Hibernate Reference Guide

JBoss Enterprise Application Platform

4.3

JBoss®
a division of Red Hat

ISBN: N/A
Publication date: Sep, 2007

The JBoss Enterprise Application Platform Edition of the Hibernate Reference Guide 3.2

Hibernate Reference Guide: JBoss Enterprise Application Platform

The GPG fingerprint of the security@redhat.com key is:

CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E

1801 Varsity Drive
Raleigh, NC 27606-2072
USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588
Research Triangle Park, NC 27709
USA

Feedback

If you spot a typo in this guide, or if you have thought of a way to make this manual better, we would love to hear from you! Submit a report in *JIRA*[1] against the Product: JBoss Enterprise Application Platform, Version: *`<version>`*, Component: *Doc*. If you have a suggestion for improving the documentation, try to be as specific as possible. If you have found an error, include the section number and some of the surrounding text so we can find it easily.

---

[1] http://jira.jboss.com/jira/browse/JBPAPP

Preface

Working with object-oriented software and a relational database can be cumbersome and time consuming in today's enterprise environments. Hibernate is an object/relational mapping tool for Java environments. The term object/relational mapping (ORM) refers to the technique of mapping a data representation from an object model to a relational data model with a SQL-based schema.

Hibernate not only takes care of the mapping from Java classes to database tables (and from Java data types to SQL data types), but also provides data query and retrieval facilities and can significantly reduce development time otherwise spent with manual data handling in SQL and JDBC.

Hibernates goal is to relieve the developer from 95 percent of common data persistence related programming tasks. Hibernate may not be the best solution for data-centric applications that only use stored-procedures to implement the business logic in the database, it is most useful with object-oriented domain models and business logic in the Java-based middle-tier. However, Hibernate can certainly help you to remove or encapsulate vendor-specific SQL code and will help with the common task of result set translation from a tabular representation to a graph of objects.

If you are new to Hibernate and Object/Relational Mapping or even Java, please follow these steps:

1. Read *Chapter 2, Introduction to Hibernate* for a tutorial with step-by-step instructions. The source code for the tutorial is included in the distribution in the `doc/reference/tutorial/` directory.

2. Read *Chapter 3, Architecture* to understand the environments where Hibernate can be used.

3. Have a look at the `eg/` directory in the Hibernate distribution, it contains a simple standalone application. Copy your JDBC driver to the `lib/` directory and edit `etc/hibernate.properties`, specifying correct values for your database. From a command prompt in the distribution directory, type `ant eg` (using Ant), or under Windows, type `build eg`.

4. Use this reference documentation as your primary source of information. Consider reading *Hibernate in Action* (http://www.manning.com/bauer) if you need more help with application design or if you prefer a step-by-step tutorial. Also visit http://caveatemptor.hibernate.org and download the example application for Hibernate in Action.

5. FAQs are answered on the Hibernate website.

6. Third party demos, examples, and tutorials are linked on the Hibernate website.

7. The Community Area on the Hibernate website is a good resource for design patterns and various integration solutions (Tomcat, JBoss AS, Struts, EJB, etc.).

If you have questions, use the user forum linked on the Hibernate website. We also provide a

JIRA issue trackings system for bug reports and feature requests. If you are interested in the development of Hibernate, join the developer mailing list. If you are interested in translating this documentation into your language, contact us on the developer mailing list.

Commercial development support, production support, and training for Hibernate is available through JBoss Inc. (see http://www.hibernate.org/SupportTraining/). Hibernate is a Professional Open Source project and a critical component of the JBoss Enterprise Middleware System (JEMS) suite of products.

Introduction to Hibernate

1. Preface

This chapter is an introductory tutorial for new users of Hibernate. We start with a simple command line application using an in-memory database and develop it in easy to understand steps.

This tutorial is intended for new users of Hibernate but requires Java and SQL knowledge. It is based on a tutorial by Michael Gloegl, the third-party libraries we name are for JDK 1.4 and 5.0. You might need others for JDK 1.3.

The source code for the tutorial is included in the distribution in the `doc/reference/tutorial/` directory.

2. Part 1 - The first Hibernate Application

First, we'll create a simple console-based Hibernate application. We use an Java database (HSQL DB), so we do not have to install any database server.

Let's assume we need a small database application that can store events we want to attend, and information about the hosts of these events.

The first thing we do, is set up our development directory and put all the Java libraries we need into it. Download the Hibernate distribution from the Hibernate website. Extract the package and place all required libraries found in `/lib` into into the `/lib` directory of your new development working directory. It should look like this:

```
.
+lib
  antlr.jar
  cglib.jar
  asm.jar
  asm-attrs.jars
  commons-collections.jar
  commons-logging.jar
  hibernate3.jar
  jta.jar
  dom4j.jar
  log4j.jar
```

This is the minimum set of required libraries (note that we also copied hibernate3.jar, the main archive) for Hibernate *at the time of writing*. The Hibernate release you are using might require more or less libraries. See the `README.txt` file in the `lib/` directory of the Hibernate distribution for more information about required and optional third-party libraries. (Actually, Log4j is not required but preferred by many developers.)

Next we create a class that represents the event we want to store in database.

2.1. The first class

Our first persistent class is a simple JavaBean class with some properties:

```
package events;

import java.util.Date;

public class Event {
    private Long id;

    private String title;
    private Date date;

    public Event() {}

    public Long getId() {
        return id;
    }

    private void setId(Long id) {
        this.id = id;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

You can see that this class uses standard JavaBean naming conventions for property getter and setter methods, as well as private visibility for the fields. This is a recommended design - but not required. Hibernate can also access fields directly, the benefit of accessor methods is robustness for refactoring. The no-argument constructor is required to instantiate an object of this class through reflection.

The `id` property holds a unique identifier value for a particular event. All persistent entity classes (there are less important dependent classes as well) will need such an identifier property if we want to use the full feature set of Hibernate. In fact, most applications (esp. web applications) need to distinguish objects by identifier, so you should consider this a feature rather than a limitation. However, we usually don't manipulate the identity of an object, hence the setter method should be private. Only Hibernate will assign identifiers when an object is saved. You can see that Hibernate can access public, private, and protected accessor methods, as well as (public, private, protected) fields directly. The choice is up to you and you can match it to fit your application design.

The no-argument constructor is a requirement for all persistent classes; Hibernate has to create objects for you, using Java Reflection. The constructor can be private, however, package visibility is required for runtime proxy generation and efficient data retrieval without bytecode instrumentation.

Place this Java source file in a directory called `src` in the development folder, and in its correct package. The directory should now look like this:

```
.
+lib
  <Hibernate and third-party libraries>
+src
  +events
    Event.java
```

In the next step, we tell Hibernate about this persistent class.

2.2. The mapping file

Hibernate needs to know how to load and store objects of the persistent class. This is where the Hibernate mapping file comes into play. The mapping file tells Hibernate what table in the database it has to access, and what columns in that table it should use.

The basic structure of a mapping file looks like this:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
[...]
</hibernate-mapping>
```

Note that the Hibernate DTD is very sophisticated. You can use it for auto-completion of XML mapping elements and attributes in your editor or IDE. You also should open up the DTD file in your text editor - it's the easiest way to get an overview of all elements and attributes and to see the defaults, as well as some comments. Note that Hibernate will not load the DTD file from the web, but first look it up from the classpath of the application. The DTD file is included in `hibernate3.jar` as well as in the `src/` directory of the Hibernate distribution.

We will omit the DTD declaration in future examples to shorten the code. It is of course not optional.

Between the two `hibernate-mapping` tags, include a `class` element. All persistent entity classes (again, there might be dependent classes later on, which are not first-class entities) need such a mapping, to a table in the SQL database:

```
<hibernate-mapping>

    <class name="events.Event" table="EVENTS">

    </class>

</hibernate-mapping>
```

So far we told Hibernate how to persist and load object of class `Event` to the table `EVENTS`, each instance represented by a row in that table. Now we continue with a mapping of the unique identifier property to the tables primary key. In addition, as we don't want to care about handling this identifier, we configure Hibernate's identifier generation strategy for a surrogate primary key column:

```
<hibernate-mapping>

    <class name="events.Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
    </class>

</hibernate-mapping>
```

The `id` element is the declaration of the identifer property, `name="id"` declares the name of the Java property - Hibernate will use the getter and setter methods to access the property. The column attribute tells Hibernate which column of the `EVENTS` table we use for this primary key. The nested `generator` element specifies the identifier generation strategy, in this case we used `native`, which picks the best strategy depending on the configured database (dialect). Hibernate supports database generated, globally unique, as well as application assigned identifiers (or any strategy you have written an extension for).

Finally we include declarations for the persistent properties of the class in the mapping file. By default, no properties of the class are considered persistent:

```

<hibernate-mapping>

    <class name="events.Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
        <property name="date" type="timestamp" column="EVENT_DATE"/>
        <property name="title"/>
    </class>

</hibernate-mapping>
```

Just as with the `id` element, the `name` attribute of the `property` element tells Hibernate which getter and setter methods to use. So, in this case, Hibernate will look for `getDate()/setDate()`, as well as `getTitle()/setTitle()`.

Why does the `date` property mapping include the `column` attribute, but the `title` doesn't? Without the `column` attribute Hibernate by default uses the property name as the column name. This works fine for `title`. However, `date` is a reserved keyword in most database, so we better map it to a different name.

The next interesting thing is that the `title` mapping also lacks a `type` attribute. The types we declare and use in the mapping files are not, as you might expect, Java data types. They are also not SQL database types. These types are so called *Hibernate mapping types*, converters which can translate from Java to SQL data types and vice versa. Again, Hibernate will try to determine the correct conversion and mapping type itself if the `type` attribute is not present in the mapping. In some cases this automatic detection (using Reflection on the Java class) might not have the default you expect or need. This is the case with the `date` property. Hibernate can't know if the property (which is of `java.util.Date`) should map to a SQL `date`, `timestamp`, or `time` column. We preserve full date and time information by mapping the property with a `timestamp` converter.

This mapping file should be saved as `Event.hbm.xml`, right in the directory next to the `Event` Java class source file. The naming of mapping files can be arbitrary, however the `hbm.xml` suffix is a convention in the Hibernate developer community. The directory structure should now look like this:

```
.
+lib
  <Hibernate and third-party libraries>
+src
  +events
    Event.java
    Event.hbm.xml
```

We continue with the main configuration of Hibernate.

2.3. Hibernate configuration

We now have a persistent class and its mapping file in place. It is time to configure Hibernate. Before we do this, we will need a database. HSQL DB, a java-based SQL DBMS, can be downloaded from the HSQL DB website. Actually, you only need the `hsqldb.jar` from this download. Place this file in the `lib/` directory of the development folder.

Create a directory called `data` in the root of the development directory - this is where HSQL DB will store its data files. Now start the database by running `java -classpath ../lib/hsqldb.jar org.hsqldb.Server` in this data directory. You can see it start up and bind to a TCP/IP socket, this is where our application will connect later. If you want to start with a fresh database during this tutorial, shutdown HSQL DB (press `CTRL + C` in the window), delete all files in the `data/` directory, and start HSQL DB again.

Hibernate is the layer in your application which connects to this database, so it needs connection information. The connections are made through a JDBC connection pool, which we also have to configure. The Hibernate distribution contains several open source JDBC connection pooling tools, but will use the Hibernate built-in connection pool for this tutorial. Note that you have to copy the required library into your classpath and use different connection pooling settings if you want to use a production-quality third party JDBC pooling software.

For Hibernate's configuration, we can use a simple `hibernate.properties` file, a slightly more sophisticated `hibernate.cfg.xml` file, or even complete programmatic setup. Most users prefer the XML configuration file:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <!-- Database connection settings -->
        <property
name="connection.driver_class">org.hsqldb.jdbcDriver</property>
        <property
name="connection.url">jdbc:hsqldb:hsql://localhost</property>
        <property name="connection.username">sa</property>
        <property name="connection.password"></property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <property
name="dialect">org.hibernate.dialect.HSQLDialect</property>

        <!-- Enable Hibernate's automatic session context management -->
        <property name="current_session_context_class">thread</property>

        <!-- Disable the second-level cache  -->
        <property
name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">create</property>

        <mapping resource="events/Event.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

Note that this XML configuration uses a different DTD. We configure Hibernate's `SessionFactory` - a global factory responsible for a particular database. If you have several databases, use several `<session-factory>` configurations, usually in several configuration files (for easier startup).

The first four `property` elements contain the necessary configuration for the JDBC connection. The dialect `property` element specifies the particular SQL variant Hibernate generates. Hibernate's automatic session management for persistence contexts will come in handy as you will soon see. The `hbm2ddl.auto` option turns on automatic generation of database schemas - directly into the database. This can of course also be turned off (by removing the config option) or redirected to a file with the help of the `SchemaExport` Ant task. Finally, we add the mapping file(s) for persistent classes to the configuration.

Copy this file into the source directory, so it will end up in the root of the classpath. Hibernate automatically looks for a file called `hibernate.cfg.xml` in the root of the classpath, on startup.

2.4. Building with Ant

We'll now build the tutorial with Ant. You will need to have Ant installed - get it from the *Ant download page*[1]. How to install Ant will not be covered here. Please refer to the *Ant manual*[2]. After you have installed Ant, we can start to create the buildfile. It will be called `build.xml` and placed directly in the development directory.

A basic build file looks like this:

```
<project name="hibernate-tutorial" default="compile">

    <property name="sourcedir" value="${basedir}/src"/>
    <property name="targetdir" value="${basedir}/bin"/>
    <property name="librarydir" value="${basedir}/lib"/>

    <path id="libraries">
        <fileset dir="${librarydir}">
            <include name="*.jar"/>
        </fileset>
    </path>

    <target name="clean">
        <delete dir="${targetdir}"/>
        <mkdir dir="${targetdir}"/>
    </target>

    <target name="compile" depends="clean, copy-resources">
      <javac srcdir="${sourcedir}"
             destdir="${targetdir}"
             classpathref="libraries"/>
    </target>

    <target name="copy-resources">
```

---

[1] http://ant.apache.org/bindownload.cgi

[2] http://ant.apache.org/manual/index.html

```
        <copy todir="${targetdir}">
            <fileset dir="${sourcedir}">
                <exclude name="**/*.java"/>
            </fileset>
        </copy>
    </target>

</project>
```

This will tell Ant to add all files in the lib directory ending with `.jar` to the classpath used for compilation. It will also copy all non-Java source files to the target directory, e.g. configuration and Hibernate mapping files. If you now run Ant, you should get this output:

```
C:\hibernateTutorial\>ant
Buildfile: build.xml

copy-resources:
     [copy] Copying 2 files to C:\hibernateTutorial\bin

compile:
    [javac] Compiling 1 source file to C:\hibernateTutorial\bin

BUILD SUCCESSFUL
Total time: 1 second
```

2.5. Startup and helpers

It's time to load and store some `Event` objects, but first we have to complete the setup with some infrastructure code. We have to startup Hibernate. This startup includes building a global `SessionFactory` object and to store it somewhere for easy access in application code. A `SessionFactory` can open up new `Session`'s. A `Session` represents a single-threaded unit of work, the `SessionFactory` is a thread-safe global object, instantiated once.

We'll create a `HibernateUtil` helper class which takes care of startup and makes accessing a `SessionFactory` convenient. Let's have a look at the implementation:

```
package util;

import org.hibernate.*;
import org.hibernate.cfg.*;

public class HibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            sessionFactory = new
Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            // Make sure you log the exception, as it might be swallowed
            System.err.println("Initial SessionFactory creation failed." +
ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

}
```

This class does not only produce the global `SessionFactory` in its static initializer (called once by the JVM when the class is loaded), but also hides the fact that it uses a static singleton. It might as well lookup the `SessionFactory` from JNDI in an application server.

If you give the `SessionFactory` a name in your configuration file, Hibernate will in fact try to bind it to JNDI after it has been built. To avoid this code completely you could also use JMX deployment and let the JMX-capable container instantiate and bind a `HibernateService` to JNDI. These advanced options are discussed in the Hibernate reference documentation.

Place `HibernateUtil.java` in the development source directory, in a package next to `events`:

```
.
+lib
  <Hibernate and third-party libraries>
+src
  +events
    Event.java
    Event.hbm.xml
  +util
    HibernateUtil.java
  hibernate.cfg.xml
+data
build.xml
```

This should again compile without problems. We finally need to configure a logging system - Hibernate uses commons logging and leaves you the choice between Log4j and JDK 1.4 logging. Most developers prefer Log4j: copy `log4j.properties` from the Hibernate distribution (it's in the `etc/` directory) to your `src` directory, next to `hibernate.cfg.xml`. Have a look at the example configuration and change the settings if you like to have more verbose output. By default, only Hibernate startup message are shown on stdout.

The tutorial infrastructure is complete - and we are ready to do some real work with Hibernate.

