Hibernate Mapping Cheat Sheet

Copyright (c) 2005-2012 NDP Software. Some Rights Reserved.

Based on http://www.xylax.net/hibernate/intro.html (no longer live)

➤ Types

`integer, long, short`	Java primitives or wrapper classes to appropriate (vendor-specific) SQL column types.
`float, double`	Java primitives or wrapper classes to appropriate (vendor-specific) SQL column types.
`character, byte`	Java primitives or wrapper classes to appropriate (vendor-specific) SQL column types.
`boolean, yes_no, true_false`	Alternative encodings for a Java boolean or java.lang.Boolean
`string`	java.lang.String to VARCHAR (or Oracle VARCHAR2).
`date, time, timestamp`	java.util.Date and its subclasses to SQL types DATE, TIME and

➤ Object Declaration

Mapping

```
<class name="Foo" table="foo">
    <id name="id" column="id" type="int">
      <generator class="assigned"/>
    </id>
    <property name="propertyName" type="typename"
      access="field|property|ClassName"
      lazy="true|false"
      column="column_name" unique="true|false"
      not-null="true|false" length="L"
      index="index_name" unique_key="unique_key_id"
      optimistic-lock="true|false"
      node="element-name|@attr-name|element/@attr|."
    />
</class>
```

➤ Simple Association (one-to-one)

Mapping

```
<class name="Foo" table="foo"
  <one-to-one name="bar" class="Bar"/>
</class>
```

```
Bar Foo.getBar() // corresponding Bar instance
```

No extra columns are needed to support this relationship; Foo and Bar share the same PK values.

Bidirectionality

This relationship can be bidirectional, with Bar having getFoo(), by simply adding a similar mapping and Foo property to Bar.

	TIMESTAMP (or equivalent).
`calendar, calendar_date`	java.util.Calendar to SQL types TIMESTAMP and DATE (or equivalent).
`big_decimal, big_integer`	java.math.BigDecimal and java.math.BigInteger to NUMERIC (or Oracle NUMBER).
`locale, timezone, currency`	java.util.Locale, java.util.TimeZone and java.util.Currency to VARCHAR (or Oracle VARCHAR2). Instances of Locale and Currency are mapped to their ISO codes. Instances of TimeZone are mapped to their ID
`class`	java.lang.Class to VARCHAR (or Oracle VARCHAR2). A Class is mapped to its fully qualified name.
`binary`	Maps byte arrays to an appropriate SQL binary type.
`text`	Maps long Java strings to a SQL CLOB or TEXT type.
`serializable`	Maps serializable Java types to an appropriate SQL binary type. You may also indicate the Hibernate type serializable with the name of

➤ Simple Reference (many-to-one)

Mapping

```
<class name="Foo" table="foo">
  ...
  <many-to-one name="bar" class="Bar"
               column="bar_id"/>
</class>
```

Foo's table has an extra column which holds the FK to Bar. Foo and Bar can have completely different PKs and the relationship will still hold.

Bidirectionality

This relationship can be declared both ways, with Bar having *getFoo()*, by simply adding a similar mapping and property to Bar. This will result in Bar's table getting an extra column *foo_id*.

➤ Basic Collection (one-to-many)

We have two classes, Foo and Bar which are related to each other as follows:

```
Set Foo.getBars() // of Bar instances
```

Mapping

```
<class name="Foo" table="foo">
  ...
  <set role="bars" table="bar">
   <key column="foo_id"/>
   <one-to-many class="Bar"
     not-found="ignore|exception"/>
   </set>
</class>
```

The *<set>* collection is respresentative of many collection types.

Bar's table requires an extra column, which holds the FK to Foo. This allows Foo to be assigned a collection of Bars based on the value of the foo_id column in Bar.

a serializable Java class or interface that does not default to a basic type.

`clob, blob`

Type mappings for the JDBC classes java.sql.Clob and java.sql.BlobThese types may be inconvenient for some applications, since the blob or clob object may not be reused outside of a transaction. (Furthermore, driver support is patchy and inconsistent.)

Bidirectionality

This relationship can be declared both ways, with Bar having getFoo(), by suitable code changes to Bar and the following schema change:

Mapping

```
<class name="Bar" table="bar">
  ...
    <many-to-one name="foo" class="Foo"
    column="foo_id"/>
</class>
```

Now your Bars will know who their Foo is. No extra columns are generated for the bidirectionality.

➤ Collection (many-to-many)

A many-to-many reference is basically a collection. Class A holds a reference to a set of class B instances (as in the one-to-many case), but B might have multiple A's.

Scenario

We have two classes, Foo and Bar which are related to each other as follows:

```
Set Foo.getBars() // of Bar instances
```

Mapping

```
<class name="Foo" table="foo">
  ...
  <set role="bars" table="foo_bar">
     <key column="foo_id"/>
     <many-to-many column="bar_id" class="Bar"/>
  </set>
</class>
```

This time we cannot have an extra column on Bar as that would dictate that each Bar has only one Foo. So instead we have an extra table, *foo_bar*, which holds the relationship between instances.

