	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).
org.postgis.MultiLineString	✓	✗	✗	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).
org.postgis.MultiPoint	✓	✗	✗	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).
org.postgis.MultiPolygon	✓	✗	✗	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).
org.postgis.Point	✓	✗	✗	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).
org.postgis.Polygon	✓	✗	✗	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).
org.postgis.PGbox2d	✓	✗	✗	Requires datanucleus-geospatial plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on PostGIS(geometry).
org.postgis.PGbox3d	✓	✗	✗	Requires datanucleus-geospatial plugin. Dirty check limited to immutable mode (must reassign field to owner if you change it). Only on PostGIS(geometry).

Oracle JGeometry Java type

Oracle provides its own geometry type for use in Oracle databases.

Java Type	DFG?	Proxy ?	PK?	Comments
oracle.spatial.geometry.JGeometry	✓	✗	✗	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)

Geospatial metadata extensions



datanucleus-geospatial has defined some metadata extensions that can be used to give additional information about the geometry types in use. You use them inside a field and the values are only used for that field specifically, or for an entity as a whole.


```

<entity-mappings>
  <package>mydomain.jtsgeometry</package>

  <entity class="mydomain.jtsgeometry.SampleGeometry">
    <extension key="spatial-dimension" value="2"/>
    <extension key="spatial-srid" value="4326"/>
    <attributes>
      <id name="id"/>
      <basic name="name"/>
      <basic name="geom">
        <extension key="mapping" value="no-userdata"/> [2]
      </basic>
    </attributes>
  </entity>

  <entity class="mydomain.jtsgeometry.SampleGeometryCollectionM">
    <extension key="spatial-dimension" value="2"/>
    <extension key="spatial-srid" value="4326"/>
    <extension key="postgis-hasMeasure" value="true"/> [3]
    <attributes>
      <id name="id"/>
      <basic name="name"/>
      <basic name="geom"/>
    </attributes>
  </entity>

  <entity class="mydomain.jtsgeometry.SampleGeometryCollection3D">
    <extension key="spatial-dimension" value="3"/>
    <extension key="spatial-srid" value="-1"/>
    <attributes>
      <id name="id"/>
      <basic name="name"/>
      <basic name="geom"/>
    </attributes>
  </entity>
</entity-mappings>

```

- [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-userdata" 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.

Java Type	EAGER?	Proxy?	PK?	Comments
java.lang.Object	✗	✗	✗	Either persisted serialised , or represents multiple possible types
java.util.Currency	✓	✗	✓	Persisted as String.
java.util.Locale	✓	✗	✓	Persisted as String.
java.util.UUID	✓	✗	✓	Persisted as String, or alternatively as native <i>uuid</i> on PostgreSQL/H2/HSQLDB when specifying <code>sql-type="uuid"</code> .
java.util.Optional<type>;	✓	✗	✗	Persisted as the type of the generic type that optional represents.
java.awt.Color	✓	✗	✗	Persisted as String or as (Integer,Integer,Integer,Integer) storing red,green,blue,alpha respectively.
java.awt.image.BufferedImage	✗	✗	✗	Persisted as serialised .
java.net.URI	✓	✗	✓	Persisted as String.
java.net.URL	✓	✗	✓	Persisted as String.
java.io.Serializable	✗	✗	✗	Persisted as serialised .
java.io.File	✗	✗	✗	Only for RDBMS, persisted to LONGVARBINARY, and retrieved as streamable so as not to adversely affect memory utilisation, hence suitable for large files.

Arrays

The vast majority of the secondary 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

Java Type	EAGER?	Proxy?	PK?	Comments
boolean[]	✗	✗	✗	See the Arrays Guide
byte[]	✗	✗	✗	See the Arrays Guide
char[]	✗	✗	✗	See the Arrays Guide
double[]	✗	✗	✗	See the Arrays Guide
float[]	✗	✗	✗	See the Arrays Guide

Java Type	EAGER?	Proxy?	PK?	Comments
int[]	✗	✗	✗	See the Arrays Guide
long[]	✗	✗	✗	See the Arrays Guide
short[]	✗	✗	✗	See the Arrays Guide
java.lang.Boolean[]	✗	✗	✗	See the Arrays Guide
java.lang.Byte[]	✗	✗	✗	See the Arrays Guide
java.lang.Character[]	✗	✗	✗	See the Arrays Guide
java.lang.Double[]	✗	✗	✗	See the Arrays Guide
java.lang.Float[]	✗	✗	✗	See the Arrays Guide
java.lang.Integer[]	✗	✗	✗	See the Arrays Guide
java.lang.Long[]	✗	✗	✗	See the Arrays Guide
java.lang.Short[]	✗	✗	✗	See the Arrays Guide
java.lang.String[]	✗	✗	✗	See the Arrays Guide
java.util.Date[]	✗	✗	✗	See the Arrays Guide
java.math.BigDecimal[]	✗	✗	✗	See the Arrays Guide
java.math.BigInteger[]	✗	✗	✗	See the Arrays Guide
java.lang.Enum[]	✗	✗	✗	See the Arrays Guide
java.util.Locale[]	✗	✗	✗	See the Arrays Guide
Entity[]	✗	✗	✗	See the Arrays Guide

Generic Type Variables

JPA does not explicitly require support for generic type variables. DataNucleus provides support for many situations with generic type variables.

The first example that is supported is where you have an abstract base class with a generic `TypeVariable` and then you specify the type in the (concrete) subclass(es).

```

@MappedSuperclass
public abstract class Base<T>
{
    private T id;
}

@Entity
public class Sub1 extends Base<Long>
{
    ...
}

@Entity
public class Sub2 extends Base<Integer>
{
    ...
}

```

Similarly you use `TypeVariables` to form relations, like this

```

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

@Entity
public class Document extends Ownable<Person>
{
    ...
}

```

Similarly, if you use a type argument in a generic declaration for a field, like this

```

public class Owner
{
    private List<? extends Element> elements;
}

public class Element
{
    ...
}

```

Clearly there are many combinations of where generics and `TypeVariables` can be used, so let us know if your generics usage isn't supported.

JPA Attribute Converters

JPA provides an API for conversion of an attribute of an Entity to its datastore value. You can define a "converter" that will convert to the datastore value and back from it, implementing this interface.

```
public interface AttributeConverter<X,Y>
{
    public Y convertToDatabaseColumn (X attributeObject);

    public X convertToEntityAttribute (Y dbData);
}
```

so if we have a simple converter to allow us to persist fields of type URL in a String form in the datastore, like this

```
public class URLStringConverter implements AttributeConverter<URL, String>
{
    public URL convertToEntityAttribute(String str)
    {
        if (str == null)
        {
            return null;
        }

        URL url = null;
        try
        {
            url = new java.net.URL(str.trim());
        }
        catch (MalformedURLException mue)
        {
            throw new IllegalStateException("Error converting the URL", mue);
        }
        return url;
    }

    public String convertToDatabaseColumn(URL url)
    {
        return url != null ? url.toString() : null;
    }
}
```

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

```

@Entity
public class MyClass
{
    @Id
    long id;

    @Basic
    @Convert(converter=URLStringConverter.class)
    URL url;

    ...
}

```

or using XML metadata

```

<basic name="url">
    <convert converter="mydomain.URLStringConverter"/>
</basic>

```

Note that in strict JPA you have to mark all converters with the `@Converter` annotation. In DataNucleus if you specify the converter class name in the `@Convert` then we know its a converter so don't really see why we need a user to annotate the converter too. We only require annotation as `@Converter` if you want the converter to always be applied to fields of a particular type. i.e if you want all URL fields to be persisted using the above converter (without needing to put `@Convert` on each field of that type) then you would add the annotation

```

@Converter(autoApply=true)
public class URLStringConverter implements AttributeConverter<URL, String>
{
    ...
}

```

Note that if you have some java type with a `@Converter` registered to *autoApply*, you can turn it off on a field-by-field basis with

```

@Convert(disableConversion=true)
URL url;

```

or using XML metadata

```

<basic name="url">
    <convert disable-conversion="true"/>
</basic>

```

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.

```
@OneToMany
@Convert(attributeName="key", converter=URLStringConverter.class)
Map<URL, OtherEntity> myMap;
```

So we specify the `attributeName` to be **key**, and to use it on the value we would set it to **value**.

If using XML metadata

```
<one-to-many name="myMap">
  <map-key-convert converter="mydomain.URLStringConverter"/>
</one-to-many>
```



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



The `AttributeConverter` objects shown here are **stateless**. `DataNucleus` allows for stateful `AttributeConverter` objects, with the state being CDI injectable, but you must be in a CDI environment for this to work. To provide CDI support for JPA, you should specify the persistence property `javax.persistence.bean.manager` to be a CDI `BeanManager` object.

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. If you have your type and want to just persist it into a single column then you should do as follows

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

@Entity
public class MyClass
{
    MyCollectionType myField;

    ...
}
```

We now define a simple converter 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 convertToEntityAttribute(String str)
    {
        if (str == null)
        {
            return null;
        }

        ...
        return myType;
    }

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

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

```
@Entity
public class MyClass
{
    @Convert(converter=MyCollectionTypeStringConverter.class)
    MyCollectionType myField;

    ...
}
```



If you want your extension of Collection/Map to be managed as a second class type then you will need to provide a *wrapper* class for it. Please refer to the [java_type](#)



Column Adapters



Supported for RDBMS.

By default, when inserting/updating into a column into an RDBMS datastore, the SQL will have a *?* and the value replaced into it. We allow the use of adapter "functions" so that the inserted value can be modified during the insert/update. Like this


```
@Extension(key="insert-function", value="TRIM(?)")
@Extension(key="update-function", value="TRIM(?)")
String myStringField;
```

So when this field of this class is persisted the SQL generated will include `TRIM(?)` rather than `?`, and any leading/trailing whitespace will be removed.

Similarly on retrieval, we also allow the equivalent.

```
@Extension(key="select-function", value="UPPER(?)")
String myStringField;
```

The `?` is replaced by the column name. So the stored datastore value will be converted to UPPERCASE before being set in the Java object retrieved.

You could use these *column adapters* to do things like encrypt/decrypt the value of a field when storing to/retrieving from the database, for example.

Value Generation

Fields of a class can either have the values set by you the user, or you can set DataNucleus to generate them for you. This is of particular importance with identity fields where you want unique identities. You can use this value generation process with the identity field(s) in JPA. There are many different "strategies" for generating values, as defined by the JPA specification. Some strategies are specific to a particular datastore, and some are generic. You should choose the strategy that best suits your target datastore. The available strategies are :-

- [AUTO](#) - this is the default and allows DataNucleus to choose the most suitable for the datastore
- [SEQUENCE](#) - this uses a datastore sequence (if supported by the datastore)
- [IDENTITY](#) - these use autoincrement/identity/serial features in the datastore (if supported by the datastore)
- [TABLE](#) - this is datastore neutral and increments a sequence value using a table.
- [UUID](#) - datastore neutral, created in the JRE. Provided by DataNucleus
- [Custom generators](#) - these are beyond the scope of the JPA spec but provided by DataNucleus

See also:-

- [JPA MetaData reference for <generated-value>](#)
- [JPA Annotation reference for @GeneratedValue](#)



the JPA spec only requires the ability to generate values for identity fields. DataNucleus allows you to do it for any field. Please bear this in mind when considering portability



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*

ValueGeneration Strategy AUTO

With this strategy DataNucleus will choose the most appropriate strategy for the datastore being used. If you define the field as String-based then it will choose [uuid-hex](#). Otherwise the field is numeric in which case it chooses [identity](#) if supported, otherwise [sequence](#) if supported, otherwise [table](#) if supported otherwise throws an exception.

On RDBMS you can get the behaviour used up until DN v3.0 by specifying the persistence property `datanucleus.rdbms.useLegacyNativeValueStrategy` as *true*.

For a class using **application identity** you need to set the *value-strategy* attribute on the primary key field. You can configure the Meta-Data for the class something like this

```

<entity class="MyClass">
  <attributes>
    <id name="myId">
      <generated-value strategy="AUTO"/>
    </id>
  </attributes>
</entity>

```

or using annotations

```

@Entity
public class MyClass
{
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long myId;
    ...
}

```

To configure a class to use this generation using **datastore identity** you need to look at the `@DatastoreId` extension annotation or the XML `<datastore-id>` tag



Which generation strategy is used internally will be JPA provider dependent. If you want to be portable and independent of a JPA providers internals you should likely avoid use of `AUTO`.

ValueGeneration Strategy SEQUENCE



Applicable to RDBMS (Oracle, PostgreSQL, SAPDB, DB2, Firebird, HSQLDB, H2, Derby, SQLServer, NuoDB).

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`. To configure a class to use this strategy using **application identity** you would add the following to the class' Meta-Data

```

<sequence-generator name="SEQ1" sequence-name="MY_SEQ" initial-value="5" allocation-size="10"/>
<entity class="MyClass">
  <attributes>
    <id name="myId">
      <generated-value strategy="SEQUENCE" generator="SEQ1"/>
    </id>
  </attributes>
</entity>

```

or using annotations

```
@Entity
@SequenceGenerator(name="SEQ1", sequenceName="MY_SEQ", initialValue=5, allocationSize
=10)
public class MyClass
{
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="SEQ1")
    private long myId;
    ...
}
```

If the sequence does not yet exist in the database at the time DataNucleus needs a new unique identifier, a new sequence is created in the database based on the JPA Meta-Data configuration.

Extension properties for configuring sequences can be set in the JPA Meta-Data (via `@Extension` or `<extension>`), see the available properties below. Unsupported properties by a database are silently ignored by DataNucleus.

Property	Description	Required
key-database-cache-size	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	No

To configure a class to use this generation using **datastore identity** you need to look at the `@DatastoreId` extension annotation or the XML `<datastore-id>` tag.

This value generator will generate values unique across different JVMs

Values generated using this generator are available in `@PrePersist`.

See also:-

- [JPA MetaData reference for <sequence-generator>](#)
- [JPA Annotation reference for @SequenceGenerator](#)

ValueGeneration Strategy IDENTITY



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

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.

Any field using this strategy will NOT be present in any INSERT statement, and will be set in the datastore as a result.



This generation strategy should only be used if there is a single "root" table for the inheritance tree. If you have more than 1 root table (e.g using subclass-table inheritance) then you should choose a different generation strategy

For a class using **application identity** you need to set the *value-strategy* attribute on the primary key field. You can configure the Meta-Data for the class something like this

```
<entity class="MyClass">
  <attributes>
    <id name="myId">
      <generated-value strategy="IDENTITY"/>
    </id>
  </attributes>
</entity>
```

or using annotations

```
@Entity
public class MyClass
{
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long myId;
    ...
}
```

Please be aware that if you have an inheritance tree with the base class defined as using "identity" then the column definition for the PK of the base table will be defined as "AUTO_INCREMENT" or "IDENTITY" or "SERIAL" (dependent on the RDBMS) and all subtables will NOT have this identifier added to their PK column definitions. This is because the identities are assigned in the base table (since all objects will have an entry in the base table).

Please note that if using optimistic transactions, this strategy will mean that the value is only set when the object is actually persisted (i.e at flush() or commit())

To configure a class to use this generation using **datastore identity** you need to look at the `@DatastoreId` extension annotation or the XML `<datastore-id>` tag

This value generator will generate values unique across different JVMs.

Values generated using this generator are NOT available in `@PrePersist`, being generated at persist only.

ValueGeneration Strategy TABLE



Applies to RDBMS, ODF, Excel, OOXML, HBase, Cassandra, MongoDB, Neo4j.

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 method require a sequence table in the database and creates one if doesn't exist.

To configure an **application identity** class to use this generation method you simply add this to the class' Meta-Data. If your class is in an inheritance tree you should define this for the base class only.