2.6. Loading and storing objects

Finally, we can use Hibernate to load and store objects. We write an `EventManager` class with a `main()` method:

```
package events;
import org.hibernate.Session;

import java.util.Date;

import util.HibernateUtil;

public class EventManager {

    public static void main(String[] args) {
        EventManager mgr = new EventManager();

        if (args[0].equals("store")) {
            mgr.createAndStoreEvent("My Event", new Date());
        }

        HibernateUtil.getSessionFactory().close();
    }

    private void createAndStoreEvent(String title, Date theDate) {

        Session session =
HibernateUtil.getSessionFactory().getCurrentSession();

        session.beginTransaction();

        Event theEvent = new Event();
        theEvent.setTitle(title);
        theEvent.setDate(theDate);

        session.save(theEvent);

        session.getTransaction().commit();
    }

}
```

We create a new `Event` object, and hand it over to Hibernate. Hibernate now takes care of the SQL and executes `INSERT` s on the database. Let's have a look at the `Session` and `Transaction`-handling code before we run this.

A `Session` is a single unit of work. For now we'll keep things simple and assume a one-to-one granularity between a Hibernate `Session` and a database transaction. To shield our code from the actual underlying transaction system (in this case plain JDBC, but it could also run with JTA) we use the `Transaction` API that is available on the Hibernate `Session`.

What does `sessionFactory.getCurrentSession()` do? First, you can call it as many times and anywhere you like, once you get hold of your `SessionFactory` (easy thanks to `HibernateUtil`). The `getCurrentSession()` method always returns the "current" unit of work. Remember that we switched the configuration option for this mechanism to "thread" in `hibernate.cfg.xml`? Hence, the current unit of work is bound to the current Java thread that executes our application. However, this is not the full picture, you also have to consider scope, when a unit of work begins and when it ends.

A `Session` begins when it is first needed, when the first call to `getCurrentSession()` is made. It is then bound by Hibernate to the current thread. When the transaction ends, either through commit or rollback, Hibernate automatically unbinds the `Session` from the thread and closes it for you. If you call `getCurrentSession()` again, you get a new `Session` and can start a new unit of work. This *thread-bound* programming model is the most popular way of using Hibernate, as it allows flexible layering of your code (transaction demarcation code can be separated from data access code, we'll do this later in this tutorial).

Related to the unit of work scope, should the Hibernate `Session` be used to execute one or several database operations? The above example uses one `Session` for one operation. This is pure coincidence, the example is just not complex enough to show any other approach. The scope of a Hibernate `Session` is flexible but you should never design your application to use a new Hibernate `Session` for *every* database operation. So even if you see it a few more times in the following (very trivial) examples, consider *session-per-operation* an anti-pattern. A real (web) application is shown later in this tutorial.

Have a look at *Chapter 12, Transactions And Concurrency* for more information about transaction handling and demarcation. We also skipped any error handling and rollback in the previous example.

To run this first routine we have to add a callable target to the Ant build file:

```
<target name="run" depends="compile">
    <java fork="true" classname="events.EventManager"
classpathref="libraries">
        <classpath path="${targetdir}"/>
        <arg value="${action}"/>
    </java>
</target>
```

The value of the `action` argument is set on the command line when calling the target:

```
C:\hibernateTutorial\>ant run -Daction=store
```

You should see, after compilation, Hibernate starting up and, depending on your configuration, lots of log output. At the end you will find the following line:

```
[java] Hibernate: insert into EVENTS (EVENT_DATE, title, EVENT_ID) values
(?, ?, ?)
```

This is the `INSERT` executed by Hibernate, the question marks represent JDBC bind parameters. To see the values bound as arguments, or to reduce the verbosity of the log, check your `log4j.properties`.

Now we'd like to list stored events as well, so we add an option to the main method:

```
if (args[0].equals("store")) {
    mgr.createAndStoreEvent("My Event", new Date());
}
else if (args[0].equals("list")) {
    List events = mgr.listEvents();
    for (int i = 0; i < events.size(); i++) {
        Event theEvent = (Event) events.get(i);
        System.out.println("Event: " + theEvent.getTitle() +
                           " Time: " + theEvent.getDate());
    }
}
```

We also add a new `listEvents() method`:

```
private List listEvents() {

    Session session = HibernateUtil.getSessionFactory().getCurrentSession();

    session.beginTransaction();

    List result = session.createQuery("from Event").list();

    session.getTransaction().commit();

    return result;
}
```

What we do here is use an HQL (Hibernate Query Language) query to load all existing `Event` objects from the database. Hibernate will generate the appropriate SQL, send it to the database and populate `Event` objects with the data. You can create more complex queries with HQL, of course.

Now, to execute and test all of this, follow these steps:

- Run `ant run -Daction=store` to store something into the database and, of course, to generate the database schema before through hbm2ddl.
- Now disable hbm2ddl by commenting out the property in your `hibernate.cfg.xml` file. Usually you only leave it turned on in continous unit testing, but another run of hbm2ddl would *drop* everything you have stored - the `create` configuration setting actually translates into "drop all tables from the schema, then re-create all tables, when the SessionFactory is build".

If you now call Ant with `-Daction=list`, you should see the events you have stored so far. You can of course also call the `store` action a few times more.

Note: Most new Hibernate users fail at this point and we see questions about *Table not found* error messages regularly. However, if you follow the steps outlined above you will not have this problem, as hbm2ddl creates the database schema on the first run, and subsequent application restarts will use this schema. If you change the mapping and/or database schema, you have to re-enable hbm2ddl once again.

3. Part 2 - Mapping associations

We mapped a persistent entity class to a table. Let's build on this and add some class associations. First we'll add people to our application, and store a list of events they participate in.

3.1. Mapping the Person class

The first cut of the `Person` class is simple:

```
package events;

public class Person {

    private Long id;
    private int age;
    private String firstname;
    private String lastname;

    public Person() {}

    // Accessor methods for all properties, private setter for 'id'

}
```

Create a new mapping file called `Person.hbm.xml` (don't forget the DTD reference at the top):

```
<hibernate-mapping>

    <class name="events.Person" table="PERSON">
        <id name="id" column="PERSON_ID">
            <generator class="native"/>
        </id>
        <property name="age"/>
        <property name="firstname"/>
        <property name="lastname"/>
    </class>

</hibernate-mapping>
```

Finally, add the new mapping to Hibernate's configuration:

```
<mapping resource="events/Event.hbm.xml"/>
<mapping resource="events/Person.hbm.xml"/>
```

We'll now create an association between these two entities. Obviously, persons can participate in events, and events have participants. The design questions we have to deal with are:

directionality, multiplicity, and collection behavior.

3.2. A unidirectional Set-based association

We'll add a collection of events to the `Person` class. That way we can easily navigate to the events for a particular person, without executing an explicit query - by calling `aPerson.getEvents()`. We use a Java collection, a `Set`, because the collection will not contain duplicate elements and the ordering is not relevant for us.

We need a unidirectional, many-valued associations, implemented with a `Set`. Let's write the code for this in the Java classes and then map it:

```
public class Person {

    private Set events = new HashSet();

    public Set getEvents() {
        return events;
    }

    public void setEvents(Set events) {
        this.events = events;
    }
}
```

Before we map this association, think about the other side. Clearly, we could just keep this unidirectional. Or, we could create another collection on the `Event`, if we want to be able to navigate it bi-directional, i.e. `anEvent.getParticipants()`. This is not necessary, from a functional perspective. You could always execute an explicit query to retrieve the participants for a particular event. This is a design choice left to you, but what is clear from this discussion is the multiplicity of the association: "many" valued on both sides, we call this a *many-to-many* association. Hence, we use Hibernate's many-to-many mapping:

```
<class name="events.Person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="native"/>
    </id>
    <property name="age"/>
    <property name="firstname"/>
    <property name="lastname"/>

    <set name="events" table="PERSON_EVENT">
        <key column="PERSON_ID"/>
        <many-to-many column="EVENT_ID" class="events.Event"/>
    </set>

</class>
```

Hibernate supports all kinds of collection mappings, a `<set>` being most common. For a many-to-many association (or *n:m* entity relationship), an association table is needed. Each row in this table represents a link between a person and an event. The table name is configured with the `table` attribute of the `set` element. The identifier column name in the association, for the person's side, is defined with the `<key>` element, the column name for the event's side with the `column` attribute of the `<many-to-many>`. You also have to tell Hibernate the class of the objects in your collection (correct: the class on the other side of the collection of references).

The database schema for this mapping is therefore:

```
    _____________        __________________
   |             |      |                  |       _____________
   |   EVENTS    |      |   PERSON_EVENT   |      |             |
   |_____________|      |__________________|      |    PERSON   |
   |             |      |                  |      |_____________|
   | *EVENT_ID   | <--> | *EVENT_ID        |      |             |
   |  EVENT_DATE |      | *PERSON_ID       | <--> | *PERSON_ID  |
   |  TITLE      |      |__________________|      |  AGE        |
   |_____________|                                |  FIRSTNAME  |
                                                  |  LASTNAME   |
                                                  |_____________|
```

3.3. Working the association

Let's bring some people and events together in a new method in `EventManager`:

```
private void addPersonToEvent(Long personId, Long eventId) {

    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session.load(Person.class, personId);
    Event anEvent = (Event) session.load(Event.class, eventId);

    aPerson.getEvents().add(anEvent);

    session.getTransaction().commit();
}
```

After loading a `Person` and an `Event`, simply modify the collection using the normal collection methods. As you can see, there is no explicit call to `update()` or `save()`, Hibernate automatically detects that the collection has been modified and needs to be updated. This is called *automatic dirty checking*, and you can also try it by modifying the name or the date property of any of your objects. As long as they are in *persistent* state, that is, bound to a particular Hibernate `Session` (i.e. they have been just loaded or saved in a unit of work), Hibernate monitors any changes and executes SQL in a write-behind fashion. The process of synchronizing the memory state with the database, usually only at the end of a unit of work, is called *flushing*. In our code, the unit of work ends with a commit (or rollback) of the database transaction - as defined by the `thread` configuration option for the `CurrentSessionContext` class.

You might of course load person and event in different units of work. Or you modify an object outside of a `Session`, when it is not in persistent state (if it was persistent before, we call this state *detached*). You can even modify a collection when it is detached:

```
private void addPersonToEvent(Long personId, Long eventId) {

    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session
            .createQuery("select p from Person p left join fetch p.events
where p.id = :pid")
            .setParameter("pid", personId)
            .uniqueResult(); // Eager fetch the collection so we can use it
detached

    Event anEvent = (Event) session.load(Event.class, eventId);

    session.getTransaction().commit();

    // End of first unit of work

    aPerson.getEvents().add(anEvent); // aPerson (and its collection) is
detached

    // Begin second unit of work

    Session session2 =
HibernateUtil.getSessionFactory().getCurrentSession();
    session2.beginTransaction();

    session2.update(aPerson); // Reattachment of aPerson

    session2.getTransaction().commit();
}
```

The call to `update` makes a detached object persistent again, you could say it binds it to a new unit of work, so any modifications you made to it while detached can be saved to the database. This includes any modifications (additions/deletions) you made to a collection of that entity object.

Well, this is not much use in our current situation, but it's an important concept you can design into your own application. For now, complete this exercise by adding a new action to the `EventManager`'s main method and call it from the command line. If you need the identifiers of a person and an event - the `save()` method returns it (you might have to modify some of the previous methods to return that identifier):

```
else if (args[0].equals("addpersontoevent")) {
    Long eventId = mgr.createAndStoreEvent("My Event", new Date());
    Long personId = mgr.createAndStorePerson("Foo", "Bar");
    mgr.addPersonToEvent(personId, eventId);
    System.out.println("Added person " + personId + " to event " + eventId);
}
```

This was an example of an association between two equally important classes, two entities. As mentioned earlier, there are other classes and types in a typical model, usually "less important". Some you have already seen, like an `int` or a `String`. We call these classes *value types*, and their instances *depend* on a particular entity. Instances of these types don't have their own identity, nor are they shared between entities (two persons don't reference the same `firstname` object, even if they have the same first name). Of course, value types can not only be found in the JDK (in fact, in a Hibernate application all JDK classes are considered value types), but you can also write dependent classes yourself, `Address` or `MonetaryAmount`, for example.

You can also design a collection of value types. This is conceptually very different from a collection of references to other entities, but looks almost the same in Java.

3.4. Collection of values

We add a collection of value typed objects to the `Person` entity. We want to store email addresses, so the type we use is `String`, and the collection is again a `Set`:

```
private Set emailAddresses = new HashSet();

public Set getEmailAddresses() {
    return emailAddresses;
}

public void setEmailAddresses(Set emailAddresses) {
    this.emailAddresses = emailAddresses;
}
```

The mapping of this `Set`:

```
<set name="emailAddresses" table="PERSON_EMAIL_ADDR">
    <key column="PERSON_ID"/>
    <element type="string" column="EMAIL_ADDR"/>
</set>
```

The difference compared with the earlier mapping is the `element` part, which tells Hibernate that the collection does not contain references to another entity, but a collection of elements of type `String` (the lowercase name tells you it's a Hibernate mapping type/converter). Once again, the `table` attribute of the `set` element determines the table name for the collection. The `key` element defines the foreign-key column name in the collection table. The `column` attribute in the `element` element defines the column name where the `String` values will actually be stored.

Have a look at the updated schema:

```

  _____________        __________________
 |             |      |                  |       _____________
 |   EVENTS    |      |   PERSON_EVENT   |      |             |
___________________
 |_____________|      |__________________|      |    PERSON   |      |
|
 |             |      |                  |      |_____________|      |
PERSON_EMAIL_ADDR |
 | *EVENT_ID   | <--> | *EVENT_ID        |      |             |
|___________________|
 |  EVENT_DATE |      | *PERSON_ID       | <--> | *PERSON_ID  | <--> |
*PERSON_ID       |
 |  TITLE      |      |__________________|      |  AGE        |      |
*EMAIL_ADDR      |
 |_____________|                                |  FIRSTNAME  |
|___________________|
                                                |  LASTNAME   |
                                                |_____________|
```

You can see that the primary key of the collection table is in fact a composite key, using both columns. This also implies that there can't be duplicate email addresses per person, which is exactly the semantics we need for a set in Java.

You can now try and add elements to this collection, just like we did before by linking persons and events. It's the same code in Java:

```
private void addEmailToPerson(Long personId, String emailAddress) {

    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session.load(Person.class, personId);

    // The getEmailAddresses() might trigger a lazy load of the collection
    aPerson.getEmailAddresses().add(emailAddress);

    session.getTransaction().commit();
}
```

This time we didnt' use a *fetch* query to initialize the collection. Hence, the call to its getter method will trigger an additional select to initialize it, so we can add an element to it. Monitor the SQL log and try to optimize this with an eager fetch.

3.5. Bi-directional associations

Next we are going to map a bi-directional association - making the association between person and event work from both sides in Java. Of course, the database schema doesn't change, we still have many-to-many multiplicity. A relational database is more flexible than a network programming language, so it doesn't need anything like a navigation direction - data can be viewed and retrieved in any possible way.

First, add a collection of participants to the `Event` Event class:

```
private Set participants = new HashSet();

public Set getParticipants() {
    return participants;
}

public void setParticipants(Set participants) {
    this.participants = participants;
}
```

Now map this side of the association too, in `Event.hbm.xml`.

```
<set name="participants" table="PERSON_EVENT" inverse="true">
    <key column="EVENT_ID"/>
    <many-to-many column="PERSON_ID" class="events.Person"/>
</set>
```

As you see, these are normal `set` mappings in both mapping documents. Notice that the column names in `key` and `many-to-many` are swapped in both mapping documents. The most important addition here is the `inverse="true"` attribute in the `set` element of the `Event`'s collection mapping.

What this means is that Hibernate should take the other side - the `Person` class - when it needs to find out information about the link between the two. This will be a lot easier to understand once you see how the bi-directional link between our two entities is created .

3.6. Working bi-directional links

First, keep in mind that Hibernate does not affect normal Java semantics. How did we create a link between a `Person` and an `Event` in the unidirectional example? We added an instance of `Event` to the collection of event references, of an instance of `Person`. So, obviously, if we want to make this link working bi-directional, we have to do the same on the other side - adding a `Person` reference to the collection in an `Event`. This "setting the link on both sides" is absolutely necessary and you should never forget doing it.

Many developers program defensive and create a link management methods to correctly set both sides, e.g. in `Person`:

```
protected Set getEvents() {
    return events;
}

protected void setEvents(Set events) {
    this.events = events;
}

public void addToEvent(Event event) {
    this.getEvents().add(event);
```

```
        event.getParticipants().add(this);
    }

    public void removeFromEvent(Event event) {
        this.getEvents().remove(event);
        event.getParticipants().remove(this);
    }
```

Notice that the get and set methods for the collection are now protected - this allows classes in the same package and subclasses to still access the methods, but prevents everybody else from messing with the collections directly (well, almost). You should probably do the same with the collection on the other side.

What about the `inverse` mapping attribute? For you, and for Java, a bi-directional link is simply a matter of setting the references on both sides correctly. Hibernate however doesn't have enough information to correctly arrange SQL `INSERT` and `UPDATE` statements (to avoid constraint violations), and needs some help to handle bi-directional associations properly. Making one side of the association `inverse` tells Hibernate to basically ignore it, to consider it a *mirror* of the other side. That's all that is necessary for Hibernate to work out all of the issues when transformation a directional navigation model to a SQL database schema. The rules you have to remember are straightforward: All bi-directional associations need one side as `inverse`. In a one-to-many association it has to be the many-side, in many-to-many association you can pick either side, there is no difference.