Bidirectionality

This relationship can be declared both ways, with Bar having getFoos(), by suitable code changes to Bar and the following schema

change:

```
<class name="Bar" table="bar">
  ...
  <set role="foos" table="foo_bar" readonly="true">
    <key column="bar_id"/>
    <many-to-many column="foo_id" class="Foo"/>
  </set>
</class>
```
Mapping

Now your Bars will know who their Foos are.

No extra columns are generated for the bidirectionality.
Note that one end of the relationship must be declared "readonly".

If you want independent collections of Foos on Bars and Bars on Foos (i.e. membership one way doesn't imply the other), you need to declare Bar's table to be *bar_foo*. That way an independent table will be used to keep track of the Foo set on Bar.

➤ Collection (raw data)

A raw data collection is a collection on a class that contains second-rank classes. First- rank class A holds a reference to a set of second-rank class B instances. This is not limited to full classes - B could even be of primitive type.

Scenario

We have one class, Foo, and a collection of Strings (e.g. people's names)

```
Set Foo.getPeople() // of String instances
```

```
<class name="Foo" table="foo">
  ...
  <set role="people" table="Person">
    <key column="foo_id"/>
    <element column="name" type="string"/>
  </set>
</class>
```
Mapping

➤ Top-level Collections

A top-level collection is a collection defined outside of the scope of an individual class and available for use in all classes in the mapping files.

Scenario

We have one class, Foo, and a collection of Strings (e.g. people's names) which we wish to make available to other classes without constantly declaring set definitions inside each one.

```
Set Foo.getNames() // of String instances
```

```
<set role="names" table="names">
  <key column="id" type="string">
  <generator class="uuid.hex"/> </key>
  <element column="name" type="string"/>
</set>
<class name="Foo" table="foo">
...
<collection name="names" column="name_id"
```
Mapping

Note that Person does not represent a class. It is simply a collection of second-rank persistent data - in this case String objects.

Bidirectionality

There's no bidirectional relationship available here as there is only one class involved.

➤ Map

A map is a simple name-value pair list stored on a first rank collection.

Scenario

First rank class Foo has a map containing people's ages indexed by their names

```
Map Foo.getAges() // of String name-value pairs
```

Mapping

```
<class name="Foo" table="foo">
  ...
  <map role="ages">
    <key column="id"/>
    <index column="name" type="string"/>
    <element column="age" type="string"/>
  </map>
</class>
```

A simple extra table, *Ages*, is used to store the name and age string-value pair. Note that the map needs its own identity column too: *id*.

Bidirectionality

```
    role="names"/>
</class>
```

Note that a top-level collection needs its own key generator, and that this cannot be of the *assigned* type since it is never exposed to a calling application.

Again, Person does not represents a class. It is simply a collection of second-rank persistent objects - in this case Strings. Note also that Names simply has *id* not *foo_id*. This is to allow it to be used by a variety of classes. Also, because we cannot use Person's *id* as a *foo_id*, we have added a *person_id* FK to Foo.

Bidirectionality

There's no bidirectional relationship available here as there is only one class involved.

➤ Entity Map

An entity map is a map who is keyed by an entity class rather than a simple property.

Scenario

Foo holds a Map of people's ages. This map is keyed by the Name entity class. Name holds a person's name as a string property.

```
Map Foo.getAges() // of Person-String instances
```

Mapping

```
<class name="Foo" table="foo">
  ...
  <map role="ages">
     <key column="id"/>
     <index-many-to-many column="person_id"
         class="Person"/>
     <element column="age" type="string"/>
  </map>
</class>
<class name="Person" table="person">
  ...
  <property name="name" column="name"
      type="string"/>
```

Bidirectionality has no meaning for a map.

```
</class>
```

As for the normal map a simple extra table, *Ages*, is used to store the Person FK and age data.

Bidirectionality

Bidirectionality has no meaning for an entity map.

➤ Subclasses

Subclasses are classes that extend another class in a standard OO inheritance relationship.

Scenario

We have one class, Foo, and another class, Bar, which is a subclass of Foo.

```
public class Bar extends Foo
```

Mapping

```
<class name="Foo" table="foo"
```

Mapping

```
discriminator-value="F">
  ...
  <discriminator column="class"/>
  ...
  <subclass name="subclass.Bar"
          discriminator-value="B">
    <property name="name" column="name"
          type="string"/>
  </subclass>
</class>
```

The *class* field holds a discriminator value. This value tells Hibernate which Java class to instantiate on loading. The subclass, Bar, has its properties stored in the Foo table.

In this case we've used one table per class hierarchy. An alternative would be one table per concrete class. We could map that by simply including two class definitions and repeating the attrs of Foo in the Bar definition.

➤ Joined Subclass

Joined subclasses are those that are mapped to a table-per-subclass design rather than a table-per-hierarchy.