```
<entity class="MyClass">
  <table-generator name="myGenerator" table="TABLE_VALUE_GEN" pkColumnName="GEN_KEY"
valueColumnName="GEN_VALUE" pkColumnValue="MyClass"/>
  <attributes>
    <id name="myId">
      <generated-value strategy="TABLE"/>
    </id>
  </attributes>
</entity>
```

or using annotations

```
@Entity
@TableGenerator(name="myGenerator", table="TABLE_VALUE_GEN", pkColumnName="GEN_KEY",
valueColumnName="GEN_VALUE", pkColumnValue="MyClass")
public class MyClass
{
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE, generator="myGenerator")
    private long myId;
    ...
}
```

This will create a table in the datastore called `TABLE_VALUE_GEN` with columns `GEN_KEY`, `GEN_VALUE` with the key for the row for this class being "MyClass".

Extension properties for configuring sequences can be set in the JPA Meta-Data (via `@Extension` or `<extension>`), see the available properties below. Unsupported properties by a database are silently ignored by DataNucleus.

Property	Description	Required
sequence-table-basis	Whether to define uniqueness on the base class name or the base table name. Since there is no "base table name" when the root class has "subclass-table" this should be set to "class" when the root class has "subclass-table" inheritance	No. Defaults to <i>class</i> , but the other option is <i>table</i>
table-name	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).	No.
column-name	Name of the column we are generating the value for (used when we have no previous sequence value and want a start point).	No.

To configure a class to use this generation using **datastore identity** you need to look at the `@DatastoreId` extension annotation or the XML `<datastore-id>` tag

This value generator will generate values unique across different JVMs

Values generated using this generator are available in `@PrePersist`.

See also:-

- [JPA MetaData reference for <table-generator>](#)
- [JPA Annotation reference for @TableGenerator](#)

ValueGeneration Strategy "UUID"



Applicable to all datastores.

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.

```

<entity class="MyClass">
  <attributes>
    <id name="myId">
      <generated-value strategy="UUID"/>
    </id>
  </attributes>
</entity>

```

or using annotations

```

public class MyClass
{
    @GeneratedValue(strategy=GenerationType.UUID)
    String myId;
}

```

This value generator will generate values unique across different JVMs.



To use this strategy via annotations you need to have the DataNucleus `javax.persistence.jar` present in the CLASSPATH.

ValueGeneration Strategy "Custom"



JPA only provides a very restricted set of value generators. DataNucleus provides various others internally. To access these you need to use a custom strategy as follows

```

<entity class="MyClass">
  <attributes>
    <id name="myId">
      <generated-value strategy="uuid"/>
    </id>
  </attributes>
</entity>

```

or using annotations


```
import org.datanucleus.api.jpa.annotations.ValueGenerator;

@Entity
public class MyClass
{
    @Id
    @ValueGenerator(strategy="uuid")
    private String myId;
    ...
}
```

This will generate java UUID Strings in the "myId" field. You can also set the "strategy" to "auid", "uuid-string", "uuid-hex", "uuid", "uuid-object", "timestamp" and "timestamp_value".

Values generated using these generators are available in `@PrePersist`.

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), or less likely as an entry in a join table. 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.



You cannot have a 1-1 relation to a long/int field! JPA is for use with object-oriented systems, not flat data.

Unidirectional (ForeignKey)

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

```
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

```

@Entity
public class Account
{
    ...

    @OneToOne
    @JoinColumn(name="USER_ID")
    User user;
}

@Entity
public class User
{
    ...
}

```

or using XML metadata

```

<entity-mappings>
  <entity class="User">
    <table name="USER"/>
    <attributes>
      <id name="id">
        <column name="USER_ID"/>
      </id>
      ...
    </entity>

  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      <id name="id">
        <column name="ACCOUNT_ID"/>
      </id>
      ...
      <one-to-one name="user">
        <join-column name="USER_ID"/>
      </one-to-one>
    </attributes>
  </entity>
</entity-mappings>

```

This will create 2 tables in the database, **USER** (for class **User**), and **ACCOUNT** (for class **Account**, with a column **USER_ID**), as shown below.



Account has the object reference to **User** (and so is the "owner" of the relation) and so its table holds the foreign-key



If you call *EntityManager.remove()* on the end of a 1-1 unidirectional relation without the relation and that object is related to another object, an exception will typically be thrown (assuming the datastore supports foreign keys). To delete this record you should remove the other objects association first.

Unidirectional (JoinTable)



Supported for RDBMS. Other datastores simply ignore the join table and store the relation in a column in the table of the object with the field.

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 knows about the **User** class, but not vice-versa, and are using a join table. A particular user could be related to several accounts. If you define the annotations for these classes as follows

```

@Entity
public class Account
{
    @OneToOne
    @JoinTable(name="ACCOUNT_USER")
    User user;

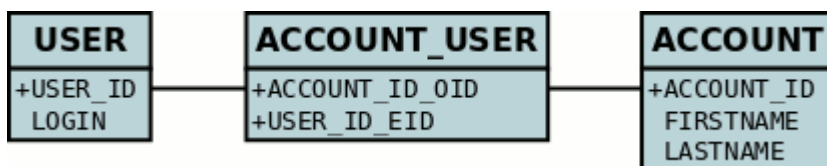
    ....
}
  
```

or using XML metadata

```
<entity-mappings>
  <entity class="User">
    <table name="USER"/>
    <attributes>
      <id name="id">
        <column name="USER_ID"/>
      </id>
      ...
    </attributes>
  </entity>

  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      <id name="id">
        <column name="ACCOUNT_ID"/>
      </id>
      ...
      <one-to-one name="user">
        <join-table name="ACCOUNT_USER"/>
      </one-to-one>
    </attributes>
  </entity>
</entity-mappings>
```

This will create 3 tables in the database, **USER** (for class **User**), **ACCOUNT** (for class **Account**), and a join table **ACCOUNT_USER**, as shown below.



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

Bidirectional (ForeignKey)

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

```

@Entity
public class Account
{
    ...

    @OneToOne
    @JoinColumn(name="USER_ID")
    User user;
}

@Entity
public class User
{
    ...

    @OneToOne(mappedBy="user")
    Account account;

    ...
}

```

or using XML metadata

```

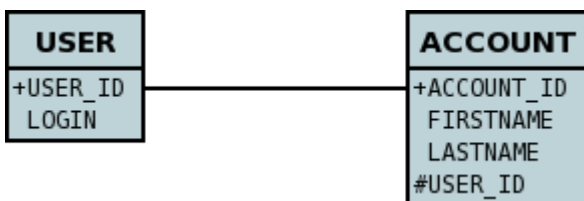
<entity-mappings>
  <entity class="User">
    <table name="USER"/>
    <attributes>
      <id name="id">
        <column name="USER_ID"/>
      </id>
      ...
      <one-to-one name="account" mapped-by="user"/>
    </attributes>
  </entity>

  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      <id name="id">
        <column name="ACCOUNT_ID"/>
      </id>
      ...
      <one-to-one name="user">
        <join-column name="USER_ID"/>
      </one-to-one>
    </attributes>
  </entity>
</entity-mappings>

```

The difference is that we added *mapped-by* to the field of **User** making it bidirectional (and putting the FK at the other side for RDBMS)

This will create 2 tables in the database, **USER** (for class **User**), and **ACCOUNT** (for class **Account**). For RDBMS it includes a **USER_ID** column in the **ACCOUNT** table, like this



For other types of datastore it will have a *USER_ID* column in the **ACCOUNT** table and a **ACCOUNT** column in the **USER** table.



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.

Bidirectional (JoinTable)



DataNucleus does not support a 1-1 BIDIRECTIONAL relation using a join table. It is not a use-case that is very common and is not part of the JPA spec. You could look at doing that as a unidirectional relation.

1-N Relations

You have a 1-N (one to many) when you have one object of a class that has a Collection of objects of another class. **Please note that Collections allow duplicates, and so the persistence process reflects this with the choice of primary keys.**

There are two 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<Entity> - Unidirectional using join table](#)
- [Collection<Entity> - Unidirectional using foreign-key](#)
- [Collection<Entity> - Bidirectional using join table](#)
- [Collection<Entity> - Bidirectional using foreign-key](#)
- [Collection<Entity> using shared join table \(DataNucleus Extension\)](#)
- [Collection<Entity> using shared foreign key \(DataNucleus Extension\)](#)
- [Collection<Simple> using join table](#)
- [Collection<Simple> using AttributeConverter into single column](#)
- [Map<Simple, Entity> using join table](#)
- [Map<Simple,Entity> - Unidirectional using foreign-key \(key stored in the value class\)](#)
- [Map<Simple,Entity> - Bidirectional using foreign-key \(key stored in the value class\)](#)
- [Map<Entity, Entity> using join table](#)
- [Map<Simple, Simple> using join table](#)
- [Map<Simple, Simple> using AttributeConverter into single column](#)
- [Map<Entity, Simple> using join table](#)



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 in the *equals*/*hashCode* methods.

Ordering of elements

You can only retain the order in a Collection if you instantiate the Collection as an implementation that supports retaining the order. If the order of elements is important to you, use a List, or instantiate using something like a TreeSet where the order is handled.

In the case of the relation field being ordered, you define the relation just like you would for a Collection but then define whether you want the relation to be either *ordered* or *indexed*.

In the case of *ordered* you would add the following to the metadata of the field

```
@OrderBy("city")
```

or using XML

```
<order-by>city</order-by>
```

This means that when the elements of the List/SortedSet/etc are retrieved then they will be ordered according to the *city* field of the element.

If instead you want an *indexed* list then the elements will have an index stored against them, hence preserving the order in which they were in the original List. This adds a surrogate column to either the table of the element (when using *foreign key*) or to the join table.

```
@OrderColumn("ORDERING")
```

or using XML

```
<order-column>ORDERING</order-column>
```



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

```
@OneToMany
@Extension(key="comparator-name", value="mydomain.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).

Collection<Entity> Unidirectional (JoinTable)



Supported for RDBMS. Other datastores simply ignore the join table and store the relation in a column in the table of the object with the field.

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

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

    ...
}

public class Address
{
    ...
}
```

If you define the annotations of the classes like this

```
public class Account
{
    ...

    @OneToMany
    @JoinTable(name="ACCOUNT_ADDRESSES",
        joinColumns={@JoinColumn(name="ACCOUNT_ID_OID")},
        inverseJoinColumns={@JoinColumn(name="ADDRESS_ID_EID")})
    Collection<Address> addresses
}

public class Address
{
    ...
}
```

or using XML

```

<entity-mappings>
  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      ...
      <one-to-many name="addresses">
        <join-table name="ACCOUNT_ADDRESSES">
          <join-column name="ACCOUNT_ID_OID"/>
          <inverse-join-column name="ADDRESS_ID_EID"/>
        </join-table>
      </one-to-many>
    </attributes>
  </entity>

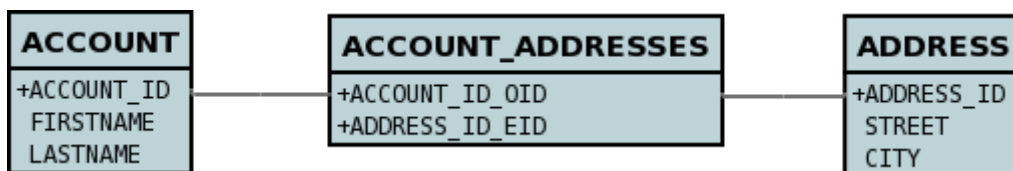
  <entity class="Address">
    <table name="ADDRESS"/>
    ...
  </entity>
</entity-mappings>

```



The crucial part is the `join-table` element on the field element (`@JoinTable` annotation) - this signals to JPA to use a join table.

This will create 3 tables in the database, one for **Address**, one for **Account**, and a join table, as shown below.



The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the `table` element below the `class` element
- To specify the names of the columns where the fields of a class are stored, specify the `column` attribute on the `basic` element.
- To specify the name of the join table, specify the `join-table` element below the `one-to-many` element with the collection.
- To specify the names of the join table columns, use the `join-column` and `inverse-join-column` elements below the `join-table` element.
- If the field type is Set then the join table will be given a primary key (since a Set cannot have duplicates), whereas for other Collection types it will not have a primary key (since duplicates are allowed).