Let's turn this into a small web application.

4. Part 3 - The EventManager web application

A Hibernate web application uses `Session` and `Transaction` almost like a standalone application. However, some common patterns are useful. We now write an `EventManagerServlet`. This servlet can list all events stored in the database, and it provides an HTML form to enter new events.

4.1. Writing the basic servlet

Create a new class in your source directory, in the `events` package:

```
package events;

// Imports

public class EventManagerServlet extends HttpServlet {

    // Servlet code
}
```

The servlet handles HTTP `GET` requests only, hence, the method we implement is `doGet()`:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response)
        throws ServletException, IOException {

    SimpleDateFormat dateFormatter = new SimpleDateFormat("dd.MM.yyyy");

    try {
        // Begin unit of work
        HibernateUtil.getSessionFactory()
                .getCurrentSession().beginTransaction();

        // Process request and render page...

        // End unit of work
        HibernateUtil.getSessionFactory()
                .getCurrentSession().getTransaction().commit();

    } catch (Exception ex) {
        HibernateUtil.getSessionFactory()
                .getCurrentSession().getTransaction().rollback();
        throw new ServletException(ex);
    }

}
```

The pattern we are applying here is called *session-per-request*. When a request hits the servlet, a new Hibernate `Session` is opened through the first call to `getCurrentSession()` on the `SessionFactory`. Then a database transaction is started - all data access as to occur inside a transaction, no matter if data is read or written (we don't use the auto-commit mode in applications).

Do *not* use a new Hibernate `Session` for every database operation. Use one Hibernate `Session` that is scoped to the whole request. Use `getCurrentSession()`, so that it is automatically bound to the current Java thread.

Next, the possible actions of the request are processed and the response HTML is rendered. We'll get to that part soon.

Finally, the unit of work ends when processing and rendering is complete. If any problem occured during processing or rendering, an exception will be thrown and the database transaction rolled back. This completes the `session-per-request` pattern. Instead of the transaction demarcation code in every servlet you could also write a servlet filter. See the Hibernate website and Wiki for more information about this pattern, called *Open Session in View* - you'll need it as soon as you consider rendering your view in JSP, not in a servlet.

4.2. Processing and rendering

Let's implement the processing of the request and rendering of the page.

```
// Write HTML header
PrintWriter out = response.getWriter();
```

```
out.println("<html><head><title>Event Manager</title></head><body>");

// Handle actions
if ( "store".equals(request.getParameter("action")) ) {

    String eventTitle = request.getParameter("eventTitle");
    String eventDate = request.getParameter("eventDate");

    if ( "".equals(eventTitle) || "".equals(eventDate) ) {
        out.println("<b><i>Please enter event title and date.</i></b>");
    } else {
        createAndStoreEvent(eventTitle, dateFormatter.parse(eventDate));
        out.println("<b><i>Added event.</i></b>");
    }
}

// Print page
printEventForm(out);
listEvents(out, dateFormatter);

// Write HTML footer
out.println("</body></html>");
out.flush();
out.close();
```

Granted, this coding style with a mix of Java and HTML would not scale in a more complex application - keep in mind that we are only illustrating basic Hibernate concepts in this tutorial. The code prints an HTML header and a footer. Inside this page, an HTML form for event entry and a list of all events in the database are printed. The first method is trivial and only outputs HTML:

```
private void printEventForm(PrintWriter out) {
    out.println("<h2>Add new event:</h2>");
    out.println("<form>");
    out.println("Title: <input name='eventTitle' length='50'/><br/>");
    out.println("Date (e.g. 24.12.2009): <input name='eventDate'
length='10'/><br/>");
    out.println("<input type='submit' name='action' value='store'/>");
    out.println("</form>");
}
```

The `listEvents()` method uses the Hibernate `Session` bound to the current thread to execute a query:

```
private void listEvents(PrintWriter out, SimpleDateFormat dateFormatter) {

    List result = HibernateUtil.getSessionFactory()
                    .getCurrentSession().createCriteria(Event.class).list();
    if (result.size() > 0) {
        out.println("<h2>Events in database:</h2>");
        out.println("<table border='1'>");
        out.println("<tr>");
        out.println("<th>Event title</th>");
        out.println("<th>Event date</th>");
        out.println("</tr>");
        for (Iterator it = result.iterator(); it.hasNext();) {
            Event event = (Event) it.next();
            out.println("<tr>");
            out.println("<td>" + event.getTitle() + "</td>");
            out.println("<td>" + dateFormatter.format(event.getDate()) +
"</td>");
            out.println("</tr>");
        }
        out.println("</table>");
    }
}
```

Finally, the `store` action is dispatched to the `createAndStoreEvent()` method, which also uses the `Session` of the current thread:

```
protected void createAndStoreEvent(String title, Date theDate) {
    Event theEvent = new Event();
    theEvent.setTitle(title);
    theEvent.setDate(theDate);

    HibernateUtil.getSessionFactory()
                    .getCurrentSession().save(theEvent);
}
```

That's it, the servlet is complete. A request to the servlet will be processed in a single `Session` and `Transaction`. As earlier in the standalone application, Hibernate can automatically bind these ojects to the current thread of execution. This gives you the freedom to layer your code and access the `SessionFactory` in any way you like. Usually you'd use a more sophisticated design and move the data access code into data access objects (the DAO pattern). See the Hibernate Wiki for more examples.

4.3. Deploying and testing

To deploy this application you have to create a web archive, a WAR. Add the following Ant target to your `build.xml`:

```
<target name="war" depends="compile">
    <war destfile="hibernate-tutorial.war" webxml="web.xml">
        <lib dir="${librarydir}">
          <exclude name="jsdk*.jar"/>
        </lib>

        <classes dir="${targetdir}"/>
    </war>
</target>
```

This target creates a file called `hibernate-tutorial.war` in your project directory. It packages all libraries and the `web.xml` descriptor, which is expected in the base directory of your project:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <servlet>
        <servlet-name>Event Manager</servlet-name>
        <servlet-class>events.EventManagerServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>Event Manager</servlet-name>
        <url-pattern>/eventmanager</url-pattern>
    </servlet-mapping>
</web-app>
```

Before you compile and deploy the web application, note that an additional library is required: `jsdk.jar`. This is the Java servlet development kit, if you don't have this library already, get it from the Sun website and copy it to your library directory. However, it will be only used for compliation and excluded from the WAR package.

To build and deploy call `ant war` in your project directory and copy the `hibernate-tutorial.war` file into your Tomcat `webapp` directory. If you don't have Tomcat installed, download it and follow the installation instructions. You don't have to change any Tomcat configuration to deploy this application though.

Once deployed and Tomcat is running, access the application at `http://localhost:8080/hibernate-tutorial/eventmanager`. Make sure you watch the Tomcat log to see Hibernate initialize when the first request hits your servlet (the static initializer in `HibernateUtil` is called) and to get the detailed output if any exceptions occurs.

5. Summary

This tutorial covered the basics of writing a simple standalone Hibernate application and a small web application.

If you already feel confident with Hibernate, continue browsing through the reference documentation table of contents for topics you find interesting - most asked are transactional processing (*Chapter 12, Transactions And Concurrency*), fetch performance (*Chapter 20, Improving performance*), or the usage of the API (*Chapter 11, Working with objects*) and the query features (*Section 4, “Querying”*).

Don't forget to check the Hibernate website for more (specialized) tutorials.

Architecture

1. Overview

A (very) high-level view of the Hibernate architecture:

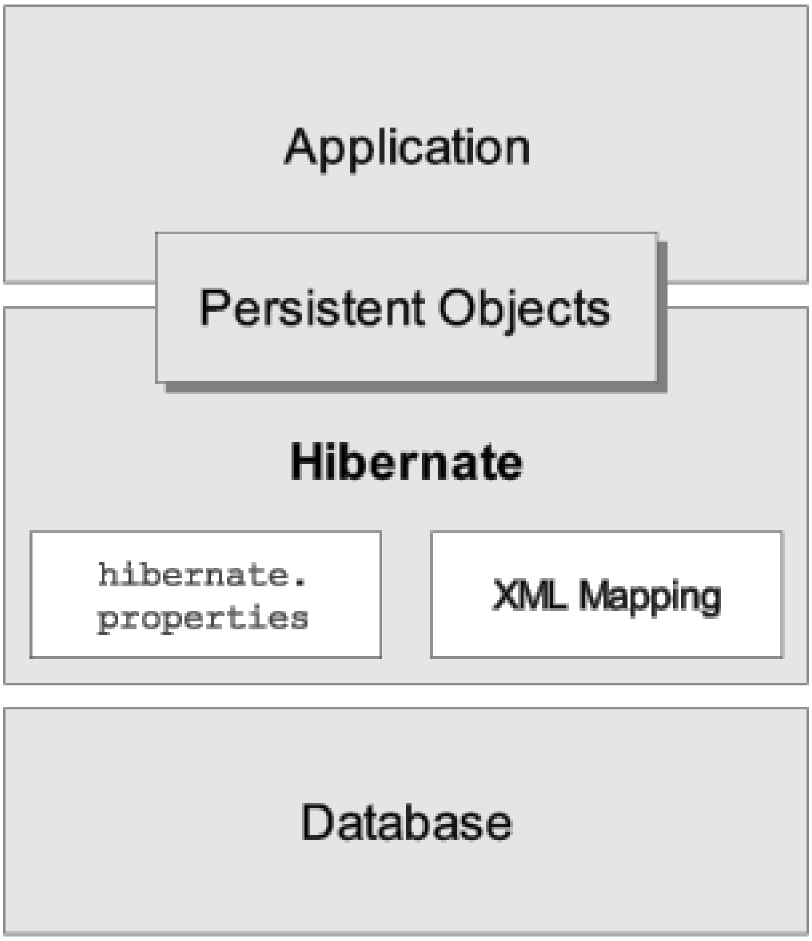


Figure 3.1. High Level view of the Hibernate Architecture

This diagram shows Hibernate using the database and configuration data to provide persistence services (and persistent objects) to the application.

We would like to show a more detailed view of the runtime architecture. Unfortunately, Hibernate is flexible and supports several approaches. We will show the two extremes. The "lite" architecture has the application provide its own JDBC connections and manage its own transactions. This approach uses a minimal subset of Hibernate's APIs:

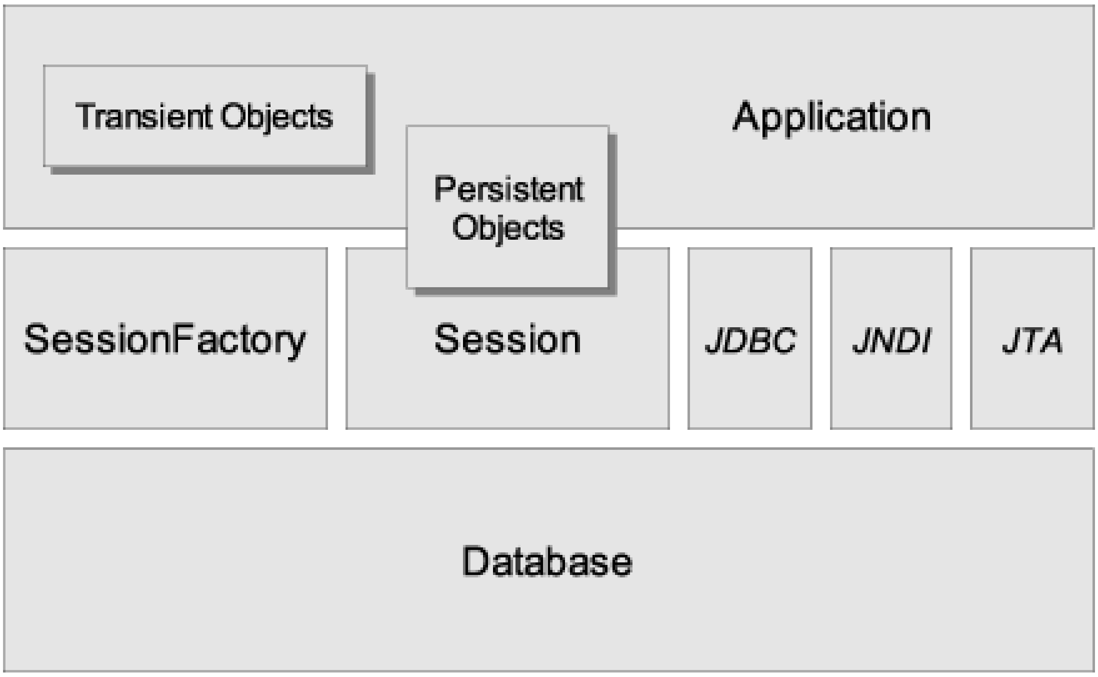
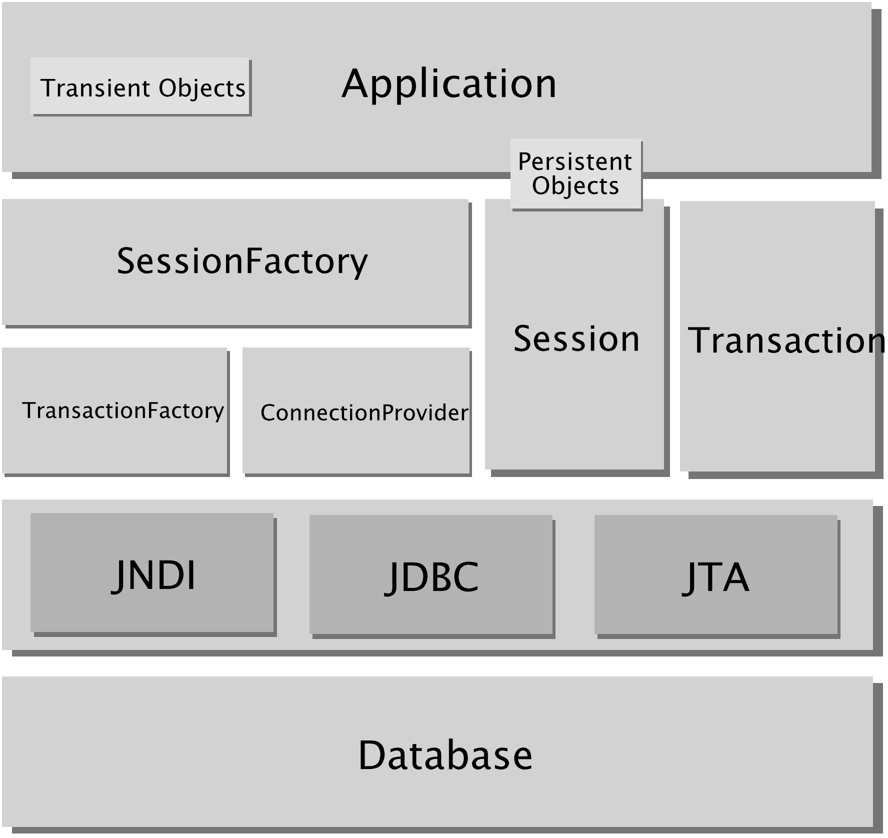


Figure 3.2. The Lite Architecture

The "full cream" architecture abstracts the application away from the underlying JDBC/JTA APIs and lets Hibernate take care of the details.



Heres some definitions of the objects in the diagrams:

SessionFactory (`org.hibernate.SessionFactory`)
: A threadsafe (immutable) cache of compiled mappings for a single database. A factory for `Session` and a client of `ConnectionProvider`. Might hold an optional (second-level) cache of data that is reusable between transactions, at a process- or cluster-level.

Session (`org.hibernate.Session`)
: A single-threaded, short-lived object representing a conversation between the application and the persistent store. Wraps a JDBC connection. Factory for `Transaction`. Holds a mandatory (first-level) cache of persistent objects, used when navigating the object graph or looking up objects by identifier.

Persistent objects and collections
: Short-lived, single threaded objects containing persistent state and business function. These might be ordinary JavaBeans/POJOs, the only special thing about them is that they are currently associated with (exactly one) `Session`. As soon as the `Session` is closed, they will be detached and free to use in any application layer (e.g. directly as data transfer objects to and from presentation).

Transient and detached objects and collections
: Instances of persistent classes that are not currently associated with a `Session`. They may

have been instantiated by the application and not (yet) persisted or they may have been instantiated by a closed `Session`.

Transaction (`org.hibernate.Transaction`)

(Optional) A single-threaded, short-lived object used by the application to specify atomic units of work. Abstracts application from underlying JDBC, JTA or CORBA transaction. A `Session` might span several `Transaction` s in some cases. However, transaction demarcation, either using the underlying API or `Transaction`, is never optional!

ConnectionProvider (`org.hibernate.connection.ConnectionProvider`)

(Optional) A factory for (and pool of) JDBC connections. Abstracts application from underlying `Datasource` or `DriverManager`. Not exposed to application, but can be extended/implemented by the developer.

TransactionFactory (`org.hibernate.TransactionFactory`)

(Optional) A factory for `Transaction` instances. Not exposed to the application, but can be extended/implemented by the developer.