Scenario

We have one class, Foo, and another class, Bar, which is a subclass of Foo.

```
public class Bar extends Foo
```

Mapping

```
<class name="Foo" table="foo">
  ...
  <property name="name" column="name" type="string"/>
  <joined-subclass name="subclass.Bar" table="bar">
    <key column="foo_id"/>
    <property name="age" column="age" type="string"/>
  </joined-subclass>
</class>
```

Here Bar inherits from Foo and so is joined using the PK *foo_id* and adds the extra data column *age*.

Bidirectionality

Inheritance relationships are only unidirectional in Java. A child can determine its parent class but the reverse has no meaning.

➤ Components

Bidirectionality

Inheritance relationships are only unidirectional in Java. A child can determine its parent class but the reverse has no meaning.

Components are Java classes that are populated from selected columns of a parent class' table. This allows second-rank classes to exist within a class whilst still mapping to a single table for efficiency.

Scenario

We have one class, Foo, and a second-rank class, FooSecond.

```
FooSecond Foo.getSecond() // returns
    // enclosed second-rank instance
```

Mapping

```
<class name="Foo" table="foo">
  ...
  <component name="second" class="FooSecond"
      lazy="true|false">
      <property name="firstName"/>
      <property name="lastName"/>
  </component>
</class>
```

Here, Foo is mapped using *id* from table *Foo*. FooSecond is mapped from the same table using *firstName* and *lastName*.

➤ Collections of Components

Collections of components are also possible. These can be especially useful when dealing with a complex top-level collection.

Scenario

We have one class, Foo, and a top-level collection of second-rank class FooSecond which Foo holds.

```
Set Foo.getSeconds() // of FooSecond instances
```

Mapping

```
<set role="seconds">
  <key column="id" type="string">
    <generator class="uuid.hex"/>
  </key>
  <composite-element class="FooSecond">
    <property name="firstName"/>
    <property name="lastName"/>
  </composite-element>
```

➤ Composite Id

Composite Identifiers are PK identifiers for classes that consist of more than one column.

Scenario

Foo has a primary key that is of type Person. Person is made up of a String name and an Address type.

```
Person Foo.getId() // PK is multi-column mapped
```

Mapping

```
<class name="Foo" table="foo">
  <composite-id name="id" class="Person">
    <key-property name="name" type="string"
        column="name"/>
    <key-many-to-one name="address" class="Address"
    column="addr_id"/>
  </composite-id>
  <property name="age" column="age" type="string"/>
```

```
</set>
<class name="Foo" table="foo">
  ...
  <collection name="seconds" column="seconds_id"
    role="seconds"/>
</class>
```

This time we have two tables. Foo has a *seconds_id* FK column and Seconds appears as a collection table in its own right. The table structure is the same as for a standard top-level collection, but this time Seconds is being treated as a component object - a composite of two columns in the collection table - and set as a single property of Foo.

Bidirectionality

There's no bidirectional relationship available here as there is only one class involved.

```
</class>
<class name="Address" table="address">
  ...
</class>
```

Foo has a composite id of type Person. This is mapped to two columns: 1) Name, 2) a many-to-one relationship with the Address class.

Either *<key-property>* or *<key-many-to-one>* declarations may exist within a composite id.

So Foo has two PK-columns: *addr_id* and *name*. These are combined to initialise Person instances.

➤ Composite Index

A Composite Index is used to provide a multi-column Map key. The semantics are very similar to Composite Id.

Scenario

Foo contains a map of ages (strings). The map is keyed by Person, where Person is mapped as a String column (name) and Address instance.

```
Map Foo.getAges()
    // Foo contains a map of ages, keyed by Person
```

Mapping

```
<class name="Foo" table="foo">
  ...
  <map role="ages">
    <key column="id"/>
    <composite-index class="Person">
    <key-property name="name" type="string"
         column="name"/>
    <key-many-to-one name="address" class="Address"
        column="addr_id"/>
  </composite-index>
  <element column="age" type="string"/>
  </map>
</class>
<class name="Address" table="address">
...
```

```
</class>
```

Here the table *Ages* is created to hold the map. The key is made up of the columns *name* and *addr_id* which are used to map Person keys. The *age* field holds the map value. Finally, the map table needs its own id which is stored in the *id* field.

➤ Ternary Associations (and beyond)

Ternary associations are those that involve three classes. Classes A, B and C are associated together in a relationship.

Scenario

We have classes Foo, Bar and Snafu which are all related together. We choose to store their relationship in Foo using a composite class BarSnafu:

```
Bar BarSnafu.getBar()
            Snafu BarSnafu.getSnafu()
            Set Foo.getBarSnafus()
```

Mapping

```
<class name="Foo" table="foo">
  ...
  <set role="barsnafus" table="foo_bar_snafu">
    <key column="foo_id"/>
    <composite-element class="BarSnafu">
      <many-to-one name="bar" class="Bar"
           column="bar_id"/>
      <many-to-one name="snafu" class="Snafu"
           column="snafu_id"/>
    </composite-element>
  </set>
</class>
...
```

So here we've got three classes. They are related together in table *foo_bar_snafu*. The result is stored as a set of BarSnafu instances on each Foo. Each BarSnafu instance references one Bar and one Snafu.

Using composite elements we can go beyond this and support 4 and more elements in a relationship.