Collection<Entity> Unidirectional (ForeignKey)

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

```
public class Account
{
    ...

    @OneToMany
    @JoinColumn(name="ACCOUNT_ID")
    Collection<Address> addresses
}

public class Address
{
    ...
}
```

or using XML metadata

```
<entity-mappings>
  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      ...
      <one-to-many name="addresses">
        <join-column name="ACCOUNT_ID"/>
      </one-to-many>
    </attributes>
  </entity>

  <entity class="Address">
    <table name="ADDRESS"/>
    ...
  </entity>
</entity-mappings>
```



you MUST specify the join-column here otherwise it defaults to a join table with JPA!

There will be 2 tables, one for **Address**, and one for **Account**. If you wish to specify the names of the column(s) used in the schema for the foreign key in the **Address** table you should use the *join-column* element within the field of the collection.



In terms of operation within your classes of assigning the objects in the relationship. You have to take your **Account** object and add the **Address** to the **Account** collection field since the **Address** knows nothing about the **Account**.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the *table* element below the **class** element
- To specify the names of the columns where the fields of a class are stored, specify the *column* attribute on the **basic** element.



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<Entity> Bidirectional (JoinTable)



Supported for RDBMS. Other datastores simply ignore the join table and store the relation in a column in the table of the object with the field.

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

```

public class Account
{
    Collection<Address> addresses;

    ...
}

public class Address
{
    Account account;

    ...
}
  
```

If you define the annotations for these classes as follows

```

public class Account
{
    ...

    @OneToMany(mappedBy="account")
    @JoinTable(name="ACCOUNT_ADDRESSES",
        joinColumns={@JoinColumn(name="ACCOUNT_ID_OID")},
        inverseJoinColumns={@JoinColumn(name="ADDRESS_ID_EID")})
    Collection<Address> addresses;
}

public class Address
{
    ...

    @ManyToOne
    Account account;
}

```

or alternatively using XML

```

<entity-mappings>
  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      ...
      <one-to-many name="addresses" mapped-by="account">
        <join-table name="ACCOUNT_ADDRESSES">
          <join-column name="ACCOUNT_ID_OID"/>
          <inverse-join-column name="ADDRESS_ID_EID"/>
        </join-table>
      </one-to-many>
    </attributes>
  </entity>

  <entity class="Address">
    <table name="ADDRESS"/>
    <attributes>
      ...
      <many-to-one name="account"/>
    </attributes>
  </entity>
</entity-mappings>

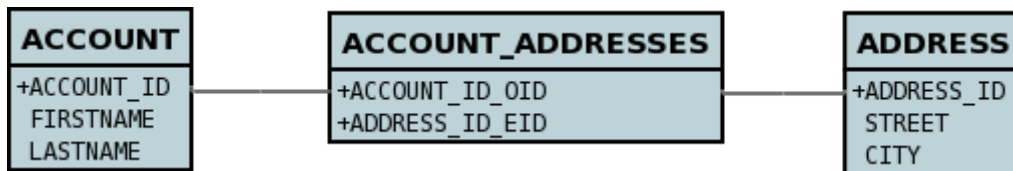
```



The crucial part is the `join-table` element on the field element (or `@JoinTable` annotation) - this signals to JPA to use a join table.

This will create 3 tables in the database, one for **Address**, one for **Account**, and a join table, as

shown below.



The join table is used to link the 2 classes via foreign keys to their primary key. This is useful where you want to retain the independence of one class from the other class.

If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the *table* element below the `<class>` element
- To specify the names of the columns where the fields of a class are stored, specify the *column* attribute on the `<basic>` element.
- To specify the name of the join table, specify the *join-table* element below the `<one-to-many>` element with the collection.
- To specify the names of the join table columns, use the `<join-column>` and `<inverse-join-column>` elements below the `<join-table>` element.
- If the field type is a Set then the join table will be given a primary key (since a Set cannot have duplicates), whereas for other Collection types no primary key is assigned.
- 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.

Collection<Entity> Bidirectional (ForeignKey)

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


```

public class Account
{
    ...

    @OneToMany(mappedBy="account")
    @JoinColumn(name="ACCOUNT_ID")
    Collection<Address> addresses
}

public class Address
{
    ...

    @ManyToOne
    Account account;
}

```

or alternatively using XML metadata

```

<entity-mappings>
  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      ...
      <one-to-many name="addresses" mapped-by="account">
        <join-column name="ACCOUNT_ID"/>
      </one-to-many>
    </attributes>
  </entity>

  <entity class="Address">
    <table name="ADDRESS"/>
    <attributes>
      ...
      <many-to-one name="account"/>
    </attributes>
  </entity>
</entity-mappings>

```



The crucial part is the *mapped-by* attribute of the field on the "1" side of the relationship. This tells the JPA implementation to look for a field called *account* on the **Address** class.

This will create 2 tables in the database, one for **Address** (including an **ACCOUNT_ID** to link to the **ACCOUNT** table), and one for **Account**. Notice the subtle difference to this set-up to that of the **Join Table** relationship earlier.



If you wish to fully define the schema table and column names etc, follow these tips

- To specify the name of the table where a class is stored, specify the *table* element below the **class*element*
- To specify the names of the columns where the fields of a class are stored, specify the *column* attribute on the **basic** element.
- When forming the relation please make sure that **you set the relation at BOTH sides** since DataNucleus would have no way of knowing which end is correct if you only set one end.



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<Entity> (Shared JoinTable)



Supported for RDBMS.



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 [shown 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

```

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

```

import org.datanucleus.api.jpa.annotations.SharedRelation;

public class Account
{
    @OneToMany
    @JoinTable(name="ACCOUNT_ADDRESSES",
        joinColumns={@JoinColumn(name="ACCOUNT_ID_OID")},
        inverseJoinColumns={@JoinColumn(name="ADDRESS_ID_EID")})
    @SharedRelation(column="ADDRESS_TYPE", value="work")
    Collection<Address> workAddresses;

    @OneToMany
    @JoinTable(name="ACCOUNT_ADDRESSES",
        joinColumns={@JoinColumn(name="ACCOUNT_ID_OID")},
        inverseJoinColumns={@JoinColumn(name="ADDRESS_ID_EID")})
    @SharedRelation(column="ADDRESS_TYPE", value="home")
    Collection<Address> homeAddresses;

    ...
}

```

or using XML metadata

```

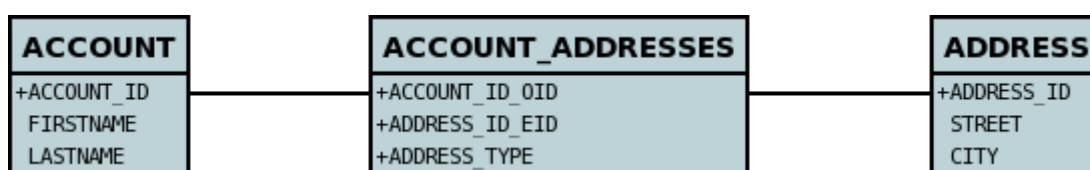
<entity-mappings>
  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      ...
      <one-to-many name="workAddresses">
        <join-table name="ACCOUNT_ADDRESSES">
          <join-column name="ACCOUNT_ID_OID"/>
          <inverse-join-column name="ADDRESS_ID_EID"/>
        </join-table>
        <extension key="relation-discriminator-column" value="ADDRESS_TYPE"/>
        <extension key="relation-discriminator-value" value="work"/>
        <!--extension key="relation-discriminator-pk" value="true"/-->
      </one-to-many>
      <one-to-many name="homeAddresses">
        <join-table name="ACCOUNT_ADDRESSES">
          <join-column name="ACCOUNT_ID_OID"/>
          <inverse-join-column name="ADDRESS_ID_EID"/>
        </join-table>
        <extension key="relation-discriminator-column" value="ADDRESS_TYPE"/>
        <extension key="relation-discriminator-value" value="home"/>
        <!--extension key="relation-discriminator-pk" value="true"/-->
      </one-to-many>
    </attributes>
  </entity>

  <entity class="Address">
    <table name="ADDRESS"/>
    ...
  </entity>
</entity-mappings>

```

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



Collection<Entity> (Shared ForeignKey)



Supported for RDBMS.



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

```
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

```
import org.datanucleus.api.jpa.annotations.SharedRelation;

public class Account
{
    ...

    @OneToMany
    @SharedRelation(column="ADDRESS_TYPE", value="work")
    Collection<Address> workAddresses;

    @OneToMany
    @SharedRelation(column="ADDRESS_TYPE", value="home")
    Collection<Address> homeAddresses;

    ...
}
```

or using XML metadata

```

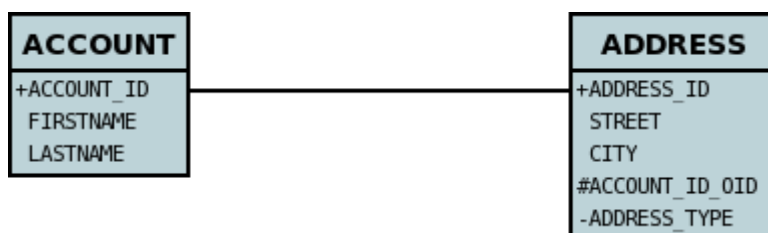
<entity-mappings>
  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      ...
      <one-to-many name="workAddresses">
        <join-column name="ACCOUNT_ID_OID"/>
        <extension key="relation-discriminator-column" value="ADDRESS_TYPE"/>
        <extension key="relation-discriminator-value" value="work"/>
      </one-to-many>
      <one-to-many name="homeAddresses">
        <join-column name="ACCOUNT_ID_OID"/>
        <extension key="relation-discriminator-column" value="ADDRESS_TYPE"/>
        <extension key="relation-discriminator-value" value="home"/>
      </one-to-many>
    </attributes>
  </entity>

  <entity class="Address">
    <table name="ADDRESS"/>
    ...
  </entity>
</entity-mappings>

```

We have defined the same foreign key for the 2 collections, named `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



Collection<Simple> (JoinTable)



Supported for RDBMS. Other datastores simply ignore the join table and store the collection in a column in the table of the object with the field.

All of the examples above show a 1-N relationship between 2 entities. 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

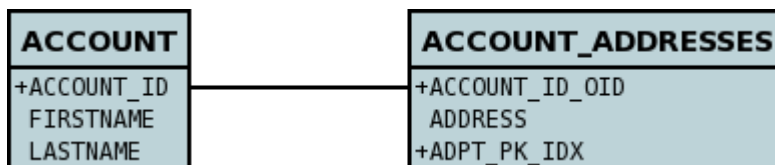
```
@Entity
public class Account
{
    ...

    @ElementCollection
    @CollectionTable(name="ACCOUNT_ADDRESSES")
    Collection<String> addresses;
}
```

or using XML metadata

```
<entity class="mydomain.Account">
  <attributes>
    ...
    <element-collection name="addresses">
      <collection-table name="ACCOUNT_ADDRESSES"/>
    </element-collection>
  </attributes>
</entity>
```

In the datastore the following is created



The **ACCOUNT** table is as before, but this time we only have the "join table". Use `@Column` on the field/method to define the column details of the element in the join table.

Collection<Simple> using AttributeConverter (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 JPA AttributeConverter.

```

public class Account
{
    ...

    @ElementCollection
    @Convert(CollectionToStringConverter.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 CollectionToStringConverter implements AttributeConverter
<Collection<String>, String>
{
    public String convertToDatabaseColumn(Collection<String> attribute)
    {
        if (attribute == null)
        {
            return null;
        }

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

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

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

```

Map<Simple, Entity> (JoinTable)



Supported for RDBMS. Other datastores simply ignore the join table and store the relation in a column in the table of the object with the field.

We have a class **Account** that contains a Map of Address objects. Here our key is a simple type (in this case a String) and the values are entities. Like this

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

    ...
}

public class Address {...}
```

If you define the annotations for these classes as follows

```
@Entity
public class Account
{
    @OneToMany
    @JoinTable
    Map<String, Address> addresses;

    ...
}

@Entity
public class Address {...}
```

or using XML

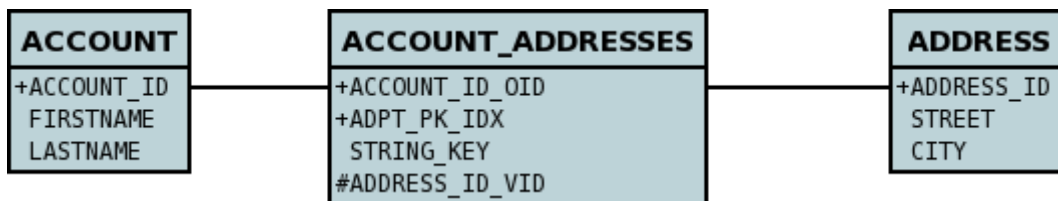
```

<entity-mappings>
  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      ...
      <one-to-many name="addresses">
        <join-table name="ACCOUNT_ADDRESSES"/>
        <column name="STRING_KEY"/>
      </one-to-many>
    </attributes>
  </entity>

  <entity class="Address">
    <table name="ADDRESS"/>
    <attributes>
      ...
    </attributes>
  </entity>
</entity-mappings>

```

This will create 3 tables in the datastore, one for **Account**, one for **Address** and a join table also containing the key.



You can configure the names of the key column(s) in the join table using the *joinColumns* attribute of `@JoinTable`, or the names of the value column(s) using `@Column` for the field/method.



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,Entity> 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. We're using a field (*alias*) in the Address class as the key of the map.

```

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

    ...
}

public class Address
{
    String alias;

    ...
}

```

In this relationship, the **Account** class has a Map of **Address** objects, yet the **Address** knows nothing about the **Account**. In this case we don't have a field in the Address to link back to the Account and so DataNucleus has to use columns in the datastore representation of the **Address** class. So we define the annotations like this

```

@Entity
public class Account
{
    @OneToMany
    @MapKey(name="alias")
    @JoinColumn(name="ACCOUNT_ID_OID")
    Map<String, Address> addresses;

    ...
}

@Entity
public class Address
{
    String alias;

    ...
}

```

or using XML metadata

```

<entity-mappings>
  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      ...
      <one-to-many name="addresses">
        <map-key name="alias"/>
        <join-column name="ACCOUNT_ID_OID"/>
      </one-to-many>
    </attributes>
  </entity>

  <entity class="Address">
    <table name="ADDRESS"/>
    <attributes>
      ...
      <basic name="alias">
        <column name="KEY" length="20"/>
      </basic>
    </attributes>
  </entity>
</entity-mappings>

```

Again there will be 2 tables, one for **Address**, and one for **Account**. If you wish to specify the names of the columns used in the schema for the foreign key in the **Address** table you should use the *join-column* element within the field of the map.



In terms of operation within your classes of assigning the objects in the relationship. You have to take your **Account** object and add the **Address** to the **Account** map field since the **Address** knows nothing about the **Account**. Also be aware that each **Address** object can have only one owner, since it has a single foreign key to the **Account**.



Since each Address object can have at most one key (due to the "Foreign Key") this mode of persistence will not allow duplicate values in the Map. If you want to allow duplicate Map values, then use the "Join Table" variant above.

Map<Simple,Entity> Bidirectional (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.