Extension Interfaces

Hibernate offers many optional extension interfaces you can implement to customize the behavior of your persistence layer. See the API documentation for details.

Given a "lite" architecture, the application bypasses the `Transaction`/`TransactionFactory` and/or `ConnectionProvider` APIs to talk to JTA or JDBC directly.

2. Instance states

An instance of a persistent classes may be in one of three different states, which are defined with respect to a *persistence context*. The Hibernate `Session` object is the persistence context:

transient

The instance is not, and has never been associated with any persistence context. It has no persistent identity (primary key value).

persistent

The instance is currently associated with a persistence context. It has a persistent identity (primary key value) and, perhaps, a corresponding row in the database. For a particular persistence context, Hibernate *guarantees* that persistent identity is equivalent to Java identity (in-memory location of the object).

detached

The instance was once associated with a persistence context, but that context was closed, or the instance was serialized to another process. It has a persistent identity and, perhaps, a corrsponding row in the database. For detached instances, Hibernate makes no guarantees about the relationship between persistent identity and Java identity.

3. JMX Integration

JMX is the J2EE standard for management of Java components. Hibernate may be managed via a JMX standard service. We provide an MBean implementation in the distribution, `org.hibernate.jmx.HibernateService`.

For an example how to deploy Hibernate as a JMX service on the JBoss Application Server, please see the JBoss User Guide. On JBoss AS, you also get these benefits if you deploy using JMX:

- *Session Management:* The Hibernate `Session`'s lifecycle can be automatically bound to the scope of a JTA transaction. This means you no longer have to manually open and close the `Session`, this becomes the job of a JBoss EJB interceptor. You also don't have to worry about transaction demarcation in your code anymore (unless you'd like to write a portable persistence layer of course, use the optional Hibernate `Transaction` API for this). You call the `HibernateContext` to access a `Session`.
- *HAR deployment:* Usually you deploy the Hibernate JMX service using a JBoss service deployment descriptor (in an EAR and/or SAR file), it supports all the usual configuration options of a Hibernate `SessionFactory`. However, you still have to name all your mapping files in the deployment descriptor. If you decide to use the optional HAR deployment, JBoss will automatically detect all mapping files in your HAR file.

Consult the JBoss AS user guide for more information about these options.

Another feature available as a JMX service are runtime Hibernate statistics. See *Section 4.6, “Hibernate statistics”*.

4. JCA Support

Hibernate may also be configured as a JCA connector. Please see the website for more details. Please note that Hibernate JCA support is still considered experimental.

5. Contextual Sessions

Most applications using Hibernate need some form of "contextual" sessions, where a given session is in effect throughout the scope of a given context. However, across applications the definition of what constitutes a context is typically different; and different contexts define different scopes to the notion of current. Applications using Hibernate prior to version 3.0 tended to utilize either home-grown `ThreadLocal`-based contextual sessions, helper classes such as `HibernateUtil`, or utilized third-party frameworks (such as Spring or Pico) which provided proxy/interception-based contextual sessions.

Starting with version 3.0.1, Hibernate added the `SessionFactory.getCurrentSession()` method. Initially, this assumed usage of `JTA` transactions, where the `JTA` transaction defined both the scope and context of a current session. The Hibernate team maintains that, given the maturity of the numerous stand-alone `JTA TransactionManager` implementations out there, most (if not all) applications should be using `JTA` transaction management whether or not they are deployed into a `J2EE` container. Based on that, the `JTA`-based contextual sessions is all you should ever need to use.

However, as of version 3.1, the processing behind `SessionFactory.getCurrentSession()` is now pluggable. To that end, a new extension interface (`org.hibernate.context.CurrentSessionContext`) and a new configuration parameter (`hibernate.current_session_context_class`) have been added to allow pluggability of the scope and context of defining current sessions.

See the Javadocs for the `org.hibernate.context.CurrentSessionContext` interface for a detailed discussion of its contract. It defines a single method, `currentSession()`, by which the implementation is responsible for tracking the current contextual session. Out-of-the-box, Hibernate comes with three implementations of this interface.

- `org.hibernate.context.JTASessionContext` - current sessions are tracked and scoped by a `JTA` transaction. The processing here is exactly the same as in the older JTA-only approach. See the Javadocs for details.
- `org.hibernate.context.ThreadLocalSessionContext` - current sessions are tracked by thread of execution. Again, see the Javadocs for details.
- `org.hibernate.context.ManagedSessionContext` - current sessions are tracked by thread of execution. However, you are responsible to bind and unbind a `Session` instance with static methods on this class, it does never open, flush, or close a `Session`.

The first two implementations provide a "one session - one database transaction" programming model, also known and used as *session-per-request*. The beginning and end of a Hibernate session is defined by the duration of a database transaction. If you use programatic transaction demarcation in plain JSE without JTA, you are adviced to use the Hibernate `Transaction` API to hide the underlying transaction system from your code. If you use JTA, use the JTA interfaces to demarcate transactions. If you execute in an EJB container that supports CMT, transaction boundaries are defined declaratively and you don't need any transaction or session demarcation operations in your code. Refer to *Chapter 12, Transactions And Concurrency* for more information and code examples.

The `hibernate.current_session_context_class` configuration parameter defines which `org.hibernate.context.CurrentSessionContext` implementation should be used. Note that for backwards compatibility, if this config param is not set but a `org.hibernate.transaction.TransactionManagerLookup` is configured, Hibernate will use the `org.hibernate.context.JTASessionContext`. Typically, the value of this parameter would just name the implementation class to use; for the three out-of-the-box implementations, however, there are two corresponding short names, "jta", "thread", and "managed".

Configuration

Because Hibernate is designed to operate in many different environments, there are a large number of configuration parameters. Fortunately, most have sensible default values and Hibernate is distributed with an example `hibernate.properties` file in `etc/` that shows the various options. Just put the example file in your classpath and customize it.

1. Programmatic configuration

An instance of `org.hibernate.cfg.Configuration` represents an entire set of mappings of an application's Java types to an SQL database. The `Configuration` is used to build an (immutable) `SessionFactory`. The mappings are compiled from various XML mapping files.

You may obtain a `Configuration` instance by instantiating it directly and specifying XML mapping documents. If the mapping files are in the classpath, use `addResource()`:

```
Configuration cfg = new Configuration()
    .addResource("Item.hbm.xml")
    .addResource("Bid.hbm.xml");
```

An alternative (sometimes better) way is to specify the mapped class, and let Hibernate find the mapping document for you:

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class);
```

Then Hibernate will look for mapping files named `/org/hibernate/auction/Item.hbm.xml` and `/org/hibernate/auction/Bid.hbm.xml` in the classpath. This approach eliminates any hardcoded filenames.

A `Configuration` also allows you to specify configuration properties:

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class)
    .setProperty("hibernate.dialect",
"org.hibernate.dialect.MySQLInnoDBDialect")
    .setProperty("hibernate.connection.datasource",
"java:comp/env/jdbc/test")
    .setProperty("hibernate.order_updates", "true");
```

This is not the only way to pass configuration properties to Hibernate. The various options include:

1. Pass an instance of `java.util.Properties` to `Configuration.setProperties()`.
2. Place `hibernate.properties` in a root directory of the classpath.

3. Set `System` properties using `java -Dproperty=value`.
4. Include `<property>` elements in `hibernate.cfg.xml` (discussed later).

`hibernate.properties` is the easiest approach if you want to get started quickly.

The `Configuration` is intended as a startup-time object, to be discarded once a `SessionFactory` is created.

2. Obtaining a SessionFactory

When all mappings have been parsed by the `Configuration`, the application must obtain a factory for `Session` instances. This factory is intended to be shared by all application threads:

```
SessionFactory sessions = cfg.buildSessionFactory();
```

Hibernate does allow your application to instantiate more than one `SessionFactory`. This is useful if you are using more than one database.

3. JDBC connections

Usually, you want to have the `SessionFactory` create and pool JDBC connections for you. If you take this approach, opening a `Session` is as simple as:

```
Session session = sessions.openSession(); // open a new Session
```

As soon as you do something that requires access to the database, a JDBC connection will be obtained from the pool.

For this to work, we need to pass some JDBC connection properties to Hibernate. All Hibernate property names and semantics are defined on the class `org.hibernate.cfg.Environment`. We will now describe the most important settings for JDBC connection configuration.

Hibernate will obtain (and pool) connections using `java.sql.DriverManager` if you set the following properties:

Property name	Purpose
`hibernate.connection.driver_class`	*jdbc driver class*
`hibernate.connection.url`	*jdbc URL*
`hibernate.connection.username`	*database user*
`hibernate.connection.password`	*database user password*
`hibernate.connection.pool_size`	*maximum number of pooled connections*

Table 4.1. Hibernate JDBC Properties

Hibernate's own connection pooling algorithm is however quite rudimentary. It is intended to help you get started and is *not intended for use in a production system* or even for performance testing. You should use a third party pool for best performance and stability. Just replace the `hibernate.connection.pool_size` property with connection pool specific settings. This will turn off Hibernate's internal pool. For example, you might like to use C3P0.

C3P0 is an open source JDBC connection pool distributed along with Hibernate in the `lib` directory. Hibernate will use its `C3P0ConnectionProvider` for connection pooling if you set `hibernate.c3p0.*` properties. If you'd like to use Proxool refer to the packaged `hibernate.properties` and the Hibernate web site for more information.

Here is an example `hibernate.properties` file for C3P0:

```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/mydatabase
hibernate.connection.username = myuser
hibernate.connection.password = secret
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=1800
hibernate.c3p0.max_statements=50
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

For use inside an application server, you should almost always configure Hibernate to obtain connections from an application server `Datasource` registered in JNDI. You'll need to set at least one of the following properties:

Propery name	Purpose
`hibernate.connection.datasource`	*datasource JNDI name*
`hibernate.jndi.url`	*URL of the JNDI provider* (optional)
`hibernate.jndi.class`	*class of the JNDI `InitialContextFactory`* (optional)
`hibernate.connection.username`	*database user* (optional)
`hibernate.connection.password`	*database user password* (optional)

Table 4.2. Hibernate Datasource Properties

Here's an example `hibernate.properties` file for an application server provided JNDI datasource:

```
hibernate.connection.datasource = java:/comp/env/jdbc/test
hibernate.transaction.factory_class = \
    org.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    org.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

JDBC connections obtained from a JNDI datasource will automatically participate in the container-managed transactions of the application server.

Arbitrary connection properties may be given by prepending "`hibernate.connnection`" to the property name. For example, you may specify a `charSet` using `hibernate.connection.charSet`.

You may define your own plugin strategy for obtaining JDBC connections by implementing the interface `org.hibernate.connection.ConnectionProvider`. You may select a custom implementation by setting `hibernate.connection.provider_class`.

4. Optional configuration properties

There are a number of other properties that control the behaviour of Hibernate at runtime. All are optional and have reasonable default values.

Warning: some of these properties are "system-level" only. System-level properties can be set only via `java -Dproperty=value` or `hibernate.properties`. They may *not* be set by the other techniques described above.

Property name	Purpose
`hibernate.dialect`	The classname of a Hibernate `Dialect` which allows Hibernate to generate SQL optimized for a particular relational database. **eg.**`full.classname.of.Dialect`
`hibernate.show_sql`	Write all SQL statements to console. This is an alternative to setting the log category `org.hibernate.SQL` to `debug`. **eg.**`true` | `false`
`hibernate.format_sql`	Pretty print the SQL in the log and console. **eg.**`true` | `false`
`hibernate.default_schema`	Qualify unqualified tablenames with the given schema/tablespace in generated SQL. **eg.**`SCHEMA_NAME`
`hibernate.default_catalog`	Qualify unqualified tablenames with the given catalog in generated SQL. **eg.**`CATALOG_NAME`
`hibernate.session_factory_name`	The `SessionFactory` will be automatically bound to this name in JNDI after it has been created. **eg.**`jndi/composite/name`

Property name	Purpose
`hibernate.max_fetch_depth`	Set a maximum "depth" for the outer join fetch tree for single-ended associations (one-to-one, many-to-one). A `0` disables default outer join fetching. **eg.** recommended values between `0` and `3`
`hibernate.default_batch_fetch_size`	Set a default size for Hibernate batch fetching of associations. **eg.** recommended values `4`, `8`, `16`
`hibernate.default_entity_mode`	Set a default mode for entity representation for all sessions opened from this `SessionFactory` `dynamic-map`, `dom4j`, `pojo`
`hibernate.order_updates`	Force Hibernate to order SQL updates by the primary key value of the items being updated. This will result in fewer transaction deadlocks in highly concurrent systems. **eg.**`true` | `false`
`hibernate.generate_statistics`	If enabled, Hibernate will collect statistics useful for performance tuning. **eg.**`true` | `false`
`hibernate.use_identifer_rollback`	If enabled, generated identifier properties will be reset to default values when objects are deleted. **eg.**`true` | `false`
`hibernate.use_sql_comments`	If turned on, Hibernate will generate comments inside the SQL, for easier debugging, defaults to `false`. **eg.**`true` | `false`

Table 4.3. Hibernate Configuration Properties

Property name	Purpose
`hibernate.jdbc.fetch_size`	A non-zero value determines the JDBC fetch size (calls `Statement.setFetchSize()`).
`hibernate.jdbc.batch_size`	A non-zero value enables use of JDBC2 batch updates by Hibernate.

Property name	Purpose
	eg. recommended values between `5` and `30`
`hibernate.jdbc.batch_versioned_data`	Set this property to `true` if your JDBC driver returns correct row counts from `executeBatch()` (it is usually safe to turn this option on). Hibernate will then use batched DML for automatically versioned data. Defaults to `false`. **eg.**`true` | `false`
`hibernate.jdbc.factory_class`	Select a custom `Batcher`. Most applications will not need this configuration property. **eg.**`classname.of.Batcher`
`hibernate.jdbc.use_scrollable_resultset`	Enables use of JDBC2 scrollable resultsets by Hibernate. This property is only necessary when using user supplied JDBC connections, Hibernate uses connection metadata otherwise. **eg.**`true` | `false`
`hibernate.jdbc.use_streams_for_binary`	Use streams when writing/reading `binary` or `serializable` types to/from JDBC (system-level property). **eg.**`true` | `false`
`hibernate.jdbc.use_get_generated_keys`	Enable use of JDBC3 `PreparedStatement.getGeneratedKeys()` to retrieve natively generated keys after insert. Requires JDBC3+ driver and JRE1.4+, set to false if your driver has problems with the Hibernate identifier generators. By default, tries to determine the driver capabilites using connection metadata. **eg.**`true|false`
`hibernate.connection.provider_class`	The classname of a custom `ConnectionProvider` which provides JDBC connections to Hibernate. **eg.**`classname.of.ConnectionProvider`
`hibernate.connection.isolation`	Set the JDBC transaction isolation level. Check `java.sql.Connection` for meaningful values but note that most databases do not support all isolation levels.

Property name	Purpose
	eg.`1, 2, 4, 8`
`hibernate.connection.autocommit`	Enables autocommit for JDBC pooled connections (not recommended). **eg.**`true` | `false`
`hibernate.connection.release_mode`	Specify when Hibernate should release JDBC connections. By default, a JDBC connection is held until the session is explicitly closed or disconnected. For an application server JTA datasource, you should use `after_statement` to aggressively release connections after every JDBC call. For a non-JTA connection, it often makes sense to release the connection at the end of each transaction, by using `after_transaction`. `auto` will choose `after_statement` for the JTA and CMT transaction strategies and `after_transaction` for the JDBC transaction strategy. **eg.**`auto` (default) | `on_close` | `after_transaction` | `after_statement` Note that this setting only affects `Session` s returned from `SessionFactory.openSession`. For `Session` s obtained through `SessionFactory.getCurrentSession`, the `CurrentSessionContext` implementation configured for use controls the connection release mode for those `Session` s. See *Section 5, “Contextual Sessions”*.
`hibernate.connection.<propertyName>`	Pass the JDBC property `propertyName` to `DriverManager.getConnection()`.
`hibernate.jndi.<propertyName>`	Pass the property `propertyName` to the JNDI `InitialContextFactory`.

Table 4.4. Hibernate JDBC and Connection Properties

Property name	Purpose
`hibernate.cache.provider_class`	The classname of a custom `CacheProvider`. **eg.**`classname.of.CacheProvider`

Property name	Purpose
`hibernate.cache.use_minimal_puts`	Optimize second-level cache operation to minimize writes, at the cost of more frequent reads. This setting is most useful for clustered caches and, in Hibernate3, is enabled by default for clustered cache implementations. **eg.**`true` | `false`
`hibernate.cache.use_query_cache`	Enable the query cache, individual queries still have to be set cachable. **eg.**`true` | `false`
`hibernate.cache.use_second_level_cache`	May be used to completely disable the second level cache, which is enabled by default for classes which specify a `<cache>` mapping. **eg.**`true` | `false`
`hibernate.cache.query_cache_factory`	The classname of a custom `QueryCache` interface, defaults to the built-in `StandardQueryCache`. **eg.**`classname.of.QueryCache`
`hibernate.cache.region_prefix`	A prefix to use for second-level cache region names. **eg.**`prefix`
`hibernate.cache.use_structured_entries`	Forces Hibernate to store data in the second-level cache in a more human-friendly format. **eg.**`true` | `false`

Table 4.5. Hibernate Cache Properties

Property name	Purpose
`hibernate.transaction.factory_class`	The classname of a `TransactionFactory` to use with Hibernate `Transaction` API (defaults to `JDBCTransactionFactory`). **eg.**`classname.of.TransactionFactory`
`jta.UserTransaction`	A JNDI name used by `JTATransactionFactory` to obtain the JTA `UserTransaction` from the application server.

Property name	Purpose
	eg.`jndi/composite/name`
`hibernate.transaction.manager_lookup_class`	The classname of a `TransactionManagerLookup` - required when JVM-level caching is enabled or when using hilo generator in a JTA environment. **eg.**`classname.of.TransactionManagerLookup`
`hibernate.transaction.flush_before_completion`	If enabled, the session will be automatically flushed during the before completion phase of the transaction. Built-in and automatic session context management is preferred, see *Section 5, “Contextual Sessions”*. **eg.**`true` | `false`
`hibernate.transaction.auto_close_session`	If enabled, the session will be automatically closed during the after completion phase of the transaction. Built-in and utomatic session context management is preferred, see *Section 5, “Contextual Sessions”*. **eg.**`true` | `false`

Table 4.6. Hibernate Transaction Properties

Property name	Purpose
`hibernate.current_session_context_class`	Supply a (custom) strategy for the scoping of the "current" `Session`. See *Section 5, “Contextual Sessions”* for more information about the built-in strategies. **eg.**`jta` | `thread` | `managed` | `custom.Class`
`hibernate.query.factory_class`	Chooses the HQL parser implementation. **eg.**`org.hibernate.hql.ast.ASTQueryTranslatorFactory` or `org.hibernate.hql.classic.ClassicQueryTranslatorFactory`
`hibernate.query.substitutions`	Mapping from tokens in Hibernate queries to SQL tokens (tokens might be function or literal names, for example). **eg.**`hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC`
`hibernate.hbm2ddl.auto`	Automatically validate or export schema DDL

Property name	Purpose
	to the database when the `SessionFactory` is created. With `create-drop`, the database schema will be dropped when the `SessionFactory` is closed explicitly. **eg.**`validate` | `update` | `create` | `create-drop`
`hibernate.cglib.use_reflection_optimizer`	Enables use of CGLIB instead of runtime reflection (System-level property). Reflection can sometimes be useful when troubleshooting, note that Hibernate always requires CGLIB even if you turn off the optimizer. You can not set this property in `hibernate.cfg.xml`. **eg.**`true` | `false`

Table 4.7. Miscellaneous Properties

4.1. SQL Dialects

You should always set the `hibernate.dialect` property to the correct `org.hibernate.dialect.Dialect` subclass for your database. If you specify a dialect, Hibernate will use sensible defaults for some of the other properties listed above, saving you the effort of specifying them manually.

RDBMS	Dialect
DB2	`org.hibernate.dialect.DB2Dialect`
DB2 AS/400	`org.hibernate.dialect.DB2400Dialect`
DB2 OS390	`org.hibernate.dialect.DB2390Dialect`
PostgreSQL	`org.hibernate.dialect.PostgreSQLDialect`
MySQL	`org.hibernate.dialect.MySQLDialect`
MySQL with InnoDB	`org.hibernate.dialect.MySQLInnoDBDialect`
MySQL with MyISAM	`org.hibernate.dialect.MySQLMyISAMDialect`
Oracle (any version)	`org.hibernate.dialect.OracleDialect`
Oracle 9i	`org.hibernate.dialect.Oracle9iDialect`
Oracle 10g	`org.hibernate.dialect.Oracle10gDialect`
Sybase	`org.hibernate.dialect.SybaseDialect`
Sybase Anywhere	`org.hibernate.dialect.SybaseAnywhereDialect`
Microsoft SQL Server	`org.hibernate.dialect.SQLServerDialect`

RDBMS	Dialect
SAP DB	`org.hibernate.dialect.SAPDBDialect`
Informix	`org.hibernate.dialect.InformixDialect`
HypersonicSQL	`org.hibernate.dialect.HSQLDialect`
Ingres	`org.hibernate.dialect.IngresDialect`
Progress	`org.hibernate.dialect.ProgressDialect`
Mckoi SQL	`org.hibernate.dialect.MckoiDialect`
Interbase	`org.hibernate.dialect.InterbaseDialect`
Pointbase	`org.hibernate.dialect.PointbaseDialect`
FrontBase	`org.hibernate.dialect.FrontbaseDialect`
Firebird	`org.hibernate.dialect.FirebirdDialect`

Table 4.8. Hibernate SQL Dialects (`hibernate.dialect`)

4.2. Outer Join Fetching

If your database supports ANSI, Oracle or Sybase style outer joins, *outer join fetching* will often increase performance by limiting the number of round trips to and from the database (at the cost of possibly more work performed by the database itself). Outer join fetching allows a whole graph of objects connected by many-to-one, one-to-many, many-to-many and one-to-one associations to be retrieved in a single SQL `SELECT`.

Outer join fetching may be disabled *globally* by setting the property `hibernate.max_fetch_depth` to `0`. A setting of `1` or higher enables outer join fetching for one-to-one and many-to-one associations which have been mapped with `fetch="join"`.

See *Section 1, “Fetching strategies”* for more information.

4.3. Binary Streams

Oracle limits the size of `byte` arrays that may be passed to/from its JDBC driver. If you wish to use large instances of `binary` or `serializable` type, you should enable `hibernate.jdbc.use_streams_for_binary`. *This is a system-level setting only.*

4.4. Second-level and query cache

The properties prefixed by `hibernate.cache` allow you to use a process or cluster scoped second-level cache system with Hibernate. See the *Section 2, “The Second Level Cache”* for more details.

4.5. Query Language Substitution

You may define new Hibernate query tokens using `hibernate.query.substitutions`. For example:

```
hibernate.query.substitutions true=1, false=0
```

would cause the tokens `true` and `false` to be translated to integer literals in the generated SQL.

```
hibernate.query.substitutions toLowercase=LOWER
```

would allow you to rename the SQL `LOWER` function.

4.6. Hibernate statistics

If you enable `hibernate.generate_statistics`, Hibernate will expose a number of metrics that are useful when tuning a running system via `SessionFactory.getStatistics()`. Hibernate can even be configured to expose these statistics via JMX. Read the Javadoc of the interfaces in `org.hibernate.stats` for more information.

5. Logging

Hibernate logs various events using Apache commons-logging.

The commons-logging service will direct output to either Apache Log4j (if you include `log4j.jar` in your classpath) or JDK1.4 logging (if running under JDK1.4 or above). You may download Log4j from `http://jakarta.apache.org`. To use Log4j you will need to place a `log4j.properties` file in your classpath, an example properties file is distributed with Hibernate in the `src/` directory.

We strongly recommend that you familiarize yourself with Hibernate's log messages. A lot of work has been put into making the Hibernate log as detailed as possible, without making it unreadable. It is an essential troubleshooting device. The most interesting log categories are the following:

Category	Function
`org.hibernate.SQL`	Log all SQL DML statements as they are executed
`org.hibernate.type`	Log all JDBC parameters
`org.hibernate.tool.hbm2ddl`	Log all SQL DDL statements as they are executed
`org.hibernate.pretty`	Log the state of all entities (max 20 entities) associated with the session at flush time
`org.hibernate.cache`	Log all second-level cache activity
`org.hibernate.transaction`	Log transaction related activity
`org.hibernate.jdbc`	Log all JDBC resource acquisition
`org.hibernate.hql.ast.AST`	Log HQL and SQL ASTs during query parsing
`org.hibernate.secure`	Log all JAAS authorization requests

Category	Function
`org.hibernate`	Log everything (a lot of information, but very useful for troubleshooting)

Table 4.9. Hibernate Log Categories

When developing applications with Hibernate, you should almost always work with `debug` enabled for the category `org.hibernate.SQL`, or, alternatively, the property `hibernate.show_sql` enabled.

6. Implementing a NamingStrategy

The interface `org.hibernate.cfg.NamingStrategy` allows you to specify a "naming standard" for database objects and schema elements.

You may provide rules for automatically generating database identifiers from Java identifiers or for processing "logical" column and table names given in the mapping file into "physical" table and column names. This feature helps reduce the verbosity of the mapping document, eliminating repetitive noise (`TBL_` prefixes, for example). The default strategy used by Hibernate is quite minimal.

You may specify a different strategy by calling `Configuration.setNamingStrategy()` before adding mappings:

```
SessionFactory sf = new Configuration()
    .setNamingStrategy(ImprovedNamingStrategy.INSTANCE)
    .addFile("Item.hbm.xml")
    .addFile("Bid.hbm.xml")
    .buildSessionFactory();
```

`org.hibernate.cfg.ImprovedNamingStrategy` is a built-in strategy that might be a useful starting point for some applications.

7. XML configuration file

An alternative approach to configuration is to specify a full configuration in a file named `hibernate.cfg.xml`. This file can be used as a replacement for the `hibernate.properties` file or, if both are present, to override properties.

The XML configuration file is by default expected to be in the root o your `CLASSPATH`. Here is an example:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
```

```
<hibernate-configuration>

    <!-- a SessionFactory instance listed as /jndi/name -->
    <session-factory
        name="java:hibernate/SessionFactory">

        <!-- properties -->
        <property
name="connection.datasource">java:/comp/env/jdbc/MyDB</property>
        <property
name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">false</property>
        <property name="transaction.factory_class">
            org.hibernate.transaction.JTATransactionFactory
        </property>
        <property
name="jta.UserTransaction">java:comp/UserTransaction</property>

        <!-- mapping files -->
        <mapping resource="org/hibernate/auction/Item.hbm.xml"/>
        <mapping resource="org/hibernate/auction/Bid.hbm.xml"/>

        <!-- cache settings -->
        <class-cache class="org.hibernate.auction.Item" usage="read-write"/>
        <class-cache class="org.hibernate.auction.Bid" usage="read-only"/>
        <collection-cache collection="org.hibernate.auction.Item.bids"
usage="read-write"/>

    </session-factory>

</hibernate-configuration>
```

As you can see, the advantage of this approach is the externalization of the mapping file names to configuration. The `hibernate.cfg.xml` is also more convenient once you have to tune the Hibernate cache. Note that is your choice to use either `hibernate.properties` or `hibernate.cfg.xml`, both are equivalent, except for the above mentioned benefits of using the XML syntax.

With the XML configuration, starting Hibernate is then as simple as

```
SessionFactory sf = new Configuration().configure().buildSessionFactory();
```

You can pick a different XML configuration file using

```
SessionFactory sf = new Configuration()
    .configure("catdb.cfg.xml")
    .buildSessionFactory();
```

8. J2EE Application Server integration

Hibernate has the following integration points for J2EE infrastructure:

- *Container-managed datasources*: Hibernate can use JDBC connections managed by the container and provided through JNDI. Usually, a JTA compatible `TransactionManager` and a `ResourceManager` take care of transaction management (CMT), esp. distributed transaction handling across several datasources. You may of course also demarcate transaction boundaries programatically (BMT) or you might want to use the optional Hibernate `Transaction` API for this to keep your code portable.

- *Automatic JNDI binding*: Hibernate can bind its `SessionFactory` to JNDI after startup.

- *JTA Session binding:* The Hibernate `Session` may be automatically bound to the scope of JTA transactions. Simply lookup the `SessionFactory` from JNDI and get the current `Session`. Let Hibernate take care of flushing and closing the `Session` when your JTA transaction completes. Transaction demarcation is either declarative (CMT) or programmatic (BMT/UserTransaction).

- *JMX deployment:* If you have a JMX capable application server (e.g. JBoss AS), you can chose to deploy Hibernate as a managed MBean. This saves you the one line startup code to build your `SessionFactory` from a `Configuration`. The container will startup your `HibernateService`, and ideally also take care of service dependencies (Datasource has to be available before Hibernate starts, etc).

Depending on your environment, you might have to set the configuration option `hibernate.connection.aggressive_release` to true if your application server shows "connection containment" exceptions.

8.1. Transaction strategy configuration

The Hibernate `Session` API is independent of any transaction demarcation system in your architecture. If you let Hibernate use JDBC directly, through a connection pool, you may begin and end your transactions by calling the JDBC API. If you run in a J2EE application server, you might want to use bean-managed transactions and call the JTA API and `UserTransaction` when needed.

To keep your code portable between these two (and other) environments we recommend the optional Hibernate `Transaction` API, which wraps and hides the underlying system. You have to specify a factory class for `Transaction` instances by setting the Hibernate configuration property `hibernate.transaction.factory_class`.

There are three standard (built-in) choices:

`org.hibernate.transaction.JDBCTransactionFactory`
delegates to database (JDBC) transactions (default)

`org.hibernate.transaction.JTATransactionFactory`
delegates to container-managed transaction if an existing transaction is underway in this context (e.g. EJB session bean method), otherwise a new transaction is started and bean-managed transaction are used.

`org.hibernate.transaction.CMTTransactionFactory`
delegates to container-managed JTA transactions

You may also define your own transaction strategies (for a CORBA transaction service, for example).

Some features in Hibernate (i.e. the second level cache, Contextual Sessions with JTA, etc.) require access to the JTA `TransactionManager` in a managed environment. In an application server you have to specify how Hibernate should obtain a reference to the `TransactionManager`, since J2EE does not standardize a single mechanism:

Transaction Factory	Application Server
`org.hibernate.transaction.JBossTransactionManagerLookup`	JBoss
`org.hibernate.transaction.WeblogicTransactionManagerLookup`	Weblogic
`org.hibernate.transaction.WebSphereTransactionManagerLookup`	WebSphere
`org.hibernate.transaction.WebSphereExtendedJTATransactionLookup`	WebSphere 6
`org.hibernate.transaction.OrionTransactionManagerLookup`	Orion
`org.hibernate.transaction.ResinTransactionManagerLookup`	Resin
`org.hibernate.transaction.JOTMTransactionManagerLookup`	JOTM
`org.hibernate.transaction.JOnASTransactionManagerLookup`	JOnAS
`org.hibernate.transaction.JRun4TransactionManagerLookup`	JRun4
`org.hibernate.transaction.BESTransactionManagerLookup`	Borland ES

Table 4.10. JTA TransactionManagers

8.2. JNDI-bound SessionFactory

A JNDI bound Hibernate `SessionFactory` can simplify the lookup of the factory and the creation of new `Session` s. Note that this is not related to a JNDI bound `Datasource`, both simply use the same registry!

If you wish to have the `SessionFactory` bound to a JNDI namespace, specify a name (eg. `java:hibernate/SessionFactory`) using the property `hibernate.session_factory_name`. If this property is omitted, the `SessionFactory` will not be bound to JNDI. (This is especially useful in environments with a read-only JNDI default implementation, e.g. Tomcat.)

When binding the `SessionFactory` to JNDI, Hibernate will use the values of `hibernate.jndi.url`, `hibernate.jndi.class` to instantiate an initial context. If they are not specified, the default `InitialContext` will be used.

Hibernate will automatically place the `SessionFactory` in JNDI after you call `cfg.buildSessionFactory()`. This means you will at least have this call in some startup code (or utility class) in your application, unless you use JMX deployment with the `HibernateService` (discussed later).

If you use a JNDI `SessionFactory`, an EJB or any other class may obtain the `SessionFactory` using a JNDI lookup.

We recommend that you bind the `SessionFactory` to JNDI in a managend environment and use a `static` singleton otherwise. To shield your application code from these details, we also recommend to hide the actual lookup code for a `SessionFactory` in a helper class, such as `HibernateUtil.getSessionFactory()`. Note that such a class is also a convenient way to startup Hibernate - see chapter 1.

8.3. Current Session context management with JTA

The easiest way to handle `Session` s and transactions is Hibernates automatic "current" `Session` management. See the discussion of *Section 5, “Contextual Sessions”* current sessions. Using the `"jta"` session context, if there is no Hibernate `Session` associated with the current JTA transaction, one will be started and associated with that JTA transaction the first time you call `sessionFactory.getCurrentSession()`. The `Session` s retrieved via `getCurrentSession()` in `"jta"` context will be set to automatically flush before the transaction completes, close after the transaction completes, and aggressively release JDBC connections after each statement. This allows the `Session` s to be managed by the lifecycle of the JTA transaction to which it is associated, keeping user code clean of such management concerns. Your code can either use JTA programmatically through `UserTransaction`, or (recommended for portable code) use the Hibernate `Transaction` API to set transaction boundaries. If you run in an EJB container, declarative transaction demarcation with CMT is preferred.

8.4. JMX deployment

The line `cfg.buildSessionFactory()` still has to be executed somewhere to get a `SessionFactory` into JNDI. You can do this either in a `static` initializer block (like the one in `HibernateUtil`) or you deploy Hibernate as a *managed service*.

Hibernate is distributed with `org.hibernate.jmx.HibernateService` for deployment on an application server with JMX capabilities, such as JBoss AS. The actual deployment and configuration is vendor specific. Here is an example `jboss-service.xml` for JBoss 4.0.x:

```
<?xml version="1.0"?>
<server>

<mbean code="org.hibernate.jmx.HibernateService"
    name="jboss.jca:service=HibernateFactory,name=HibernateFactory">
    <!-- Required services -->
    <depends>jboss.jca:service=RARDeployer</depends>
    <depends>jboss.jca:service=LocalTxCM,name=HsqlDS</depends>

    <!-- Bind the Hibernate service to JNDI -->
    <attribute name="JndiName">java:/hibernate/SessionFactory</attribute>

    <!-- Datasource settings -->
    <attribute name="Datasource">java:HsqlDS</attribute>
    <attribute name="Dialect">org.hibernate.dialect.HSQLDialect</attribute>

    <!-- Transaction integration -->
    <attribute name="TransactionStrategy">
        org.hibernate.transaction.JTATransactionFactory</attribute>
    <attribute name="TransactionManagerLookupStrategy">
        org.hibernate.transaction.JBossTransactionManagerLookup</attribute>
    <attribute name="FlushBeforeCompletionEnabled">true</attribute>
    <attribute name="AutoCloseSessionEnabled">true</attribute>

    <!-- Fetching options -->
    <attribute name="MaximumFetchDepth">5</attribute>

    <!-- Second-level caching -->
    <attribute name="SecondLevelCacheEnabled">true</attribute>
    <attribute
name="CacheProviderClass">org.hibernate.cache.EhCacheProvider</attribute>
    <attribute name="QueryCacheEnabled">true</attribute>

    <!-- Logging -->
    <attribute name="ShowSqlEnabled">true</attribute>

    <!-- Mapping files -->
    <attribute
name="MapResources">auction/Item.hbm.xml,auction/Category.hbm.xml</attribute>

</mbean>

</server>
```

This file is deployed in a directory called `META-INF` and packaged in a JAR file with the extension `.sar` (service archive). You also need to package Hibernate, its required third-party libraries, your compiled persistent classes, as well as your mapping files in the same archive. Your enterprise beans (usually session beans) may be kept in their own JAR file, but you may include this EJB JAR file in the main service archive to get a single (hot-)deployable unit. Consult the JBoss AS documentation for more information about JMX service and EJB deployment.

Persistent Classes

Persistent classes are classes in an application that implement the entities of the business problem (e.g. Customer and Order in an E-commerce application). Not all instances of a persistent class are considered to be in the persistent state - an instance may instead be transient or detached.

Hibernate works best if these classes follow some simple rules, also known as the Plain Old Java Object (POJO) programming model. However, none of these rules are hard requirements. Indeed, Hibernate3 assumes very little about the nature of your persistent objects. You may express a domain model in other ways: using trees of `Map` instances, for example.

1. A simple POJO example

Most Java applications require a persistent class representing felines.

```
package eg;
import java.util.Set;
import java.util.Date;

public class Cat {
    private Long id; // identifier

    private Date birthdate;
    private Color color;
    private char sex;
    private float weight;
    private int litterId;

    private Cat mother;
    private Set kittens = new HashSet();

    private void setId(Long id) {
        this.id=id;
    }
    public Long getId() {
        return id;
    }

    void setBirthdate(Date date) {
        birthdate = date;
    }
    public Date getBirthdate() {
        return birthdate;
    }

    void setWeight(float weight) {
        this.weight = weight;
    }
    public float getWeight() {
        return weight;
    }
    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }

    void setSex(char sex) {
        this.sex=sex;
    }
    public char getSex() {
        return sex;
    }

    void setLitterId(int id) {
        this.litterId = id;
    }
    public int getLitterId() {
        return litterId;
    }

    void setMother(Cat mother) {
        this.mother = mother;
    }
    public Cat getMother() {
        return mother;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    public Set getKittens() {
        return kittens;
    }

    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
    kitten.setMother(this);
kitten.setLitterId( kittens.size() );
        kittens.add(kitten);
    }
}
```

There are four main rules to follow here:

1.1. Implement a no-argument constructor

`Cat` has a no-argument constructor. All persistent classes must have a default constructor (which may be non-public) so that Hibernate can instantiate them using `Constructor.newInstance()`. We strongly recommend having a default constructor with at least *package* visibility for runtime proxy generation in Hibernate.

1.2. Provide an identifier property (optional)

`Cat` has a property called `id`. This property maps to the primary key column of a database table. The property might have been called anything, and its type might have been any primitive type, any primitive "wrapper" type, `java.lang.String` or `java.util.Date`. (If your legacy database table has composite keys, you can even use a user-defined class with properties of these types - see the section on composite identifiers later.)

The identifier property is strictly optional. You can leave them off and let Hibernate keep track of object identifiers internally. We do not recommend this, however.

In fact, some functionality is available only to classes which declare an identifier property:

- Transitive reattachment for detached objects (cascade update or cascade merge) - see *Section 11, “Transitive persistence”*
- `Session.saveOrUpdate()`
- `Session.merge()`

We recommend you declare consistently-named identifier properties on persistent classes. We further recommend that you use a nullable (ie. non-primitive) type.

1.3. Prefer non-final classes (optional)

A central feature of Hibernate, *proxies*, depends upon the persistent class being either non-final, or the implementation of an interface that declares all public methods.

You can persist `final` classes that do not implement an interface with Hibernate, but you won't be able to use proxies for lazy association fetching - which will limit your options for performance tuning.

You should also avoid declaring `public final` methods on the non-final classes. If you want to use a class with a `public final` method, you must explicitly disable proying by setting `lazy="false"`.

1.4. Declare accessors and mutators for persistent fields (optional)

`Cat` declares accessor methods for all its persistent fields. Many other ORM tools directly persist instance variables. We believe it is better to provide an indirection between the relational schema and internal data structures of the class. By default, Hibernate persists JavaBeans style properties, and recognizes method names of the form `getFoo`, `isFoo` and `setFoo`. You may switch to direct field access for particular properties, if needed.

Properties need *not* be declared public - Hibernate can persist a property with a default, `protected` or `private` get / set pair.

2. Implementing inheritance

A subclass must also observe the first and second rules. It inherits its identifier property from the superclass, `Cat`.

```
package eg;

public class DomesticCat extends Cat {
        private String name;

        public String getName() {
                return name;
        }
        protected void setName(String name) {
                this.name=name;
        }
}
```

3. Implementing equals() and hashCode()

You have to override the `equals()` and `hashCode()` methods if you

- intend to put instances of persistent classes in a `Set` (the recommended way to represent many-valued associations) *and*
- intend to use reattachment of detached instances

Hibernate guarantees equivalence of persistent identity (database row) and Java identity only inside a particular session scope. So as soon as we mix instances retrieved in different sessions, we must implement `equals()` and `hashCode()` if we wish to have meaningful semantics for `Set` s.

The most obvious way is to implement `equals()`/`hashCode()` by comparing the identifier value of both objects. If the value is the same, both must be the same database row, they are therefore equal (if both are added to a `Set`, we will only have one element in the `Set`). Unfortunately, we can't use that approach with generated identifiers! Hibernate will only assign identifier values to objects that are persistent, a newly created instance will not have any identifier value! Furthermore, if an instance is unsaved and currently in a `Set`, saving it will assign an identifier value to the object. If `equals()` and `hashCode()` are based on the identifier value, the hash code would change, breaking the contract of the `Set`. See the Hibernate website for a full discussion of this problem. Note that this is not a Hibernate issue, but normal Java semantics of object identity and equality.

We recommend implementing `equals()` and `hashCode()` using *Business key equality*. Business key equality means that the `equals()` method compares only the properties that form the business key, a key that would identify our instance in the real world (a *natural* candidate key):

```
public class Cat {

    ...
    public boolean equals(Object other) {
        if (this == other) return true;
        if ( !(other instanceof Cat) ) return false;

        final Cat cat = (Cat) other;

        if ( !cat.getLitterId().equals( getLitterId() ) ) return false;
        if ( !cat.getMother().equals( getMother() ) ) return false;

        return true;
    }

    public int hashCode() {
        int result;
        result = getMother().hashCode();
        result = 29 * result + getLitterId();
        return result;
    }

}
```

Note that a business key does not have to be as solid as a database primary key candidate (see *Section 1.3, “Considering object identity”*). Immutable or unique properties are usually good candidates for a business key.

4. Dynamic models

Note that the following features are currently considered experimental and may change in the near future.

Persistent entities don't necessarily have to be represented as POJO classes or as JavaBean objects at runtime. Hibernate also supports dynamic models (using `Map` s of `Map` s at runtime) and the representation of entities as DOM4J trees. With this approach, you don't write persistent classes, only mapping files.

By default, Hibernate works in normal POJO mode. You may set a default entity representation mode for a particular `SessionFactory` using the `default_entity_mode` configuration option (see *Table 4.3, “Hibernate Configuration Properties”*.

The following examples demonstrates the representation using `Map` s. First, in the mapping file, an `entity-name` has to be declared instead of (or in addition to) a class name:

```
<hibernate-mapping>

    <class entity-name="Customer">

        <id name="id"
            type="long"
            column="ID">
            <generator class="sequence"/>
        </id>

        <property name="name"
            column="NAME"
```

```
              type="string"/>

          <property name="address"
              column="ADDRESS"
              type="string"/>

          <many-to-one name="organization"
              column="ORGANIZATION_ID"
              class="Organization"/>

          <bag name="orders"
              inverse="true"
              lazy="false"
              cascade="all">
              <key column="CUSTOMER_ID"/>
              <one-to-many class="Order"/>
          </bag>

      </class>

  </hibernate-mapping>
```

Note that even though associations are declared using target class names, the target type of an associations may also be a dynamic entity instead of a POJO.

After setting the default entity mode to `dynamic-map` for the `SessionFactory`, we can at runtime work with `Map` s of `Map` s:

```
Session s = openSession();
Transaction tx = s.beginTransaction();
Session s = openSession();

// Create a customer
Map david = new HashMap();
david.put("name", "David");

// Create an organization
Map foobar = new HashMap();
foobar.put("name", "Foobar Inc.");

// Link both
david.put("organization", foobar);

// Save both
s.save("Customer", david);
s.save("Organization", foobar);

tx.commit();
s.close();
```

The advantages of a dynamic mapping are quick turnaround time for prototyping without the need for entity class implementation. However, you lose compile-time type checking and will very likely deal with many exceptions at runtime. Thanks to the Hibernate mapping, the database schema can easily be normalized and sound, allowing to add a proper domain model implementation on top later on.

Entity representation modes can also be set on a per `Session` basis:

```
Session dynamicSession = pojoSession.getSession(EntityMode.MAP);

// Create a customer
Map david = new HashMap();
david.put("name", "David");
dynamicSession.save("Customer", david);
...
dynamicSession.flush();
dynamicSession.close()
...
// Continue on pojoSession
```

Please note that the call to `getSession()` using an `EntityMode` is on the `Session` API, not the `SessionFactory`. That way, the new `Session` shares the underlying JDBC connection, transaction, and other context information. This means you don't have tocall `flush()` and `close()` on the secondary `Session`, and also leave the transaction and connection handling to the primary unit of work.

More information about the XML representation capabilities can be found in *Chapter 19, XML Mapping*.

5. Tuplizers

`org.hibernate.tuple.Tuplizer`, and its sub-interfaces, are responsible for managing a particular representation of a piece of data, given that representation's `org.hibernate.EntityMode`. If a given piece of data is thought of as a data structure, then a tuplizer is the thing which knows how to create such a data structure and how to extract values from and inject values into such a data structure. For example, for the POJO entity mode, the correpsonding tuplizer knows how create the POJO through its constructor and how to access the POJO properties using the defined property accessors. There are two high-level types of Tuplizers, represented by the `org.hibernate.tuple.entity.EntityTuplizer` and `org.hibernate.tuple.component.ComponentTuplizer` interfaces. `EntityTuplizer` s are responsible for managing the above mentioned contracts in regards to entities, while `ComponentTuplizer` s do the same for components.

Users may also plug in their own tuplizers. Perhaps you require that a `java.util.Map` implementation other than `java.util.HashMap` be used while in the dynamic-map entity-mode; or perhaps you need to define a different proxy generation strategy than the one used by default. Both would be achieved by defining a custom tuplizer implementation. Tuplizers definitions are attached to the entity or component mapping they are meant to manage. Going back to the example of our customer entity:

```
<hibernate-mapping>
```

```
    <class entity-name="Customer">
        <!--
            Override the dynamic-map entity-mode
            tuplizer for the customer entity
        -->
        <tuplizer entity-mode="dynamic-map"
                class="CustomMapTuplizerImpl"/>

        <id name="id" type="long" column="ID">
            <generator class="sequence"/>
        </id>

        <!-- other properties -->
        ...
    </class>
</hibernate-mapping>


public class CustomMapTuplizerImpl
        extends org.hibernate.tuple.entity.DynamicMapEntityTuplizer {
    // override the buildInstantiator() method to plug in our custom map...
    protected final Instantiator buildInstantiator(
            org.hibernate.mapping.PersistentClass mappingInfo) {
        return new CustomMapInstantiator( mappingInfo );
    }

    private static final class CustomMapInstantiator
            extends org.hibernate.tuple.DynamicMapInstantitor {
        // override the generateMap() method to return our custom map...
    protected final Map generateMap() {
    return new CustomMap();
    }
    }
}
```

Basic O/R Mapping

1. Mapping declaration

Object/relational mappings are usually defined in an XML document. The mapping document is designed to be readable and hand-editable. The mapping language is Java-centric, meaning that mappings are constructed around persistent class declarations, not table declarations.

Note that, even though many Hibernate users choose to write the XML by hand, a number of tools exist to generate the mapping document, including XDoclet, Middlegen and AndroMDA.

Lets kick off with an example mapping:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat"
            table="cats"
            discriminator-value="C">

                <id name="id">
                        <generator class="native"/>
                </id>

                <discriminator column="subclass"
                     type="character"/>

                <property name="weight"/>

                <property name="birthdate"
                    type="date"
                    not-null="true"
                    update="false"/>

                <property name="color"
                    type="eg.types.ColorUserType"
                    not-null="true"
                    update="false"/>

                <property name="sex"
                    not-null="true"
                    update="false"/>

                <property name="litterId"
                    column="litterId"
                    update="false"/>

                <many-to-one name="mother"
                    column="mother_id"
                    update="false"/>
                <set name="kittens"
                    inverse="true"
                    order-by="litter_id">
                        <key column="mother_id"/>
                        <one-to-many class="Cat"/>
                </set>

                <subclass name="DomesticCat"
                    discriminator-value="D">

                        <property name="name"
                            type="string"/>

                </subclass>

        </class>

        <class name="Dog">
                <!-- mapping for Dog could go here -->
        </class>

</hibernate-mapping>
```

We will now discuss the content of the mapping document. We will only describe the document elements and attributes that are used by Hibernate at runtime. The mapping document also contains some extra optional attributes and elements that affect the database schemas exported by the schema export tool. (For example the `not-null` attribute.)

1.1. Doctype

All XML mappings should declare the doctype shown. The actual DTD may be found at the URL above, in the directory `hibernate-x.x.x/src/org/hibernate` or in `hibernate3.jar`. Hibernate will always look for the DTD in its classpath first. If you experience lookups of the DTD using an Internet connection, check your DTD declaration against the contents of your claspath.

1.1.1. EntityResolver

As mentioned previously, Hibernate will first attempt to resolve DTDs in its classpath. The manner in which it does this is by registering a custom `org.xml.sax.EntityResolver` implementation with the SAXReader it uses to read in the xml files. This custom `EntityResolver` recognizes two different systemId namespaces.

- a `hibernate namespace` is recognized whenever the resolver encounteres a systemId starting with `http://hibernate.sourceforge.net/`; the resolver attempts to resolve these entities via the classlaoder which loaded the Hibernate classes.

- a `user namespace` is recognized whenever the resolver encounteres a systemId using a `classpath://` URL protocol; the resolver will attempt to resolve these entities via (1) the current thread context classloader and (2) the classloader which loaded the Hibernate classes.

An example of utilizing user namespacing:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" [
    <!ENTITY types SYSTEM "classpath://your/domain/types.xml">
]>

<hibernate-mapping package="your.domain">
    <class name="MyEntity">
        <id name="id" type="my-custom-id-type">
            ...
        </id>
    </class>
    & types;
</hibernate-mapping>
```

Where `types.xml` is a resource in the `your.domain` package and contains a custom *Section 2.3, “Custom value types”* typedef.

1.2. hibernate-mapping

This element has several optional attributes. The `schema` and `catalog` attributes specify that tables referred to in this mapping belong to the named schema and/or catalog. If specified, tablenames will be qualified by the given schema and catalog names. If missing, tablenames will be unqualified. The `default-cascade` attribute specifies what cascade style should be assumed for properties and collections which do not specify a `cascade` attribute. The `auto-import` attribute lets us use unqualified class names in the query language, by default.