```

public class Account
{
    long id;

    Map<String, Address> addresses;

    ...
}

public class Address
{
    long id;

    String alias;

    Account account;

    ...
}

```

With these classes we want to store a foreign-key in the value table **ADDRESS**, and we want to use the "alias" field in the Address class as the key to the map. If you define the Meta-Data for these classes as follows

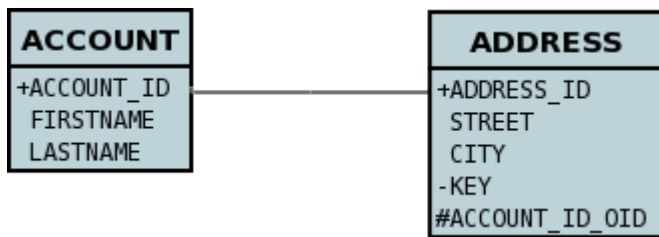
```

<entity-mappings>
  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      ...
      <one-to-many name="addresses" mapped-by="account">
        <map-key name="alias"/>
      </one-to-many>
    </attributes>
  </entity>

  <entity class="Address">
    <table name="ADDRESS"/>
    <attributes>
      ...
      <basic name="alias">
        <column name="KEY" length="20"/>
      </basic>
      <many-to-one name="account">
        <join-column name="ACCOUNT_ID_OID"/>
      </many-to-one>
    </attributes>
  </entity>
</entity-mappings>

```

This will create 2 tables in the datastore. One for **Account**, and one for **Address**. The table for **Address** will contain the key field as well as an index to the **Account** record (notated by the *mapped-by* tag).



Map<Simple, Simple> (JoinTable)



Supported for RDBMS. Other datastores simply ignore the join table and store the map in a column in the table of the object with the field.

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

```
@Entity
public class Account
{
    @ElementCollection
    @CollectionTable
    Map<String, String> addresses;

    ...
}
```

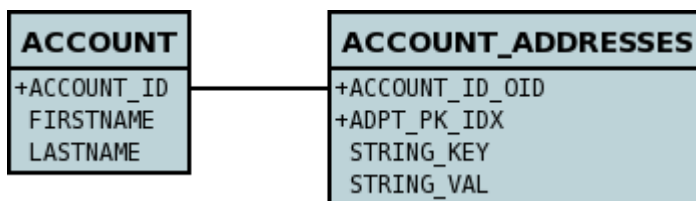
or using XML

```

<entity-mappings>
  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      ...
      <element-collection name="addresses">
        <collection-table name="ACCOUNT_ADDRESSES">
          <join-column name="STRING_VAL"/>
        </collection-table>
        <column name="STRING_KEY"/>
      </element-collection>
    </attributes>
  </entity>
</entity-mappings>

```

This results in just 2 tables. The "join" table contains both the key AND the value.



You can configure the names of the key column(s) in the join table using the *joinColumns* attribute of `@CollectionTable`, or the names of the value column(s) using `@Column` for the field/method.

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 (Column)

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

```

public class Account
{
  ...

  @ElementCollection
  @Convert(MapStringStringToStringConverter.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 convertToDatabaseColumn(Map<String, String> attribute)
    {
        if (attribute == null)
        {
            return null;
        }

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

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

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

Map<Entity, Entity> (JoinTable)



Supported for RDBMS. Other datastores simply ignore the join table and store the relation in a column in the table of the object with the field.

We have a class **Account** that contains a Map of Address objects. Here our key is an entity type and the values is an entity type also. Like this


```

public class Account
{
    Map<Name, Address> addresses;

    ...
}

public class Name {...}

public class Address {...}

```

If you define the annotations for these classes as follows

```

@Entity
public class Account
{
    @OneToMany
    @JoinTable
    Map<Name, Address> addresses;

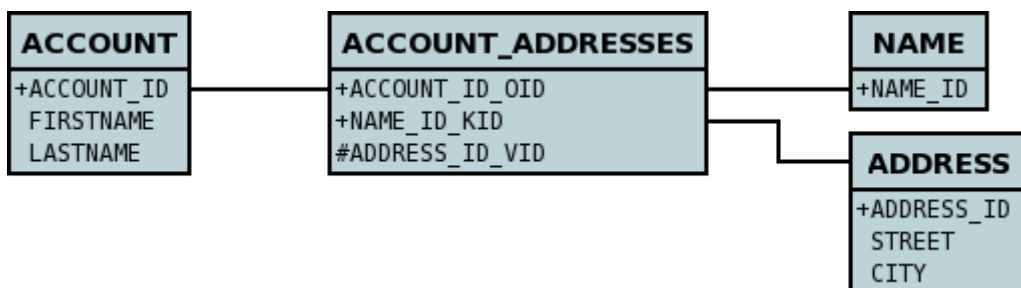
    ...
}

@Entity
public class Name {...}

@Entity
public class Address {...}

```

This will create 4 tables in the datastore, one for **Account**, one for **Name**, one for **Address** and a join table to link them.



You can configure the names of the key column(s) in the join table using the *joinColumns* attribute of `@JoinTable`, or the names of the value column(s) using `@Column` for the field/method.



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<Entity, Simple> (JoinTable)



Supported for RDBMS. Other datastores simply ignore the join table and store the relation in a column in the table of the object with the field.

Here our key is an entity type and the value is a simple type (in this case a String).



JPA does NOT properly allow for this in its specification. Other implementations introduced the following hack so we also provide it. Note that there is no `@OneToMany` annotation here so this is seemingly not a relation to JPA (hence our description of this as a hack). Anyway use it to workaround JPA's lack of feature.

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

```
@Entity
public class Account
{
    @ElementCollection
    @JoinTable
    Map<Address, String> addressLookup;

    ...
}

@Entity
public class Address {...}
```

This will create 3 tables in the datastore, one for **Account**, one for **Address** and a join table also containing the value.

You can configure the names of the columns in the join table using the *joinColumns* attributes of the various annotations.

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.



You cannot have an N-1 relation to a long or int field! JPA is for use with object-oriented systems, not flat data.

Unidirectional (ForeignKey)

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

```
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

```
@Entity
public class Account
{
    ...

    @ManyToOne
    User user;
}
```

or using XML metadata

```

<entity-mappings>
  <entity class="User">
    <table name="USER"/>
    <attributes>
      <id name="id">
        <column name="USER_ID"/>
      </id>
      ...
    </attributes>
  </entity>

  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      <id name="id">
        <column name="ACCOUNT_ID"/>
      </id>
      ...
      <many-to-one name="user"/>
    </attributes>
  </entity>
</entity-mappings>

```

This will create 2 tables in the database, **USER** (for class **User**), and **ACCOUNT** (for class **Account**), and a foreign-key in the **ACCOUNT** table, just like for the case of a [One-to-One relation](#).



in the case of non-RDBMS datastores there is simply a "column" in the **ACCOUNT** "table", storing the "id" of the related object

Unidirectional (JoinTable)



Supported for RDBMS. Other datastores simply ignore the join table and store the relation in a column in the table of the object with the field.

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 are using a join table. A particular user could be related to several accounts. If you define the annotations for these classes as follows

```
@Entity
public class Account
{
    @ManyToOne
    @JoinTable(name="ACCOUNT_USER")
    User user;

    ....
}
```

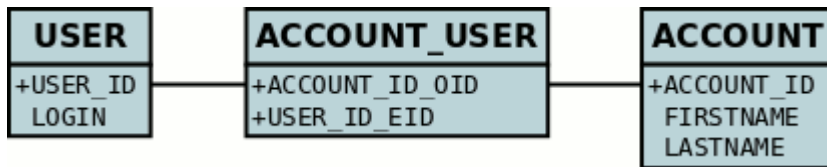
or using XML metadata

```
<entity-mappings>
  <entity class="User">
    <table name="USER"/>
    <attributes>
      <id name="id">
        <column name="USER_ID"/>
      </id>
      ...
    </attributes>
  </entity>

  <entity class="Account">
    <table name="ACCOUNT"/>
    <attributes>
      <id name="id">
        <column name="ACCOUNT_ID"/>
      </id>
      ...
      <many-to-one name="user">
        <join-table name="ACCOUNT_USER"/>
      </many-to-one>
    </attributes>
  </entity>
</entity-mappings>
```

alternatively using annotations

This will create 3 tables in the database, **USER** (for class **User**), **ACCOUNT** (for class **Account**), and a join table **ACCOUNT_USER**, as shown below.



For non-RDBMS datastores there is no join-table, simply a "column" in the **ACCOUNT** "table", storing the "id" of the related object

Bidirectional (ForeignKey)

This relationship is described in the guide for [1-N relationships](#). This [relation](#) 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.

Bidirectional (JoinTable)

This relationship is described in the guide for [1-N relationships](#). This [relation](#) uses a Join Table to link to the "N" object, with this table holding the relationship.



For non-RDBMS datastores there is no join table, and each side will have a "column" (or equivalent) in the "table", storing the "id" of the related object(s).

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 Collection, Set, List or subclasses of these, although the only one that supports a true M-N is Set.

With DataNucleus this can be set up as described in this section, using what is called a *Join Table* relationship. Let's take the following example and describe how to model it with the different types of collection classes. We have 2 classes, **Product** and **Supplier** as below.

```
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.



If you want to have a M-N relation between classes and think of adding extra information to the join table, please think about where in your Java model that "extra information" is stored, and consider how JPA can know where to persist it. You would model this situation by creating an intermediate persistable class containing the extra information and make the relations to this intermediate class.

The various possible relationships are described below.

- [M-N Set relation](#)
- [M-N Ordered List 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 in the equals/hashCode methods.

Using Set



Supported for RDBMS. Other datastores simply ignore the join table and store the relation in a column in the table of the object with the field.

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


```

public class Product
{
    ...

    @ManyToMany(mappedBy="products")
    @JoinTable(name="PRODUCTS_SUPPLIERS",
        joinColumns={@JoinColumn(name="PRODUCT_ID")},
        inverseJoinColumns={@JoinColumn(name="SUPPLIER_ID")})
    Set<Supplier> suppliers
}

public class Supplier
{
    ...

    @ManyToMany
    Set<Product> products;

    ...
}

```

or using XML metadata

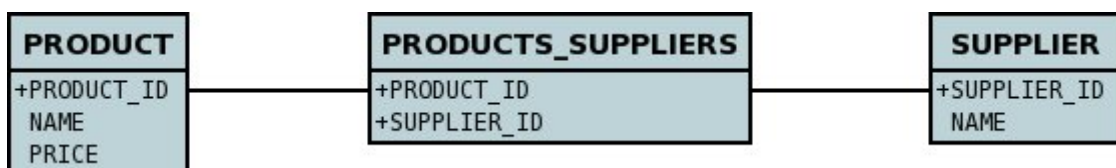
```

<entity-mappings>
  <entity class="mydomain.Product">
    <table name="PRODUCT"/>
    <attributes>
      <id name="id">
        <column name="PRODUCT_ID"/>
      </id>
      ...
      <many-to-many name="suppliers" mapped-by="products">
        <join-table name="PRODUCTS_SUPPLIERS">
          <join-column name="PRODUCT_ID"/>
          <inverse-join-column name="SUPPLIER_ID"/>
        </join-table>
      </many-to-many>
    </attributes>
  </entity>

  <entity class="mydomain.Supplier">
    <table name="SUPPLIER"/>
    <attributes>
      <id name="id">
        <column name="SUPPLIER_ID"/>
      </id>
      ...
      <many-to-many name="products"/>
    </attributes>
  </entity>
</entity-mappings>

```

Note how we have specified the information only once regarding join table name, and join column names as well as the `<join-table>@JoinTable`. This is the JPA standard way of specification, and results in a single join table. The "mapped-by" ties the two fields together.



Using Ordered Lists



Supported for RDBMS. Other datastores simply ignore the join table and store the relation in a column in the table of the object with the field.

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
{
    ...

    @ManyToMany
    @JoinTable(name="PRODUCTS_SUPPLIERS",
        joinColumns={@JoinColumn(name="PRODUCT_ID")},
        inverseJoinColumns={@JoinColumn(name="SUPPLIER_ID")})
    @OrderBy("id")
    List<Supplier> suppliers
}

public class Supplier
{
    ...