```
<hibernate-mapping
         schema="schemaName"
         catalog="catalogName"
         default-cascade="cascade_style"
         default-access="field|property|ClassName"
         default-lazy="true|false"
         auto-import="true|false"
         package="package.name"
 />
```

`schema` (optional): The name of a database schema.

`catalog` (optional): The name of a database catalog.

`default-cascade` (optional - defaults to `none`): A default cascade style.

`default-access` (optional - defaults to `property`): The strategy Hibernate should use for accessing all properties. Can be a custom implementation of `PropertyAccessor`.

`default-lazy` (optional - defaults to `true`): The default value for unspecifed `lazy` attributes of class and collection mappings.

`auto-import` (optional - defaults to `true`): Specifies whether we can use unqualified class names (of classes in this mapping) in the query language.

`package` (optional): Specifies a package prefix to assume for unqualified class names in the mapping document.

If you have two persistent classes with the same (unqualified) name, you should set `auto-import="false"`. Hibernate will throw an exception if you attempt to assign two classes to the same "imported" name.

Note that the `hibernate-mapping` element allows you to nest several persistent `<class>` mappings, as shown above. It is however good practice (and expected by some tools) to map only a single persistent class (or a single class hierarchy) in one mapping file and name it after the persistent superclass, e.g. `Cat.hbm.xml`, `Dog.hbm.xml`, or if using inheritance, `Animal.hbm.xml`.

1.3. class

You may declare a persistent class using the `class` element:

```
<class
        name="ClassName"
        table="tableName"
        discriminator-value="discriminator_value"
        mutable="true|false"
        schema="owner"
        catalog="catalog"
        proxy="ProxyInterface"
        dynamic-update="true|false"
        dynamic-insert="true|false"
        select-before-update="true|false"
        polymorphism="implicit|explicit"
        where="arbitrary sql where condition"
        persister="PersisterClass"
        batch-size="N"
        optimistic-lock="none|version|dirty|all"
        lazy="true|false"
        entity-name="EntityName"
        check="arbitrary sql check condition"
        rowid="rowid"
        subselect="SQL expression"
        abstract="true|false"
        node="element-name"
/>
```

`name` (optional): The fully qualified Java class name of the persistent class (or interface). If this attribute is missing, it is assumed that the mapping is for a non-POJO entity.

`table` (optional - defaults to the unqualified class name): The name of its database table.

`discriminator-value` (optional - defaults to the class name): A value that distiguishes individual subclasses, used for polymorphic behaviour. Acceptable values include `null` and `not null`.

`mutable` (optional, defaults to `true`): Specifies that instances of the class are (not) mutable.

`schema` (optional): Override the schema name specified by the root `<hibernate-mapping>` element.

`catalog` (optional): Override the catalog name specified by the root `<hibernate-mapping>` element.

`proxy` (optional): Specifies an interface to use for lazy initializing proxies. You may specify the name of the class itself.

`dynamic-update` (optional, defaults to `false`): Specifies that `UPDATE` SQL should be generated at runtime and contain only those columns whose values have changed.

`dynamic-insert` (optional, defaults to `false`): Specifies that `INSERT` SQL should be generated at runtime and contain only the columns whose values are not null.

`select-before-update` (optional, defaults to `false`): Specifies that Hibernate should *never* perform an SQL `UPDATE` unless it is certain that an object is actually modified. In certain cases (actually, only when a transient object has been associated with a new session using `update()`), this means that Hibernate will perform an extra SQL `SELECT` to determine if an `UPDATE` is actually required.

`polymorphism` (optional, defaults to `implicit`): Determines whether implicit or explicit query polymorphism is used.

`where` (optional) specify an arbitrary SQL `WHERE` condition to be used when retrieving objects of this class

`persister` (optional): Specifies a custom `ClassPersister`.

`batch-size` (optional, defaults to `1`) specify a "batch size" for fetching instances of this class by identifier.

`optimistic-lock` (optional, defaults to `version`): Determines the optimistic locking strategy.

`lazy` (optional): Lazy fetching may be completely disabled by setting `lazy="false"`.

`entity-name` (optional, defaults to the class name): Hibernate3 allows a class to be mapped multiple times (to different tables, potentially), and allows entity mappings that are represented by Maps or XML at the Java level. In these cases, you should provide an explicit arbitrary name for the entity. See *Section 4, “Dynamic models”* and *Chapter 19, XML Mapping* for more information.

`check` (optional): A SQL expression used to generate a multi-row *check* constraint for automatic schema generation.

`rowid` (optional): Hibernate can use so called ROWIDs on databases which support. E.g. on Oracle, Hibernate can use the `rowid` extra column for fast updates if you set this option to `rowid`. A ROWID is an implementation detail and represents the physical location of a stored tuple.

`subselect` (optional): Maps an immutable and read-only entity to a database subselect. Useful if you want to have a view instead of a base table, but don't. See below for more information.

`abstract` (optional): Used to mark abstract superclasses in `<union-subclass>` hierarchies.

It is perfectly acceptable for the named persistent class to be an interface. You would then declare implementing classes of that interface using the `<subclass>` element. You may persist any *static* inner class. You should specify the class name using the standard form ie. `eg.Foo$Bar`.

Immutable classes, `mutable="false"`, may not be updated or deleted by the application. This allows Hibernate to make some minor performance optimizations.

The optional `proxy` attribute enables lazy initialization of persistent instances of the class. Hibernate will initially return CGLIB proxies which implement the named interface. The actual persistent object will be loaded when a method of the proxy is invoked. See "Proxies for Lazy Initialization" below.

Implicit polymorphism means that instances of the class will be returned by a query that names any superclass or implemented interface or the class and that instances of any subclass of the class will be returned by a query that names the class itself. *Explicit* polymorphism means that class instances will be returned only by queries that explicitly name that class and that queries that name the class will return only instances of subclasses mapped inside this `<class>` declaration as a `<subclass>` or `<joined-subclass>`. For most purposes the default, `polymorphism="implicit"`, is appropriate. Explicit polymorphism is useful when two different classes are mapped to the same table (this allows a "lightweight" class that contains a subset of the table columns).

The `persister` attribute lets you customize the persistence strategy used for the class. You may, for example, specify your own subclass of `org.hibernate.persister.EntityPersister` or you might even provide a completely new implementation of the interface `org.hibernate.persister.ClassPersister` that implements persistence via, for example, stored procedure calls, serialization to flat files or LDAP. See `org.hibernate.test.CustomPersister` for a simple example (of "persistence" to a `Hashtable`).

Note that the `dynamic-update` and `dynamic-insert` settings are not inherited by subclasses and so may also be specified on the `<subclass>` or `<joined-subclass>` elements. These settings may increase performance in some cases, but might actually decrease performance in others. Use judiciously.

Use of `select-before-update` will usually decrease performance. It is very useful to prevent a database update trigger being called unnecessarily if you reattach a graph of detached instances to a `Session`.

If you enable `dynamic-update`, you will have a choice of optimistic locking strategies:

- `version` check the version/timestamp columns
- `all` check all columns
- `dirty` check the changed columns, allowing some concurrent updates
- `none` do not use optimistic locking

We *very* strongly recommend that you use version/timestamp columns for optimistic locking with Hibernate. This is the optimal strategy with respect to performance and is the only strategy that correctly handles modifications made to detached instances (ie. when `Session.merge()` is used).

There is no difference between a view and a base table for a Hibernate mapping, as expected this is transparent at the database level (note that some DBMS don't support views properly, especially with updates). Sometimes you want to use a view, but can't create one in the database (ie. with a legacy schema). In this case, you can map an immutable and read-only entity to a given SQL subselect expression:

```
<class name="Summary">
    <subselect>
        select item.name, max(bid.amount), count(*)
        from item
        join bid on bid.item_id = item.id
        group by item.name
    </subselect>
    <synchronize table="item"/>
    <synchronize table="bid"/>
    <id name="name"/>
    ...
</class>
```

Declare the tables to synchronize this entity with, ensuring that auto-flush happens correctly, and that queries against the derived entity do not return stale data. The `<subselect>` is available as both as an attribute and a nested mapping element.

1.4. id

Mapped classes *must* declare the primary key column of the database table. Most classes will also have a JavaBeans-style property holding the unique identifier of an instance. The `<id>` element defines the mapping from that property to the primary key column.

```
<id
```

```
        name="propertyName"
        type="typename"
        column="column_name"
        unsaved-value="null|any|none|undefined|id_value"
        access="field|property|ClassName">
        node="element-name|@attribute-name|element/@attribute|."

        <generator class="generatorClass"/>
</id>
```

`name` (optional): The name of the identifier property.

`type` (optional): A name that indicates the Hibernate type.

`column` (optional - defaults to the property name): The name of the primary key column.

`unsaved-value` (optional - defaults to a "sensible" value): An identifier property value that indicates that an instance is newly instantiated (unsaved), distinguishing it from detached instances that were saved or loaded in a previous session.

`access` (optional - defaults to `property`): The strategy Hibernate should use for accessing the property value.

If the `name` attribute is missing, it is assumed that the class has no identifier property.

The `unsaved-value` attribute is almost never needed in Hibernate3.

There is an alternative `<composite-id>` declaration to allow access to legacy data with composite keys. We strongly discourage its use for anything else.

1.4.1. Generator

The optional `<generator>` child element names a Java class used to generate unique identifiers for instances of the persistent class. If any parameters are required to configure or initialize the generator instance, they are passed using the `<param>` element.

```
<id name="id" type="long" column="cat_id">
        <generator class="org.hibernate.id.TableHiLoGenerator">
                <param name="table">uid_table</param>
                <param name="column">next_hi_value_column</param>
        </generator>
</id>
```

All generators implement the interface `org.hibernate.id.IdentifierGenerator`. This is a very simple interface; some applications may choose to provide their own specialized implementations. However, Hibernate provides a range of built-in implementations. There are shortcut names for the built-in generators:

`increment`

generates identifiers of type `long`, `short` or `int` that are unique only when no other process is inserting data into the same table. *Do not use in a cluster.*

`identity`
supports identity columns in DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL. The returned identifier is of type `long`, `short` or `int`.

`sequence`
uses a sequence in DB2, PostgreSQL, Oracle, SAP DB, McKoi or a generator in Interbase. The returned identifier is of type `long`, `short` or `int`

`hilo`
uses a hi/lo algorithm to efficiently generate identifiers of type `long`, `short` or `int`, given a table and column (by default `hibernate_unique_key` and `next_hi` respectively) as a source of hi values. The hi/lo algorithm generates identifiers that are unique only for a particular database.

`seqhilo`
uses a hi/lo algorithm to efficiently generate identifiers of type `long`, `short` or `int`, given a named database sequence.

`uuid`
uses a 128-bit UUID algorithm to generate identifiers of type string, unique within a network (the IP address is used). The UUID is encoded as a string of hexadecimal digits of length 32.

`guid`
uses a database-generated GUID string on MS SQL Server and MySQL.

`native`
picks `identity`, `sequence` or `hilo` depending upon the capabilities of the underlying database.

`assigned`
lets the application to assign an identifier to the object before `save()` is called. This is the default strategy if no `<generator>` element is specified.

`select`
retrieves a primary key assigned by a database trigger by selecting the row by some unique key and retrieving the primary key value.

`foreign`
uses the identifier of another associated object. Usually used in conjunction with a `<one-to-one>` primary key association.

`sequence-identity`
a specialized sequence generation strategy which utilizes a database sequence for the actual value generation, but combines this with JDBC3 getGeneratedKeys to actually return the generated identifier value as part of the insert statement execution. This strategy is only known to be supported on Oracle 10g drivers targetted for JDK 1.4. Note comments on

these insert statements are disabled due to a bug in the Oracle drivers.

1.4.2. Hi/lo algorithm

The `hilo` and `seqhilo` generators provide two alternate implementations of the hi/lo algorithm, a favorite approach to identifier generation. The first implementation requires a "special" database table to hold the next available "hi" value. The second uses an Oracle-style sequence (where supported).

```
<id name="id" type="long" column="cat_id">
        <generator class="hilo">
                <param name="table">hi_value</param>
                <param name="column">next_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

```
<id name="id" type="long" column="cat_id">
        <generator class="seqhilo">
                <param name="sequence">hi_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

Unfortunately, you can't use `hilo` when supplying your own `Connection` to Hibernate. When Hibernate is using an application server datasource to obtain connections enlisted with JTA, you must properly configure the `hibernate.transaction.manager_lookup_class`.

1.4.3. UUID algorithm

The UUID contains: IP address, startup time of the JVM (accurate to a quarter second), system time and a counter value (unique within the JVM). It's not possible to obtain a MAC address or memory address from Java code, so this is the best we can do without using JNI.

1.4.4. Identity columns and sequences

For databases which support identity columns (DB2, MySQL, Sybase, MS SQL), you may use `identity` key generation. For databases that support sequences (DB2, Oracle, PostgreSQL, Interbase, McKoi, SAP DB) you may use `sequence` style key generation. Both these strategies require two SQL queries to insert a new object.

```
<id name="id" type="long" column="person_id">
        <generator class="sequence">
                <param name="sequence">person_id_sequence</param>
        </generator>
</id>
```

```
<id name="id" type="long" column="person_id" unsaved-value="0">
```

```
        <generator class="identity"/>
</id>
```

For cross-platform development, the `native` strategy will choose from the `identity`, `sequence` and `hilo` strategies, dependant upon the capabilities of the underlying database.

1.4.5. Assigned identifiers

If you want the application to assign identifiers (as opposed to having Hibernate generate them), you may use the `assigned` generator. This special generator will use the identifier value already assigned to the object's identifier property. This generator is used when the primary key is a natural key instead of a surrogate key. This is the default behavior if you do no specify a `<generator>` element.

Choosing the `assigned` generator makes Hibernate use `unsaved-value="undefined"`, forcing Hibernate to go to the database to determine if an instance is transient or detached, unless there is a version or timestamp property, or you define `Interceptor.isUnsaved()`.

1.4.6. Primary keys assigned by triggers

For legacy schemas only (Hibernate does not generate DDL with triggers).

```
<id name="id" type="long" column="person_id">
        <generator class="select">
                <param name="key">socialSecurityNumber</param>
        </generator>
</id>
```

In the above example, there is a unique valued property named `socialSecurityNumber` defined by the class, as a natural key, and a surrogate key named `person_id` whose value is generated by a trigger.

1.5. composite-id

```
<composite-id
        name="propertyName"
        class="ClassName"
        mapped="true|false"
        access="field|property|ClassName">
        node="element-name|."

        <key-property name="propertyName" type="typename"
column="column_name"/>
        <key-many-to-one name="propertyName class="ClassName"
column="column_name"/>
        ......
</composite-id>
```

For a table with a composite key, you may map multiple properties of the class as identifier properties. The `<composite-id>` element accepts `<key-property>` property mappings and `<key-many-to-one>` mappings as child elements.

```
<composite-id>
        <key-property name="medicareNumber"/>
        <key-property name="dependent"/>
</composite-id>
```

Your persistent class *must* override `equals()` and `hashCode()` to implement composite identifier equality. It must also implements `Serializable`.

Unfortunately, this approach to composite identifiers means that a persistent object is its own identifier. There is no convenient "handle" other than the object itself. You must instantiate an instance of the persistent class itself and populate its identifier properties before you can `load()` the persistent state associated with a composite key. We call this approach an *embedded* composite identifier, and discourage it for serious applications.

A second approach is what we call a *mapped* composite identifier, where the identifier properties named inside the `<composite-id>` element are duplicated on both the persistent class and a separate identifier class.

```
<composite-id class="MedicareId" mapped="true">
        <key-property name="medicareNumber"/>
        <key-property name="dependent"/>
</composite-id>
```

In this example, both the composite identifier class, `MedicareId`, and the entity class itself have properties named `medicareNumber` and `dependent`. The identifier class must override `equals()` and `hashCode()` and implement. `Serializable`. The disadvantage of this approach is quite obvious - code duplication.

The following attributes are used to specify a mapped composite identifier:

- `mapped` (optional, defaults to `false`): indicates that a mapped composite identifier is used, and that the contained property mappings refer to both the entity class and the composite identifier class.
- `class` (optional, but required for a mapped composite identifier): The class used as a composite identifier.

We will describe a third, even more convenient approach where the composite identifier is implemented as a component class in *Section 4, “Components as composite identifiers”*. The attributes described below apply only to this alternative approach:

- `name` (optional, required for this approach): A property of component type that holds the composite identifier (see chapter 9).
- `access` (optional - defaults to `property`): The strategy Hibernate should use for accessing the

property value.
- `class` (optional - defaults to the property type determined by reflection): The component class used as a composite identifier (see next section).

This third approach, an *identifier component* is the one we recommend for almost all applications.

1.6. discriminator

The `<discriminator>` element is required for polymorphic persistence using the table-per-class-hierarchy mapping strategy and declares a discriminator column of the table. The discriminator column contains marker values that tell the persistence layer what subclass to instantiate for a particular row. A restricted set of types may be used: `string`, `character`, `integer`, `byte`, `short`, `boolean`, `yes_no`, `true_false`.