    @ManyToMany
    @OrderBy("id")
    List<Product> products
}

```

or using XML metadata

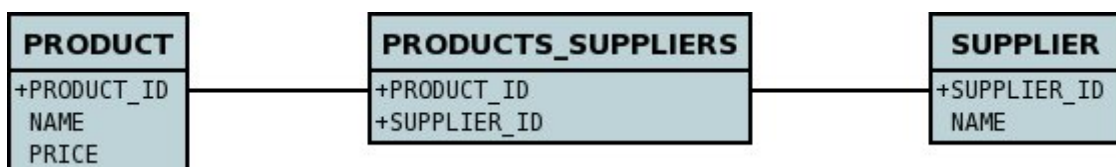
```

<entity-mappings>
  <entity class="mydomain.Product">
    <table name="PRODUCT"/>
    <attributes>
      <id name="id">
        <column name="PRODUCT_ID"/>
      </id>
      ...
      <many-to-many name="suppliers" mapped-by="products">
        <order-by>name</order-by>
        <join-table name="PRODUCTS_SUPPLIERS">
          <join-column name="PRODUCT_ID"/>
          <inverse-join-column name="SUPPLIER_ID"/>
        </join-table>
      </many-to-many>
    </attributes>
  </entity>

  <entity class="mydomain.Supplier">
    <table name="SUPPLIER"/>
    <attributes>
      <id name="id">
        <column name="SUPPLIER_ID"/>
      </id>
      ...
      <many-to-many name="products">
        <order-by>name</order-by>
      </many-to-many>
    </attributes>
  </entity>
</entity-mappings>

```

There will be 3 tables, one for **Product**, one for **Supplier**, and the join table. The difference from the Set example is that we now have `<order-by>@OrderBy` at both sides of the relation. This has no effect in the datastore schema but when the Lists are retrieved they are ordered using the specified *order-by*.



You cannot have a many-to-many relation using indexed lists since both sides would need its own index.

Arrays

JPA defines support the persistence of arrays but only arrays of byte, Byte, char, Character. DataNucleus supports all types of arrays, as follows

- [Single Column](#) - the array is byte-streamed into a single column in the table of the containing object.
- [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
- [Entity array via JoinTable](#) - the array is stored via a "join" table, with FK across to the element Entity.
- [Entity array via ForeignKey](#) - the array is stored via a FK in the element Entity.

Single Column Arrays (serialised)

Let's suppose you have a class something like this

```
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. We then define MetaData something like this

```
<entity class="Account">
  <table name="ACCOUNT"/>
  <attributes>
    ...
    <basic name="permissions">
      <column name="PERMISSIONS"/>
      <lob/>
    </basic>
    ...
  </attributes>
</entity>
```

This results in a datastore schema as follows

ACCOUNT
+ACCOUNT_ID
FIRST_NAME
LAST_NAME
PERMISSIONS

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[]*



When using PostgreSQL you can persist an array of *int*, *short*, *long*, *Integer*, *Short*, *Long* into a column of type *int array*, or an array of *String* into a column of type *text array*.

See also :-

- [MetaData reference for <basic> element](#)
- [Annotations reference for @Basic](#)

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

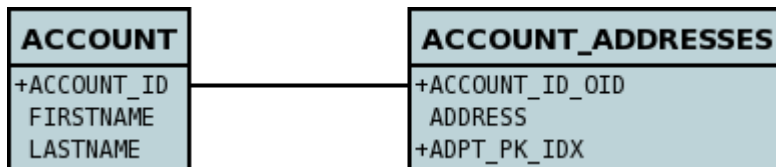
```
@Entity
public class Account
{
    ...

    @ElementCollection
    @CollectionTable(name="ACCOUNT_ADDRESSES")
    String[] addresses;
}
```

or using XML metadata

```
<entity class="mydomain.Account">
  <attributes>
    ...
    <element-collection name="addresses">
      <collection-table name="ACCOUNT_ADDRESSES"/>
    </element-collection>
  </attributes>
</entity>
```

In the datastore the following is created



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

Entity array persisted into Join Tables

DataNucleus will support arrays persisted into a join table. Let's take the example of a class `Account` with an array of `Permission` objects, so we have

```
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

```
@Entity
public class Account
{
    ...

    @OneToMany
    @JoinTable(name="ACCOUNT_PERMISSIONS", joinColumns={@Column(name="ACCOUNT_ID")},
    inverseJoinColumns={@Column(name="PERMISSION_ID")})
    @OrderColumn(name="PERMISSION_ORDER_IDX")
    Permission[] permissions;
}

@Entity
public class Permission
{
    ...
}
```

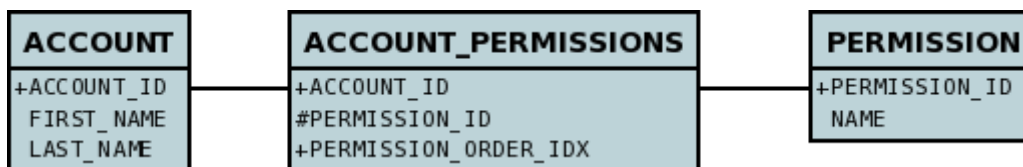
or using XML metadata

```

<entity class="mydomain.Account">
  <attributes>
    ...
    <one-to-many name="permissions">
      <join-table name="ACCOUNT_PERMISSIONS">
        <join-column name="ACCOUNT_ID"/>
        <inverse-join-column name="PERMISSION_ID"/>
      </join-table>
      <order-column name="PERMISSION_ORDER_IDX"/>
    </one-to-many>
  </attributes>
</entity>
<entity name="Permission" table="PERMISSION">
</entity>

```

This results in a datastore schema as follows



Entity array 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 an entity. Let's take the same example above. So we have

```

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 foreign-key in the table for the **Permission** class. We define the **MetaData** as follows


```

@Entity
public class Account
{
    @OneToMany
    @JoinColumn(name="ACCOUNT_ID")
    @OrderColumn(name="PERMISSION_ORDER_IDX")
    Permission[] permissions;

    ....
}

@Entity
public class Permission
{
    ...
}

```

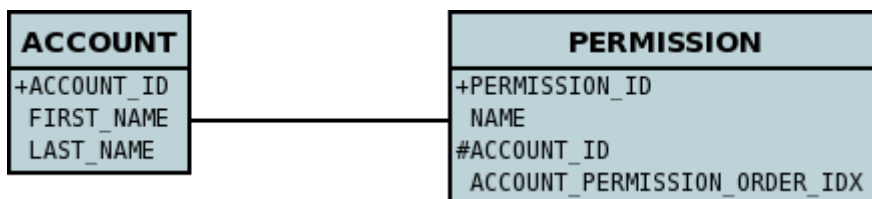
or using XML metadata

```

<entity class="mydomain.Account">
  <attributes>
    ...
    <one-to-many name="permissions">
      <join-column name="ACCOUNT_ID"/>
      <order-column name="PERMISSION_ORDER_IDX"/>
    </one-to-many>
  </attributes>
</entity>
<entity name="Permission" table="PERMISSION">
</entity>

```

This results in a datastore schema as follows



Interfaces



JPA doesn't define support for persisting fields of type interface, but DataNucleus provides an extension whereby the implementations of the interface are entities. It follows the same general process as for [java.lang.Object](#) since both interfaces and `java.lang.Object` are basically *references* to some entity.

To demonstrate interface handling let's introduce some classes. Suppose you have an interface with a selection of classes implementing the interface something like this

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

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

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

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

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

```
public class ShapeHolder
{
    protected Shape shape=null;
    ...
}
```

With non-RDBMS datastores there is a single column holding the interface field value. With RDBMS you have the option of storing this interface field value in several different ways, as below.



DataNucleus allows the following strategies for mapping this field

- **per-implementation** : a FK is created for each implementation so that the datastore can provide referential integrity. The other advantage is that since there are FKs then querying can be performed. The disadvantage is that if there are many implementations then the table can become large with many columns not used
- **identity** : a single column is added and this stores the class name of the implementation stored, as well as the identity of the object. The advantage is that if you have large numbers of implementations then this can cope with no schema change. The disadvantages are that no querying can be performed, and that there is no referential integrity.
- **xcalia** : a slight variation on "identity" whereby there is a single column yet the contents of that column are consistent with what Xcalia XIC JDO implementation stored there.

The user controls which one of these is to be used by specifying the *extension* **mapping-strategy** on the field containing the interface. The default is "per-implementation"

In terms of the implementations of the interface, you can either leave the field to accept any *known about* implementation, or you can restrict it to only accept some implementations (see "implementation-classes" metadata extension). If you are leaving it to accept any persistable implementation class, then you need to be careful that such implementations are known to DataNucleus at the point of encountering the interface field. By this we mean, DataNucleus has to have encountered the metadata for the implementation so that it can allow for the implementation when handling the field. You can force DataNucleus to know about a persistable class by using `persistence.xml` to specify the classes.

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.

```

@Entity
public class Square implement Shape
{
    ...
}
@Entity
public class Circle implement Shape
{
    ...
}
@Entity
public class Rectangle implement Shape
{
    ...
}

```

The global way means that when mapping that field DataNucleus will look at all Entities it knows about that implement the specified interface.

DataNucleus also allows users to specify a list of classes implementing the interface on a field-by-field basis, defining which of these implementations are accepted for a particular interface field. To do this you define the Meta-Data like this

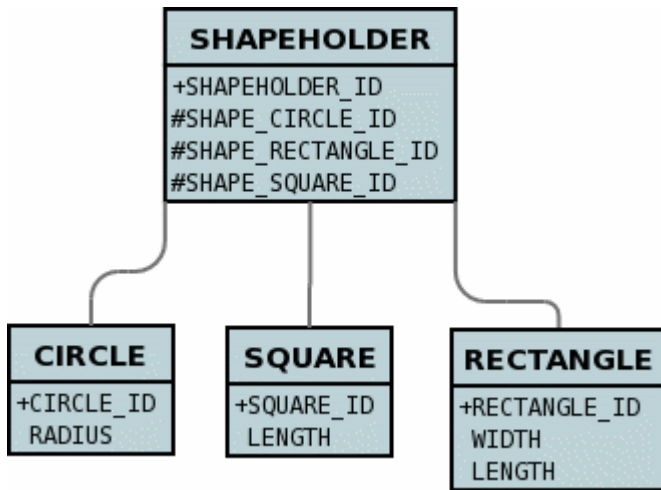
```

@Entity
public class ShapeHolder
{
    @OneToOne
    @Extension(key="implementation-classes",
        value="mydomain.Circle,mydomain.Rectangle,mydomain.Square")
    Shape shape;

    ...
}

```

That is, for any interface object in a class to be persisted, you define the possible implementation classes that can be stored there. DataNucleus interprets this information and will map the above example classes to the following in the database



So DataNucleus adds foreign keys from the containers table to all of the possible implementation tables for the *shape* field.

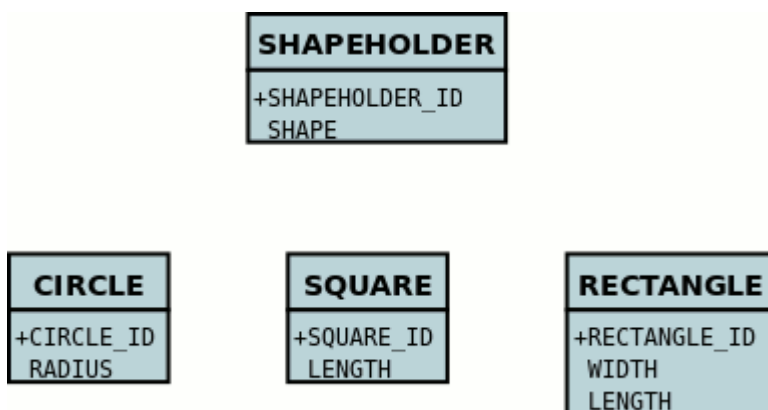
If we use **mapping-strategy** of "identity" then we get a different datastore schema.

```

@Entity
public class ShapeHolder
{
    @OneToOne
    @Extension(key="implementation-classes", value
    ="mydomain.Circle,mydomain.Rectangle,mydomain.Square")
    @Extension(key="mapping-strategy", value="identity")
    Shape shape;

    ...
}
  
```

and the datastore schema becomes



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

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 **implementation-classes** if you want to restrict the collection to only contain particular implementations of an interface. For example

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

    ...
}
```

So the *shapes* field is a Collection of *mydomain.Shape* and it will accept the implementations of type **Circle**, **Rectangle**, **Square** and **Triangle**. If you omit the **implementation-classes** extension then you have to give DataNucleus a way of finding the metadata for the implementations prior to encountering this field.

Dynamic Schema Updates (RDBMS)

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



JPA doesn't specify support for persisting fields of type `java.lang.Object`, however DataNucleus does support this where the values of that field are entities themselves. This follows the same general process as for [Interfaces](#) since both interfaces and `java.lang.Object` are basically *references* to some entity.



`java.lang.Object` cannot be used to persist non-entities with fixed schema datastore (e.g RDBMS). Think of how you would expect it to be stored if you think it ought to

DataNucleus allows the following ways of persisting Object fields :-

- **per-implementation** : a FK is created for each "implementation" so that the datastore can provide referential integrity. The other advantage is that since there are FKs then querying can be performed. The disadvantage is that if there are many implementations then the table can become large with many columns not used
- **identity** : a single column is added and this stores the class name of the "implementation" stored, as well as the identity of the object. The disadvantages are that no querying can be performed, and that there is no referential integrity.
- **xcalia** : a slight variation on "identity" whereby there is a single column yet the contents of that column are consistent with what Xcalia XIC JDO implementation stored there.

The user controls which one of these is to be used by specifying the extension **mapping-strategy** on the field containing the interface. The default is *per-implementation*.

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

```
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 annotations like this

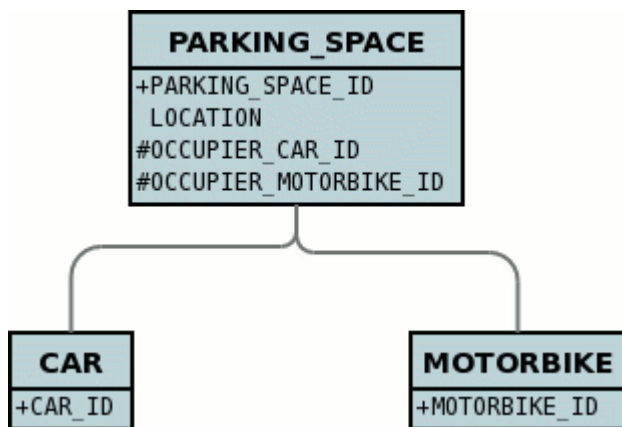
```

@Entity
public class ParkingSpace
{
    String location;

    @OneToOne
    @Extension(key="implementation-classes", value
    ="mydomain.vehicles.Car,mydomain.vehicles.Motorbike")
    Object occupier;
}

```

This will result in the following database schema.



So DataNucleus adds foreign keys from the ParkingSpace table to all of the possible implementation tables for the *occupier* field.

In conclusion, when using "per-implementation" mapping for any `java.lang.Object` field in a class to be persisted (as non-serialised), you **must** define the possible "implementation" classes that can be stored there.

If we use **mapping-strategy** of "identity" then we get a different datastore schema.

```

public class ParkingSpace
{
    String location;

    @OneToOne
    @Extension(key="implementation-classes", value
    ="mydomain.vehicles.Car,mydomain.vehicles.Motorbike")
    @Extension(key="mapping-strategy", value="identity")
    Object occupier;
}

```

and the datastore schema becomes

PARKING_SPACE
+PARKING_SPACE_ID
LOCATION
OCCUPIER

CAR
+CAR_ID

MOTORBIKE
+MOTORBIKE_ID

and the column "OCCUPIER" will contain strings such as *mydomain.vehicles.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 "lob", like this

```
public class MyClass
{
    @Lob
    Object myObject;
}
```

Please refer to the [serialised fields guide](#) for more details of storing objects in this way.

Embedded Fields

The JPA persistence strategy typically involves persisting the fields of any class into its own table, and representing any relationships from the fields of that class across to other tables. There are occasions when this is undesirable, maybe due to an existing datastore schema, because a more convenient datastore model is required, or maybe you simply don't want an instance of some type to have its own *identity* and so that type will only co-exist with an owner object. JPA allows the persistence of fields as *embedded* typically into the same table as the "owning" class. To define a class to be *embedded* into other objects you make use of the `@Embeddable` annotation (instead of `@Entity`) in this case.

```
@Embeddable
public class MyClass {}
```

or using XML metadata

```
<embeddable name="mydomain.MyClass">
    ...
</embeddable>
```

With the above MetaData (using the *embeddable* definition), in our application any objects of the class **MyClass** can be embedded into other objects.

JPA's definition of embedding encompasses several types of fields. These are described below

- [Embedded Entities](#) - where you have a 1-1 relationship and you want to embed the other Entity into the same table as the your object
- [Embedded Nested Entities](#) - like the first example except that the other object also has another Entity that also should be embedded
- [Embedded Collection elements](#) - where you want to embed the elements of a collection into a join table (instead of persisting them into their own table)
- [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)

Embedded entity structure

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.



You can have inheritance in an embedded entity but only using *SINGLE_TABLE* strategy using a discriminator (you must define it in the metadata of the embedded type). Note that this is a DataNucleus extension since JPA doesn't define any support for embedded inherited entities.

Embedding Entities (1-1)



Applicable to RDBMS, Excel, OOXML, 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 JPA and DataNucleus you can persist the related entity object as embedded into the same table. This results in a single table in the datastore rather than one for each of the 2 classes.

Let's take an example. We are modelling a **Computer**, and in our simple model our **Computer** has a graphics card and a sound card. So we model these cards using a **ComputerCard** class. So our classes become

```

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

COMPUTER
+COMPUTER_ID
OS_NAME
#GRAPHICSCARD_ID
#SOUNDCARD_ID

COMPUTERCARD
+COMPUTERCARD_ID
MANUFACTURER
TYPE

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
    @AttributeOverride(name="manufacturer", column=@Column(name
="GRAPHICS_MANUFACTURER"))
    @AttributeOverride(name="type", column=@Column(name="GRAPHICS_TYPE"))
    private ComputerCard graphicsCard;

    @Embedded
    @AttributeOverride(name="manufacturer", column=@Column(name="SOUND_MANUFACTURER"))
    @AttributeOverride(name="type", column=@Column(name="SOUND_TYPE"))
    private ComputerCard soundCard;

    ...
}

@Embeddable
public class ComputerCard
{
    ...
}

```

Or using XML metadata

```

<entity name="mydomain.Computer">
  <attributes>
    <basic name="operatingSystem">
      <column="OS_NAME"/>
    </basic>
    <embedded name="graphicsCard">
      <attribute-override name="manufacturer">
        <column="GRAPHICS_MANUFACTURER"/>
      </attribute-override>
      <attribute-override name="type">
        <column="GRAPHICS_TYPE"/>
      </attribute-override>
    </embedded>
    <embedded name="soundCard">
      <attribute-override name="manufacturer">
        <column="SOUND_MANUFACTURER"/>
      </attribute-override>
      <attribute-override name="type">
        <column="SOUND_TYPE"/>
      </attribute-override>
    </embedded>
  </attributes>
</entity>
<embeddable name="mydomain.ComputerCard">
  <attributes>
    <basic name="manufacturer"/>
    <basic name="type"/>
  </attributes>
</embeddable>

```

So here we will end up with a table called **COMPUTER** with columns **COMPUTER_ID**, **OS_NAME**, **GRAPHICS_MANUFACTURER**, **GRAPHICS_TYPE**, **SOUND_MANUFACTURER**, **SOUND_TYPE**. If we call `persist()` on any objects of type **Computer**, they will be persisted into this table.

COMPUTER
+COMPUTER_ID
OS_NAME
GRAPHICS_MANUFACTURER
GRAPHICS_TYPE
SOUND_MANUFACTURER
SOUND_TYPE

Null embedded objects



DataNucleus supports persistence of null embedded objects using the following metadata

```
@Extension(key="null-indicator-column", value="MY_COL")
@Extension(key="null-indicator-value", value="SomeValue")
```

and these will be used when persisting and retrieving the embedded object.

See also :-

- [MetaData reference for <embedded> element](#)
- [Annotations reference for @Embeddable](#)
- [Annotations reference for @Embedded](#)

MongoDB embedding control

For MongoDB you have one further control over how the persistable object is embedded. Since the datastore effectively is a JSON document, the default is to nest the embedded object, so our example could be represented as

```
{ "OS_NAME" : "Windows" ,
  "COMPUTER_ID" : 1 ,
  "graphicsCard" : { "GRAPHICS_MANUFACTURER" : "NVideo" ,
                     "GRAPHICS_TYPE" : "AGP"},
  "soundCard" : { "SOUND_MANUFACTURER" : "Intel" ,
                  "SOUND_TYPE" : "Other"}
}
```

If you set the field(s) to use **flat** embedding using the **nested** extension, like this

```
public class Computer
{
    @Embedded
    @AttributeOverride(name="manufacturer", column=@Column(name
="GRAPHICS_MANUFACTURER"))
    @AttributeOverride(name="type", column=@Column(name="GRAPHICS_TYPE"))
    @Extension(vendorName="datanucleus", key="nested", value="false")
    private ComputerCard graphicsCard;

    @Embedded
    @AttributeOverride(name="manufacturer", column=@Column(name="SOUND_MANUFACTURER"))
    @AttributeOverride(name="type", column=@Column(name="SOUND_TYPE"))
    @Extension(vendorName="datanucleus", key="nested", value="false")
    private ComputerCard soundCard;

    ...
}
```

then the resultant representation will be

```
{ "OS_NAME" : "Windows",
  "COMPUTER_ID" : 1,
  "GRAPHICS_MANUFACTURER" : "NVidia",
  "GRAPHICS_TYPE" : "AGP",
  "SOUND_MANUFACTURER" : "Intel",
  "SOUND_TYPE" : "Other"
}
```

Embedding Nested Entities



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

In the above example we had an embeddable entity within an entity. What if our embeddable object also contain another embeddable entity? Using the above example, what if **ComputerCard** contains an object of type **Connector** ?

```
@Embeddable
public class ComputerCard
{
    @Embedded
    Connector connector;

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

    ...
}

@Embeddable
public class Connector
{
    int type;
}
```

We want to store all of these objects into the same record in the **COMPUTER** table.


```

<entity name="mydomain.Computer">
  <attributes>
    <basic name="operatingSystem">
      <column="OS_NAME"/>
    </basic>
    <embedded name="graphicsCard">
      <attribute-override name="manufacturer">
        <column="GRAPHICS_MANUFACTURER"/>
      </attribute-override>
      <attribute-override name="type">
        <column="GRAPHICS_TYPE"/>
      </attribute-override>
      <attribute-override name="connector.type">
        <column="GRAPHICS_CONNECTOR_TYPE"/>
      </attribute-override>
    </embedded>
    <embedded name="soundCard">
      <attribute-override name="manufacturer">
        <column="SOUND_MANUFACTURER"/>
      </attribute-override>
      <attribute-override name="type">
        <column="SOUND_TYPE"/>
      </attribute-override>
      <attribute-override name="connector.type">
        <column="SOUND_CONNECTOR_TYPE"/>
      </attribute-override>
    </embedded>
  </attributes>
</entity>
<embeddable name="mydomain.ComputerCard">
  <attributes>
    <basic name="manufacturer"/>
    <basic name="type"/>
  </attributes>
</embeddable>
<embeddable name="mydomain.Connector">
  <attributes>
    <basic name="type"/>
  </attributes>
</embeddable>

```

So we simply nest the embedded definition of the **Connector** objects within the embedded definition of the **ComputerCard** definitions for **Computer**. JPA supports this to as many levels as you require! The **Connector** objects will be persisted into the **GRAPHICS_CONNECTOR_TYPE**, and **SOUND_CONNECTOR_TYPE** columns in the **COMPUTER** table.

COMPUTER
+COMPUTER_ID
OS_NAME
GRAPHICS_MANUFACTURER
GRAPHICS_TYPE
GRAPHICS_CONNECTOR_TYPE
SOUND_MANUFACTURER
SOUND_TYPE
SOUND_CONNECTOR_TYPE

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 using *SINGLE_TABLE* strategy with a discriminator (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)**. 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.

So we define our classes, using annotations, as follows

```

@Entity
public class Network
{
    private String name;

    @Embedded
    @ElementCollection
    @CollectionTable(name="NETWORK_DEVICES", joinColumns=@JoinColumn(name=
"NETWORK_ID"))
    private Collection<Device> devices = new HashSet<>();

    public Network(String name)
    {
        this.name = name;
    }

    ...
}

@Embeddable
public class Device
{
    @Column(name="DEVICE_NAME")
    private String name;

    @Column(name="DEVICE_IP_ADDR")
    private String ipAddress;

    public Device(String name, String addr)
    {
        this.name = name;
        this.ipAddress = addr;
    }

    ...
}

```

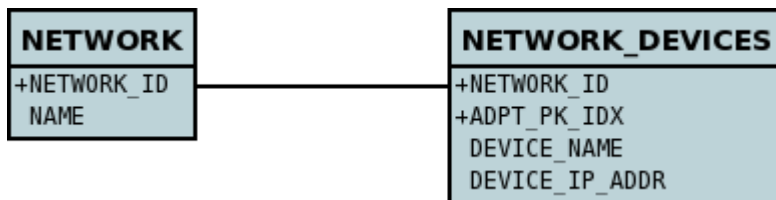
Alternatively using XML MetaData

```

<entity name="mydomain.Network">
  <attributes>
    ...
    <element-collection name="devices">
      <collection-table name="NETWORK_DEVICES">
        <join-column name="NETWORK_ID"/>
      </collection-table>
    </element-collection>
  </attributes>
</entity>
<embeddable name="mydomain.Device">
  <attributes>
    <basic name="name">
      <column="DEVICE_NAME"/>
    </basic>
    <basic name="ipAddress">
      <column="DEVICE_IP_ADDR"/>
    </basic>
  </attributes>
</embeddable>

```

So here we will end up with a table **NETWORK** with columns **NETWORK_ID**, and **NAME**, and a table **NETWORK_DEVICES** with columns **NETWORK_ID**, (optional **ADPT_PK_IDX**), **DEVICE_NAME**, **DEVICE_IP_ADDR**. When we persist a **Network** object, any devices are persisted into the **NETWORK_DEVICES** table.



If you want to override the name of the fields of the embedded element in the collection table, you could use `@AttributeOverride` (when using annotations) or `<attribute-override>` (when using XML).



By default a primary key constraint will be added to the table containing the embedded elements, formed from the id of the owner plus all columns representing fields of the element. You can turn this off using `@Extension(key="primary-key" value="false")`

See also :-

- [Metadata reference for <embeddable> element](#)
- [Metadata reference for <embedded> element](#)
- [Metadata reference for <element-collection> element](#)
- [Metadata reference for <collection-table> element](#)
- [Annotations reference for @Embeddable](#)

- [Annotations reference for @Embedded](#)
- [Annotations reference for @ElementCollection](#)

MongoDB embedded representation

Since the datastore with MongoDB is effectively a JSON document, our example would be represented as

```
{ "NAME" : "A Name" ,
  "NETWORK_ID" : 1 ,
  "devices" :
  [
    { "DEVICE_NAME" : "Laptop" ,
      "DEVICE_IP_ADDR" : "192.168.1.2" } ,
    { "DEVICE_NAME" : "Desktop" ,
      "DEVICE_IP_ADDR" : "192.168.1.3" } ,
    { "DEVICE_NAME" : "Smart TV" ,
      "DEVICE_IP_ADDR" : "192.168.1.4" }
  ]
}
```

Embedding Map Keys/Values



Applicable to RDBMS, MongoDB

In a typical 1-N map relationship between classes, the classes in the relationship are persisted to their own table, and a join table forms the map linkage. With JPA 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 using *SINGLE_TABLE* strategy with a discriminator (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, entity graphs and fetch specifications 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 **Films**, keyed by a String alias. So we define our classes as

```

@Entity
public class FilmLibrary
{
    private String owner;

    @Embedded
    @ElementCollection
    @CollectionTable(name="FILM_LIBRARY_FILMS")
    @MapKeyColumn(name="FILM_ALIAS")
    private Map<String, Film> films = new HashMap<>();

    public FilmLibrary(String owner)
    {
        this.owner = owner;
    }

    ...
}

public class Film
{
    @Column(name="FILM_NAME")
    private String name;

    @Column(name="FILM_DIRECTOR")
    private String director;

    public Film(String name, String director)
    {
        this.name = name;
        this.director = director;
    }

    ...
}

```

So here we will end up with a table **FILM_LIBRARY** with columns **FILM_LIBRARY_ID**, and **OWNER**, and a table **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.



Note that if you want to override the name of the fields of the embedded key/value in the table of the owner, you should use **@AttributeOverride** (when using annotations) or **<attribute-override>** (when using XML). In the case of fields of an embedded key you should set the name as

"key.{fieldName}" and in the case of fields of an embedded value you should set the name as "value.{fieldName}".

Serialised Fields

JPA provides a way for users to specify that a field will be persisted *serialised*. This is of use, for example, to collections/maps/arrays which typically are stored using join tables or foreign-keys to other records. By specifying that a field is serialised a column will be added to store that field and the field will be serialised into it.

JPA's definition of serialising applies to any field and all in the same way, unlike the situation with JDO which provides much more flexibility. Perhaps the most important thing to bear in mind when deciding to serialise a field is that that object in the field being serialised must implement *java.io.Serializable*.

Serialised Fields



Applicable to RDBMS, HBase, MongoDB

If you wish to serialise a particular field into a single column (in the table of the class), you need to simply mark the field as a "lob" (large object). Let's take an example. We have the following classes

```
public class Farm
{
    Collection<Animal> animals;

    ...
}

public class Animal
{
    ...
}
```

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

```
@Entity
public class Farm
{
    @Lob
    Collection<Animal> animals;

    ...
}
```

or using XML metadata


```

<entity class="Farm">
  <table name="FARM"/>
  <attributes>
    ...
    <basic name="animals">
      <column name="ANIMALS"/>
      <lob/>
    </basic>
    ...
  </attributes>
</entity>

```

So we make use of the `<lob>` element / `@Lob` annotation. This specification results in a table like this

FARM
+ID
NAME
ANIMALS



Queries cannot be performed on collections stored as serialised.



If the field that we want to serialise is of type String, byte array, char array, Byte array or Character array then the field will be serialised into a CLOB column rather than BLOB.

See also :-

- [MetaData reference for <basic> element](#)
- [Annotations reference for @Lob](#)

Serialise to File



Applicable to RDBMS

Note this is not part of the JPA spec, but is available in DataNucleus to ease your usage. If you have a non-relation field that implements Serializable you have the option of serialising it into a file on the local disk. This could be useful where you have a large file and don't want to persist very large objects into your RDBMS. Obviously this will mean that the field is no longer queryable, but then if its a large file you likely don't care about that. So let's give an example

```

@Entity
public class Person
{
    @Id
    long id;

    @Basic
    @Lob
    @Extension(vendorName="datanucleus", key="serializeToFileLocation", value
="person_avatars")
    AvatarImage image;
}

```

or using XML metadata

```

<entity class="Person">
    <attributes>
        ...
        <basic name="image">
            <lob/>
            <extension key="serializeToFileLocation" value="person_avatars"
        </basic>
        ...
    </attributes>
</entity>

```

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.

Datastore Schema

We have shown [earlier](#) how you define 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, according to the JPA specs rules.



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](#) - mapping classes/fields to table/columns
- [Column nullability and default value](#) - what values can be stored in a column
- [Column Types](#) - supported column types
- [Position of a column in a table](#) - allowing ordering of columns in the schema
- [Index Constraints](#) - used to mark fields that are referenced often as indexes so that when they are used the performance is optimised.
- [Unique Constraints](#) - placed on fields that should have a unique value. That is, only one object will have a particular value.
- [Foreign-Key Constraints](#) - used to interrelate objects, and allow the datastore to keep the integrity of the data in the datastore.
- [Primary-Key Constraints](#) - allow the PK to be set, and also to have a name.
- [RDBMS Views](#) - mapping a class to an RDBMS View instead of a Table

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

```
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 `ESTABLISHMENT`, and has columns `NAME`, `DIRECTION`, `PHONE` and `NUMBER_OF_ROOMS` (amongst other things). So we define our Meta-Data like this

```
<entity class="Hotel">
  <table name="ESTABLISHMENT"/>
  <attributes>
    <basic name="name">
      <column name="NAME"/>
    </basic>
    <basic name="address">
      <column name="DIRECTION"/>
    </basic>
    <basic name="telephoneNumber">
      <column name="PHONE"/>
    </basic>
    <basic name="numberOfRooms">
      <column name="NUMBER_OF_ROOMS"/>
    </basic>
  </attributes>
</entity>
```

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

```
@Table(name="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 consistent with the JPA specification. The table name will be based on the class name, and the column names will be based on the field names and the role of the field (if part of a relationship).

See also :-

- [Identifier Guide](#) - defining the identifiers to use for table/column names
- [MetaData reference for <column> element](#)

Column nullability and default values

So we've seen how to specify the basic structure of a table, naming the table and its columns, and how to control the types of the columns. We can extend this further to control whether the columns are allowed to contain nulls. Let's take a related class for our hotel. Here we have a class to model the payments made to the hotel.

```
public class Payment
{
    Customer customer;
    String bankTransferReference;
    String currency;
    double amount;
}
```

In this class we can model payments from a customer of an amount. Where the customer pays by bank transfer we can save the reference number. Since the bank transfer reference is optional we want that column to be nullable. So let's specify the `MetaData` for the class.

```
<entity class="Payment">
  <attributes>
    <one-to-one name="customer">
      <primary-key-join-column name="CUSTOMER_ID"/>
    </one-to-one>
    <basic name="bankTransferReference">
      <column name="TRANSFER_REF" nullable="true"/>
    </basic>
    <basic name="currency">
      <column name="CURRENCY"/>
    </basic>
    <basic name="amount">
      <column name="AMOUNT"/>
    </basic>
  </attributes>
</entity>
```

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

So we make use of the *nullable* attribute. The table, when created by DataNucleus, will then provide the nullability that we require.

Unfortunately with JPA there is no way to specify a default value for a field when it hasn't been set (unlike JDO where you can do that).

See also :-

- [MetaData reference for <column> element](#)

Column types

DataNucleus will provide a default type for any columns that it creates, but it will allow users to override this default. The default that DataNucleus chooses is always based on the Java type for the field being mapped. For example a Java field of type "int" will be mapped to a column type of INTEGER in RDBMS datastores. Similarly String will be mapped to VARCHAR.

JPA provides 2 ways of influencing the column DDL generated.

- You can specify the *columnDefinition* of `@Column/<column>` but you have to provide the complete DDL for that column (without the column name), and hence can lose database independence by using this route. e.g "VARCHAR(255)"
- Use `@Column/<column>` attributes and specify the *length/precision/scale* of the column, as well as whether it is unique etc. It will make use of the Java type to come up with a default datastore type for the column. Sadly JPA doesn't allow specification of the precise datastore type (except for BLOB/CLOB/TIME/TIMESTAMP cases). DataNucleus provides an extension to overcome this gap in the JPA spec. Here we make use of a DataNucleus extension annotation `@JdbcType` or *jdbc-type* extension attribute for `<column>`. Note that we could alternatively have made use of the DataNucleus extension annotation `@SqlType` or *sql-type* extension attribute for `<column>` to specify a low-level SQL type. Like this

```
<entity name="Payment">
  <attributes>
    <one-to-one name="customer">
      <primary-key-join-column name="CUSTOMER_ID"/>
    </one-to-one>
    <basic name="bankTransferReference">
      <column name="TRANSFER_REF" nullable="true" length="255"/>
    </basic>
    <basic name="currency">
      <column name="CURRENCY" length="3" jdbc-type="CHAR"/>
    </basic>
    <basic name="amount">
      <column name="AMOUNT" precision="10" scale="2"/>
    </basic>
  </attributes>
</entity>
```

You could alternatively specify these using annotations should you so wish. So we have defined TRANSFER_REF to use VARCHAR(255) column type, CURRENCY to use (VAR)CHAR(3) column type, and AMOUNT to use DECIMAL(10,2) column type.

See also :-

- [Types Guide](#) - defining mapping of Java types
- [RDBMS Types Guide](#) - defining mapping of Java types to JDBC/SQL types
- [MetaData reference for <column> element](#)

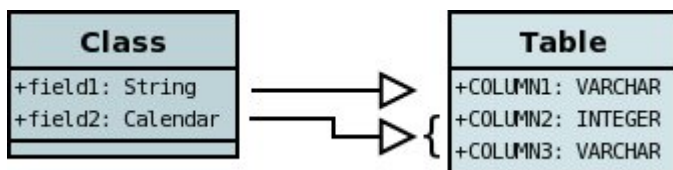
RDBMS Column Types



Applicable to RDBMS.

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 JPQL 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

Java Type	Number of Columns	Queryable	JDBC Type(s)
boolean	1	✓	BIT , CHAR ('Y','N'), BOOLEAN, TINYINT, SMALLINT, NUMERIC
byte	1	✓	TINYINT , SMALLINT, NUMERIC
char	1	✓	CHAR , INTEGER, NUMERIC
double	1	✓	DOUBLE , DECIMAL, FLOAT
float	1	✓	FLOAT , REAL, DOUBLE, DECIMAL
int	1	✓	INTEGER , BIGINT, NUMERIC
long	1	✓	BIGINT , NUMERIC, DOUBLE, DECIMAL, INTEGER
short	1	✓	SMALLINT , INTEGER, NUMERIC
boolean[]	1	✓ [5]	LONGVARBINARY, BLOB
byte[]	1	✓ [5]	LONGVARBINARY, BLOB
char[]	1	✓ [5]	LONGVARBINARY, BLOB

Java Type	Number of Columns	Queryable	JDBC Type(s)
double[]	1	✓ [5]	LONGVARBINARY, BLOB
float[]	1	✓ [5]	LONGVARBINARY, BLOB
int[]	1	✓ [5]	LONGVARBINARY, BLOB
long[]	1	✓ [5]	LONGVARBINARY, BLOB
short[]	1	✓ [5]	LONGVARBINARY, BLOB
java.lang.Boolean	1	✓	BIT , CHAR('Y','N'), BOOLEAN, TINYINT, SMALLINT
java.lang.Byte	1	✓	TINYINT , SMALLINT, NUMERIC
java.lang.Character	1	✓	CHAR , INTEGER, NUMERIC
java.lang.Double	1	✓	DOUBLE , DECIMAL, FLOAT
java.lang.Float	1	✓	FLOAT , REAL, DOUBLE, DECIMAL
java.lang.Integer	1	✓	INTEGER , BIGINT, NUMERIC
java.lang.Long	1	✓	BIGINT , NUMERIC, DOUBLE, DECIMAL, INTEGER
java.lang.Short	1	✓	SMALLINT , INTEGER, NUMERIC
java.lang.Boolean[]	1	✓ [5]	LONGVARBINARY, BLOB
java.lang.Byte[]	1	✓ [5]	LONGVARBINARY, BLOB
java.lang.Character[]	1	✓ [5]	LONGVARBINARY, BLOB
java.lang.Double[]	1	✓ [5]	LONGVARBINARY, BLOB
java.lang.Float[]	1	✓ [5]	LONGVARBINARY, BLOB
java.lang.Integer[]	1	✓ [5]	LONGVARBINARY, BLOB
java.lang.Long[]	1	✓ [5]	LONGVARBINARY, BLOB
java.lang.Short[]	1	✓ [5]	LONGVARBINARY, BLOB
java.lang.Number	1	✓	
java.lang.Object	1		LONGVARBINARY, BLOB
java.lang.String [8]	1	✓	VARCHAR , CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [6], UNIQUEIDENTIFIER [7], XMLTYPE [9]
java.lang.StringBuffer [8]	1	✓	VARCHAR , CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [6], UNIQUEIDENTIFIER [7], XMLTYPE [9]
java.lang.String[]	1	✓ [5]	LONGVARBINARY, BLOB
java.lang.Enum	1	✓	LONGVARBINARY, BLOB, VARCHAR, INTEGER

Java Type	Number of Columns	Queryable	JDBC Type(s)
java.lang.Enum[]	1	✓ [5]	LONGVARBINARY, BLOB
java.math.BigDecimal	1	✓	DECIMAL , NUMERIC
java.math.BigInteger	1	✓	NUMERIC , DECIMAL
java.math.BigDecimal[]	1	✓ [5]	LONGVARBINARY, BLOB
java.math.BigInteger[]	1	✓ [5]	LONGVARBINARY, BLOB
java.sql.Date	1	✓	DATE , TIMESTAMP
java.sql.Time	1	✓	TIME , TIMESTAMP
java.sql.Timestamp	1	✓	TIMESTAMP
java.util.ArrayList	0	✓	
java.util.BitSet	0	✗	LONGVARBINARY, BLOB
java.util.Calendar [3]	1 or 2	✗	INTEGER, VARCHAR, CHAR
java.util.Collection	0	✓	
java.util.Currency	1	✓	VARCHAR , CHAR
java.util.Date	1	✓	TIMESTAMP , DATE, CHAR, BIGINT
java.util.Date[]	1	✓ [5]	LONGVARBINARY, BLOB
java.util.GregorianCalendar [2]	1 or 2	✗	INTEGER, VARCHAR, CHAR
java.util.HashMap	0	✓	
java.util.HashSet	0	✓	
java.util.Hashtable	0	✓	
java.util.LinkedHashMap	0	✓	
java.util.LinkedHashSet	0	✓	
java.util.LinkedList	0	✓	
java.util.List	0	✓	
java.util.Locale [8]	1	✓	VARCHAR , CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [6], UNIQUEIDENTIFIER [7], XMLTYPE [9]
java.util.Locale[]	1	✓ [5]	LONGVARBINARY, BLOB
java.util.Map	0	✓	
java.util.Properties	0	✓	
java.util.PriorityQueue	0	✓	
java.util.Queue	0	✓	
java.util.Set	0	✓	

Java Type	Number of Columns	Queryable	JDBC Type(s)
java.util.SortedMap	0	✓	
java.util.SortedSet	0	✓	
java.util.Stack	0	✓	
java.util.TimeZone [8]	1	✓	VARCHAR , CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [6], UNIQUEIDENTIFIER [7], XMLTYPE [9]
java.util.TreeMap	0	✓	
java.util.TreeSet	0	✓	
java.util.UUID [8]	1	✓	VARCHAR , CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [6], UNIQUEIDENTIFIER [7], XMLTYPE [9]
java.util.Vector	0	✓	
java.awt.Color [1]	4	✗	INTEGER
java.awt.Point [2]	2	✗	INTEGER
java.awt.image.BufferedImage [4]	1	✗	LONGVARBINARY, BLOB
java.net.URI [8]	1	✓	VARCHAR , CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [6], UNIQUEIDENTIFIER [7], XMLTYPE [9]
java.net.URL [8]	1	✓	VARCHAR , CHAR, LONGVARCHAR, CLOB, BLOB, DATALINK [6], UNIQUEIDENTIFIER [7], XMLTYPE [9]
java.io.Serializable	1	✗	LONGVARBINARY, BLOB
Entity	1	✓	[embedded]
Entity[]	1	✓ [5]	

- **[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 [Column Adapter guide](#).
- **[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 an extension point.

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

JDBC Type	Supported	Restrictions
ARRAY	✓	Only for PostgreSQL array type
BIGINT	✓	
BINARY	✓	Only for geospatial types on MySQL
BIT	✓	
BLOB	✓	
BOOLEAN	✓	
CHAR	✓	
CLOB	✓	
DATALINK	✓	Only on DB2
DATE	✓	
DECIMAL	✓	
DISTINCT	✗	
DOUBLE	✓	
FLOAT	✓	
INTEGER	✓	
JAVA_OBJECT	✗	
LONGVARBINARY	✓	
LONGVARCHAR	✓	
NCHAR	✓	
NULL	✗	
NUMERIC	✓	
NVARCHAR	✓	
OTHER	✓	
REAL	✓	
REF	✗	
SMALLINT	✓	

JDBC Type	Supported	Restrictions
STRUCT	✓	Only for geospatial types on Oracle
TIME	✓	
TIMESTAMP	✓	
TINYINT	✓	
VARBINARY	✓	
VARCHAR	✓	

Field/Column Positioning 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. JPA doesn't allow configuration of this, but DataNucleus provides the following vendor extension.

Using XML metadata this would be

```
<entity name="mydomain.Person">
  <attributes>
    <id name="personNum">
      <column position="0"/>
    </id>
    <basic name="firstName">
      <column position="1"/>
    </basic>
    <basic name="lastName">
      <column position="2"/>
    </basic>
  </attributes>
</entity>
```

or using (DataNucleus) annotations

```
import org.datanucleus.api.jpa.annotations.ColumnPosition;

@Entity
public class Person
{
    @Id
    @ColumnPosition(0)
    long personNum;

    @ColumnPosition(1)
    String firstName;

    @ColumnPosition(2)
    String lastName;
}
```

Index Constraints



Applicable to RDBMS, NeoDatis, MongoDB

Many datastores provide the ability to have indexes defined to give performance benefits. With RDBMS the indexes are specified on the table and the indexes to the rows are stored separately. In the same way an ODBMS typically allows indexes to be specified on the fields of the class, and these are managed by the datastore. JPA allows you to define the indexes on a table-by-table basis by metadata as in the following example (note that you cannot specify indexes on a field basis like in JDO)

```
import javax.persistence.Index;

@Entity
@Table(indexes={@Index(name="SOME_VAL_IDX", columnList="SOME_VALUE")})
public class MyClass
{
    @Column(name="SOME_VALUE")
    long someValue;

    ...
}
```



The JPA `@Index` annotation is only applicable at a class level. DataNucleus provides its own `@Index` annotation that you can specify on a field/method to signify that the column(s) for this field/method will be indexed. Like this

```
@Entity
public class MyClass
{
    @org.datanucleus.api.jpa.annotations.Index(name="VAL_IDX")
    long someValue;

    ...
}
```

Specifying index type



Some RDBMS allow you to specify the *type* of an index, such as HASH etc. You can control this as follows

```
@Index(name="MY_BOOKING_IDX")
@Extension(key="index-type", value="HASH"))
```

which will create an index with `CREATE UNIQUE HASH INDEX MY_BOOKING_IDX ON BOOKING (BOOKING)`

Enhanced index creation



A final extension is where you just want to dump an amount of SQL onto the end of the `CREATE INDEX` statement (totally RDBMS dependent). We would advise against using this method due to its dependency on the RDBMS

```
@Index(name="MY_BOOKING_IDX")
@Extension(key="extended-setting", value=" USING HASH"))
```

See also :-

- [MetaData reference for <index> element](#)
- [Annotations reference for @Index](#)
- [Annotations reference for @Index \(class level\)](#)

Cassandra : index USING



Cassandra allows creation of indexes with an optional USING keyword. You can specify this via the

following extension

```
@Index(name="MY_BOOKING_IDX")
@Extension(key="cassandra.createIndex.using", value
="org.apache.cassandra.index.sasi.SASIIndex"))
```

and the **USING** clause will be appended to any **CREATE INDEX** issued during schema generation.

Unique Constraints



Applicable to RDBMS, NeoDatis, MongoDB

Some datastores provide the ability to have unique constraints defined on tables to give extra control over data integrity. JPA provides a mechanism for defining such unique constraints. Let's take an example class, and show how to specify this

```
public class Person
{
    String forename;
    String surname;
    String nickname;
    ...
}
```

and here we want to impose uniqueness on the "nickname" field, so there is only one Person known as "DataNucleus Guru" for example !

```
<entity class="Person">
  <table name="PEOPLE"/>
  <attributes>
    ...
    <basic name="nickname">
      <column name="SURNAME" unique="true"/>
    </basic>
    ...
  </attributes>
</entity>
```

The second use of unique constraints is where we want to impose uniqueness across composite columns. So we reuse the class above, and this time we want to impose a constraint that there is only one Person with a particular "forename+surname".

```

<entity class="Person">
  <table name="PEOPLE">
    <unique-constraint>
      <column-name>FORENAME</column-name>
      <column-name>SURNAME</column-name>
    </unique-constraint>
  </table>
  <attributes>
    ...
    <basic name="forename">
      <column name="FORENAME"/>
    </basic>
    <basic name="surname">
      <column name="SURNAME"/>
    </basic>
    ...
  </attributes>
</entity>

```

In the same way we can also impose unique constraints on `<join-table>` and `<secondary-table>`

See also :-

- [MetaData reference for <column> element](#)
- [MetaData reference for <unique-constraint> element](#)
- [Annotations reference for @Column](#)
- [Annotations reference for @UniqueConstraint](#)

Foreign Key Constraints



Applicable to RDBMS

When objects have relationships with one object containing, for example, a Collection of another object, it is common to store a foreign key in the datastore representation to link the two associated tables. Moreover, it is common to define behaviour about what happens to the dependent object when the owning object is deleted. Should the deletion of the owner cause the deletion of the dependent object maybe ? JPA supports defining the foreign key for relation fields as per the following example


```

public class MyClass
{
    ...

    @OneToOne
    @JoinColumn(name="OTHER_ID", foreignKey=@ForeignKey(name="OTHER_FK",
        foreignKeyDefinition="FOREIGN KEY (OTHER_ID) REFERENCES MY_OTHER_TBL
(MY_OTHER_ID) ]"))
    MyOtherClass other;
}

```

Note that when you don't specify any foreign key the JPA provider is free to add the foreign keys that it considers are necessary.

In the case of a 1-N/M-N relation using a join table the equivalent example would be

```

public class Account
{
    ...

    @OneToMany
    @JoinTable(foreignKey=@ForeignKey(name="ACCOUNT_FK"), inverseForeignKey
=@ForeignKey(name="ADDRESS_FK"))
    Collection<Address> addresses;
}

public class Address
{
    ...
}

```

or alternatively using XML

```

<entity-mappings>
  <entity class="Account">
    <attributes>
      ...
      <one-to-many name="addresses">
        <join-table>
          <foreign-key name="ACCOUNT_FK"/>
          <inverse-foreign-key name="ADDRESS_FK"/>
        </join-table>
      </one-to-many>
    </attributes>
  </entity>

  <entity class="Address">
    <attributes>
      ...
    </attributes>
  </entity>
</entity-mappings>

```

Primary Key Constraints



Applicable to RDBMS

In RDBMS datastores, it is accepted as good practice to have a primary key on all tables. You specify in other parts of the MetaData which fields are part of the primary key (if using application identity). Unfortunately JPA doesn't allow specification of the name of the primary key constraint, nor of whether join tables are given a primary key constraint at all.

RDBMS : Views



Applicable to RDBMS.



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. This provides more flexibility to the user where they have data and need to display it in their application. Support for Views is described below.

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

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

So we define a class to receive the values from this **View**, and define how it is mapped to the view.

```

package mydomain.views;

@Entity
@Table("SALEABLE_PRODUCT")
@Extension(vendorName="datanucleus", key="view-definition", value="CREATE VIEW
SALEABLE_PRODUCT
(
    {this.id},
    {this.name},
    {this.price},
    {this.currency}
) AS
SELECT ID, NAME, CURRENT_PRICE AS PRICE, CURRENCY FROM PRODUCT
WHERE PRODUCT.STATUS_ID = 1")
public class SaleableProduct
{
    @Id
    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;
    }
}

```

Please note the following

- 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 JPA querying operation.

```
Query<MyViewClass> q = em.createQuery("SELECT p FROM SaleableProduct p",  
SaleableProduct.class);  
List<MyViewClass> results = q.getResultList();
```

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

Secondary Tables



Applicable to RDBMS

The standard JPA persistence strategy is to persist an object of a class into its own table. In some situations you may wish to map the class to a primary table as well as one or more secondary tables. For example when you have a Java class that could have been split up into 2 separate classes yet, for whatever reason, has been written as a single class, however you have a legacy datastore and you need to map objects of this class into 2 tables. JPA allows persistence of fields of a class into *secondary* tables.



A Secondary table entry maps 1-1 to an owner table entry, and has the same primary key as the owner.

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 mydomain.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

```

@Entity
@Table(name="PRINTER")
@SecondaryTable(name="PRINTER_TONER", pkJoinColumns=@PrimaryKeyJoinColumn(name="PRINTER_REFID"))
public class Printer
{
    ...

    @Column(name="MODEL", table="PRINTER_TONER")
    String tonerModel;

    @Column(name="LIFETIME", table="PRINTER_TONER")
    int tonerLifetime;
}

```

or using XML metadata

```

<entity class="Printer">
  <table name="PRINTER"/>
  <secondary-table name="PRINTER_TONER">
    <primary-key-join-column name="PRINTER_REFID"/>
  </secondary-table>

  <attributes>
    ...
    <basic name="tonerModel">
      <column name="MODEL" table="PRINTER_TONER"/>
    </basic>
    <basic name="tonerLifetime">
      <column name="LIFETIME" table="PRINTER_TONER"/>
    </basic>
  </attributes>
</entity>

```

Here we have defined that objects of the **Printer** class will be stored in the primary table **PRINTER**. In addition we have defined that some fields are stored in the table **PRINTER_TONER**.

- We declare the "secondary-table"(s) that we will be using at the start of the definition.
- We define *tonerModel* and *tonerLifetime* to use columns in the table **PRINTER_TONER**. This uses the "table" attribute of <column>
- Whilst defining the secondary table(s) we will be using, we also define the join column to be called **PRINTER_REFID**.

This results in the following database tables :-

PRINTER	PRINTER_TONER
+PRINTER_ID MAKE MODEL	+PRINTER_REFID MODEL LIFETIME

So we now have our primary and secondary database tables. The primary key of the **PRINTER_TONER** table serves as a foreign key to the primary class. Whenever we persist a **Printer** object a row will be inserted into **both** of these tables.

See also :-

- [MetaData reference for <secondary-table> element](#)
- [MetaData reference for <column> element](#)
- [Annotations reference for @SecondaryTable](#)
- [Annotations reference for @Column](#)

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.



Some identifiers are actually reserved keywords with RDBMS, meaning that to use them you have to quote them. DataNucleus JPA quotes these automatically for you so you don't have to think about it, whereas other JPA providers force you to quote these yourself!

With RDBMS the generation of identifier names is controlled by an IdentifierFactory, and DataNucleus provides a default implementation for JPA. You can [provide your own RDBMS IdentifierFactory plugin](#) to give your own preferred naming if so desired. For RDBMS 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.

- [jpa](#) RDBMS IdentifierFactory (default for JPA persistence for RDBMS)

With non-RDBMS the generation of identifier names is controlled by a NamingFactory and again a default implementation for JPA. 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* to give your own preferred naming if so desired. Set it to the symbolic name of the factory you want to use.

- [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

```
public class MyClass
{
    String myField1;
    Collection<MyElement> elements1; // Using join table
    Collection<MyElement> elements2; // Using foreign-key
}

class MyElement
{
    String myElementField;
    MyClass myClass2;
}
```

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** (unless *entityName* had been specified, in

which case use that)

- When using datastore identity **MYCLASS** will have a column called **MYCLASS_ID**
- *MyClass.myField1* will be persisted into a column called **MYFIELD1**
- *MyElement* will be persisted into a table named **MYELEMENT**
- *MyClass.elements1* will be persisted into a join table called **MYCLASS_MYELEMENT**
- **MYCLASS_ELEMENTS1** will have columns called **MYCLASS_MYCLASS_ID** (FK to owner table) and **ELEMENTS1_ELEMENT_ID** (FK to element table)
- *MyClass.elements2* will be persisted into a column **ELEMENTS2_MYCLASS_ID** (FK to owner) table
- Any discriminator column will be called **DTYPE**
- Any index column in a List for field *MyClass.myField1* will be called **MYFIELD1_ORDER**
- Any adapter column added to a join table to form part of the primary key will be called **IDX**
- Any version column for a table will be called **VERSION**

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 *IdentifierFactory* mean that, assuming that the user doesn't specify any <column> elements :-

- *MyClass* will be persisted into a table named **MYCLASS** (unless *entityName* had been specified, in which case use that)
- When using datastore identity **MYCLASS** will have a column called **MYCLASS_ID**
- *MyClass.myField1* will be persisted into a column called **MYFIELD1**
- *MyElement* will be persisted into a table named **MYELEMENT**
- *MyClass.elements1* will be persisted into a join table called **MYCLASS_MYELEMENT**
- **MYCLASS_ELEMENTS1** will have columns called **MYCLASS_MYCLASS_ID** (FK to owner table) and **ELEMENTS1_ELEMENT_ID** (FK to element table)
- *MyClass.elements2* will be persisted into a column **ELEMENTS2_MYCLASS_ID** (FK to owner) table
- Any discriminator column will be called **DTYPE**
- Any index column in a List for field *MyClass.myField1* will be called **MYFIELD1_ORDER**
- Any adapter column added to a join table to form part of the primary key will be called **IDX**
- Any version column for a table will be called **VERSION**

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).

NOTE : 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.

NOTE : 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)