```
<discriminator
        column="discriminator_column"
        type="discriminator_type"
        force="true|false"
        insert="true|false"
        formula="arbitrary sql expression"
/>
```

`column` (optional - defaults to `class`) the name of the discriminator column.

`type` (optional - defaults to `string`) a name that indicates the Hibernate type

`force` (optional - defaults to `false`) "force" Hibernate to specify allowed discriminator values even when retrieving all instances of the root class.

`insert` (optional - defaults to `true`) set this to `false` if your discriminator column is also part of a mapped composite identifier. (Tells Hibernate to not include the column in SQL `INSERT` s.)

`formula` (optional) an arbitrary SQL expression that is executed when a type has to be evaluated. Allows content-based discrimination.

Actual values of the discriminator column are specified by the `discriminator-value` attribute of the `<class>` and `<subclass>` elements.

The `force` attribute is (only) useful if the table contains rows with "extra" discriminator values that are not mapped to a persistent class. This will not usually be the case.

Using the `formula` attribute you can declare an arbitrary SQL expression that will be used to evaluate the type of a row:

```
<discriminator
    formula="case when CLASS_TYPE in ('a', 'b', 'c') then 0 else 1 end"
    type="integer"/>
```

1.7. version (optional)

The `<version>` element is optional and indicates that the table contains versioned data. This is particularly useful if you plan to use *long transactions* (see below).

```
<version
        column="version_column"
        name="propertyName"
        type="typename"
        access="field|property|ClassName"
        unsaved-value="null|negative|undefined"
        generated="never|always"
        insert="true|false"
        node="element-name|@attribute-name|element/@attribute|."
/>
```

`column` (optional - defaults to the property name): The name of the column holding the version number.

`name`: The name of a property of the persistent class.

`type` (optional - defaults to `integer`): The type of the version number.

`access` (optional - defaults to `property`): The strategy Hibernate should use for accessing the property value.

`unsaved-value` (optional - defaults to `undefined`): A version property value that indicates that an instance is newly instantiated (unsaved), distinguishing it from detached instances that were saved or loaded in a previous session. (`undefined` specifies that the identifier property value should be used.)

`generated` (optional - defaults to `never`): Specifies that this version property value is actually generated by the database. See the discussion of *Section 6, “Generated Properties”* generated properties.

`insert` (optional - defaults to `true`): Specifies whether the version column should be included in SQL insert statements. May be set to `false` if and only if the database column is defined with a default value of `0`.

Version numbers may be of Hibernate type `long`, `integer`, `short`, `timestamp` or `calendar`.

A version or timestamp property should never be null for a detached instance, so Hibernate will detect any instance with a null version or timestamp as transient, no matter what other `unsaved-value` strategies are specified. *Declaring a nullable version or timestamp property is an easy way to avoid any problems with transitive reattachment in Hibernate, especially useful for people using assigned identifiers or composite keys!*

1.8. timestamp (optional)

The optional `<timestamp>` element indicates that the table contains timestamped data. This is intended as an alternative to versioning. Timestamps are by nature a less safe implementation of optimistic locking. However, sometimes the application might use the timestamps in other ways.

```
<timestamp
        column="timestamp_column"
        name="propertyName"
        access="field|property|ClassName"
        unsaved-value="null|undefined"
        source="vm|db"
        generated="never|always"
        node="element-name|@attribute-name|element/@attribute|."
/>
```

`column` (optional - defaults to the property name): The name of a column holding the timestamp.

`name`: The name of a JavaBeans style property of Java type `Date` or `Timestamp` of the persistent class.

`access` (optional - defaults to `property`): The strategy Hibernate should use for accessing the property value.

`unsaved-value` (optional - defaults to `null`): A version property value that indicates that an instance is newly instantiated (unsaved), distinguishing it from detached instances that were saved or loaded in a previous session. (`undefined` specifies that the identifier property value should be used.)

`source` (optional - defaults to `vm`): From where should Hibernate retrieve the timestamp value? From the database, or from the current JVM? Database-based timestamps incur an overhead because Hibernate must hit the database in order to determine the "next value", but will be safer for use in clustered environments. Note also, that not all `Dialect` s are known to support retrieving of the database's current timestamp, while others might be unsafe for usage in locking due to lack of precision (Oracle 8 for example).

`generated` (optional - defaults to `never`): Specifies that this timestamp property value is actually generated by the database. See the discussion of *Section 6, “Generated Properties”* generated properties.

Note that `<timestamp>` is equivalent to `<version type="timestamp">`. And `<timestamp source="db">` is equivalent to `<version type="dbtimestamp">`

1.9. property

The `<property>` element declares a persistent, JavaBean style property of the class.

```
<property
        name="propertyName"
        column="column_name"
        type="typename"
        update="true|false"
        insert="true|false"
        formula="arbitrary SQL expression"
        access="field|property|ClassName"
        lazy="true|false"
        unique="true|false"
        not-null="true|false"
        optimistic-lock="true|false"
        generated="never|insert|always"
        node="element-name|@attribute-name|element/@attribute|."
        index="index_name"
        unique_key="unique_key_id"
        length="L"
        precision="P"
        scale="S"
/>
```

`name`: the name of the property, with an initial lowercase letter.

`column` (optional - defaults to the property name): the name of the mapped database table column. This may also be specified by nested `<column>` element(s).

`type` (optional): a name that indicates the Hibernate type.

`update, insert` (optional - defaults to `true`) : specifies that the mapped columns should be included in SQL `UPDATE` and/or `INSERT` statements. Setting both to `false` allows a pure "derived" property whose value is initialized from some other property that maps to the same colum(s) or by a trigger or other application.

`formula` (optional): an SQL expression that defines the value for a *computed* property. Computed properties do not have a column mapping of their own.

`access` (optional - defaults to `property`): The strategy Hibernate should use for accessing the property value.

`lazy` (optional - defaults to `false`): Specifies that this property should be fetched lazily when the instance variable is first accessed (requires build-time bytecode instrumentation).

`unique` (optional): Enable the DDL generation of a unique constraint for the columns. Also, allow this to be the target of a `property-ref`.

`not-null` (optional): Enable the DDL generation of a nullability constraint for the columns.

`optimistic-lock` (optional - defaults to `true`): Specifies that updates to this property do or do not require acquisition of the optimistic lock. In other words, determines if a version increment should occur when this property is dirty.

`generated` (optional - defaults to `never`): Specifies that this property value is actually generated by the database. See the discussion of *Section 6, “Generated Properties”* generated properties.

typename could be:

1. The name of a Hibernate basic type (eg. `integer, string, character, date,`

`timestamp, float, binary, serializable, object, blob`).
2. The name of a Java class with a default basic type (eg. `int, float, char, java.lang.String, java.util.Date, java.lang.Integer, java.sql.Clob`).
3. The name of a serializable Java class.
4. The class name of a custom type (eg. `com.illflow.type.MyCustomType`).

If you do not specify a type, Hibernate will use reflection upon the named property to take a guess at the correct Hibernate type. Hibernate will try to interpret the name of the return class of the property getter using rules 2, 3, 4 in that order. However, this is not always enough. In certain cases you will still need the `type` attribute. (For example, to distinguish between `Hibernate.DATE` and `Hibernate.TIMESTAMP`, or to specify a custom type.)

The `access` attribute lets you control how Hibernate will access the property at runtime. By default, Hibernate will call the property get/set pair. If you specify `access="field"`, Hibernate will bypass the get/set pair and access the field directly, using reflection. You may specify your own strategy for property access by naming a class that implements the interface `org.hibernate.property.PropertyAccessor`.

An especially powerful feature are derived properties. These properties are by definition read-only, the property value is computed at load time. You declare the computation as a SQL expression, this translates to a `SELECT` clause subquery in the SQL query that loads an instance:

```
<property name="totalPrice"
    formula="( SELECT SUM (li.quantity*p.price) FROM LineItem li, Product p
                WHERE li.productId = p.productId
                AND li.customerId = customerId
                AND li.orderNumber = orderNumber )"/>
```

Note that you can reference the entities own table by not declaring an alias on a particular column (`customerId` in the given example). Also note that you can use the nested `<formula>` mapping element if you don't like to use the attribute.

1.10. many-to-one

An ordinary association to another persistent class is declared using a `many-to-one` element. The relational model is a many-to-one association: a foreign key in one table is referencing the primary key column(s) of the target table.

```
<many-to-one
        name="propertyName"
        column="column_name"
        class="ClassName"
        cascade="cascade_style"
        fetch="join|select"
        update="true|false"
        insert="true|false"
        property-ref="propertyNameFromAssociatedClass"
        access="field|property|ClassName"
        unique="true|false"
        not-null="true|false"
        optimistic-lock="true|false"
        lazy="proxy|no-proxy|false"
        not-found="ignore|exception"
        entity-name="EntityName"
        formula="arbitrary SQL expression"
        node="element-name|@attribute-name|element/@attribute|."
        embed-xml="true|false"
        index="index_name"
        unique_key="unique_key_id"
        foreign-key="foreign_key_name"
/>
```

`name`: The name of the property.

`column` (optional): The name of the foreign key column. This may also be specified by nested `<column>` element(s).

`class` (optional - defaults to the property type determined by reflection): The name of the associated class.

`cascade` (optional): Specifies which operations should be cascaded from the parent object to the associated object.

`fetch` (optional - defaults to `select`): Chooses between outer-join fetching or sequential select fetching.

`update, insert` (optional - defaults to `true`) specifies that the mapped columns should be included in SQL `UPDATE` and/or `INSERT` statements. Setting both to `false` allows a pure "derived" association whose value is initialized from some other property that maps to the same colum(s) or by a trigger or other application.

`property-ref`: (optional) The name of a property of the associated class that is joined to this foreign key. If not specified, the primary key of the associated class is used.

`access` (optional - defaults to `property`): The strategy Hibernate should use for accessing the property value.

`unique` (optional): Enable the DDL generation of a unique constraint for the foreign-key column. Also, allow this to be the target of a `property-ref`. This makes the association multiplicity effectively one to one.

`not-null` (optional): Enable the DDL generation of a nullability constraint for the foreign key columns.

`optimistic-lock` (optional - defaults to `true`): Specifies that updates to this property do or do not require acquisition of the optimistic lock. In other words, dertermines if a version increment should occur when this property is dirty.

`lazy` (optional - defaults to `proxy`): By default, single point associations are proxied. `lazy="no-proxy"` specifies that the property should be fetched lazily when the instance variable is first accessed (requires build-time bytecode instrumentation). `lazy="false"` specifies that the association will always be eagerly fetched.

`not-found` (optional - defaults to `exception`): Specifies how foreign keys that reference missing rows will be handled: `ignore` will treat a missing row as a null association.

`entity-name` (optional): The entity name of the associated class.

`formula` (optional): an SQL expression that defines the value for a *computed* foreign key.

Setting a value of the `cascade` attribute to any meaningful value other than `none` will propagate certain operations to the associated object. The meaningful values are the names of Hibernate's basic operations, `persist, merge, delete, save-update, evict, replicate, lock, refresh`, as well as the special values `delete-orphan` and `all` and comma-separated combinations of operation names, for example, `cascade="persist,merge,evict"` or `cascade="all,delete-orphan"`. See *Section 11, “Transitive persistence”* for a full explanation. Note that single valued associations (many-to-one and one-to-one associations) do not support orphan delete.

A typical `many-to-one` declaration looks as simple as this:

```
<many-to-one name="product" class="Product" column="PRODUCT_ID"/>
```

The `property-ref` attribute should only be used for mapping legacy data where a foreign key refers to a unique key of the associated table other than the primary key. This is an ugly relational model. For example, suppose the `Product` class had a unique serial number, that is not the primary key. (The `unique` attribute controls Hibernate's DDL generation with the SchemaExport tool.)

```
<property name="serialNumber" unique="true" type="string"
column="SERIAL_NUMBER"/>
```

Then the mapping for `OrderItem` might use:

```
<many-to-one name="product" property-ref="serialNumber"
column="PRODUCT_SERIAL_NUMBER"/>
```

This is certainly not encouraged, however.

If the referenced unique key comprises multiple properties of the associated entity, you should map the referenced properties inside a named `<properties>` element.

If the referenced unique key is the property of a component, you may specify a property path:

```
<many-to-one name="owner" property-ref="identity.ssn" column="OWNER_SSN"/>
```

1.11. one-to-one

A one-to-one association to another persistent class is declared using a `one-to-one` element.

```
<one-to-one
        name="propertyName"
        class="ClassName"
        cascade="cascade_style"
        constrained="true|false"
        fetch="join|select"
        property-ref="propertyNameFromAssociatedClass"
        access="field|property|ClassName"
        formula="any SQL expression"
        lazy="proxy|no-proxy|false"
        entity-name="EntityName"
        node="element-name|@attribute-name|element/@attribute|."
        embed-xml="true|false"
        foreign-key="foreign_key_name"
/>
```

`name`: The name of the property.

`class` (optional - defaults to the property type determined by reflection): The name of the associated class.

`cascade` (optional) specifies which operations should be cascaded from the parent object to the associated object.

`constrained` (optional) specifies that a foreign key constraint on the primary key of the mapped table references the table of the associated class. This option affects the order in which `save()` and `delete()` are cascaded, and determines whether the association may be proxied (it is also used by the schema export tool).

`fetch` (optional - defaults to `select`): Chooses between outer-join fetching or sequential select fetching.

`property-ref`: (optional) The name of a property of the associated class that is joined to the primary key of this class. If not specified, the primary key of the associated class is used.

`access` (optional - defaults to `property`): The strategy Hibernate should use for accessing the property value.

`formula` (optional): Almost all one to one associations map to the primary key of the owning entity. In the rare case that this is not the case, you may specify a some other column, columns or expression to join on using an SQL formula. (See `org.hibernate.test.onetooneformula` for an example.)

`lazy` (optional - defaults to `proxy`): By default, single point associations are proxied.

`lazy="no-proxy"` specifies that the property should be fetched lazily when the instance variable is first accessed (requires build-time bytecode instrumentation). `lazy="false"` specifies that the association will always be eagerly fetched. *Note that if `constrained="false"`, proxying is impossible and Hibernate will eager fetch the association!*

`entity-name` (optional): The entity name of the associated class.

There are two varieties of one-to-one association:

- primary key associations
- unique foreign key associations

Primary key associations don't need an extra table column; if two rows are related by the association then the two table rows share the same primary key value. So if you want two objects to be related by a primary key association, you must make sure that they are assigned the same identifier value!

For a primary key association, add the following mappings to `Employee` and `Person`, respectively.

```
<one-to-one name="person" class="Person"/>
```

```
<one-to-one name="employee" class="Employee" constrained="true"/>
```

Now we must ensure that the primary keys of related rows in the PERSON and EMPLOYEE tables are equal. We use a special Hibernate identifier generation strategy called `foreign`:

```
<class name="person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="foreign">
            <param name="property">employee</param>
        </generator>
    </id>
    ...
    <one-to-one name="employee"
        class="Employee"
        constrained="true"/>
</class>
```

A newly saved instance of `Person` is then assigned the same primary key value as the `Employee` instance refered with the `employee` property of that `Person`.

Alternatively, a foreign key with a unique constraint, from `Employee` to `Person`, may be expressed as:

```
<many-to-one name="person" class="Person" column="PERSON_ID" unique="true"/>
```

And this association may be made bidirectional by adding the following to the `Person` mapping:

```
<one-to-one name"employee" class="Employee" property-ref="person"/>
```

1.12. natural-id

```
<natural-id mutable="true|false"/>
        <property ... />
        <many-to-one ... />
        ......
</natural-id>
```

Even though we recommend the use of surrogate keys as primary keys, you should still try to identify natural keys for all entities. A natural key is a property or combination of properties that is unique and non-null. If it is also immutable, even better. Map the properties of the natural key inside the `<natural-id>` element. Hibernate will generate the necessary unique key and nullability constraints, and your mapping will be more self-documenting.

We strongly recommend that you implement `equals()` and `hashCode()` to compare the natural key properties of the entity.

This mapping is not intended for use with entities with natural primary keys.

- `mutable` (optional, defaults to `false`): By default, natural identifier properties as assumed to be immutable (constant).

1.13. component, dynamic-component

The `<component>` element maps properties of a child object to columns of the table of a parent class. Components may, in turn, declare their own properties, components or collections. See "Components" below.

```
<component
        name="propertyName"
        class="className"
        insert="true|false"
        update="true|false"
        access="field|property|ClassName"
        lazy="true|false"
        optimistic-lock="true|false"
        unique="true|false"
        node="element-name|."
>

        <property ...../>
        <many-to-one .... />
        ........
</component>
```

`name`: The name of the property.

`class` (optional - defaults to the property type determined by reflection): The name of the component (child) class.

`insert`: Do the mapped columns appear in SQL `INSERT` s?

`update`: Do the mapped columns appear in SQL `UPDATE` s?

`access` (optional - defaults to `property`): The strategy Hibernate should use for accessing the property value.

`lazy` (optional - defaults to `false`): Specifies that this component should be fetched lazily when the instance variable is first accessed (requires build-time bytecode instrumentation).

`optimistic-lock` (optional - defaults to `true`): Specifies that updates to this component do or do not require acquisition of the optimistic lock. In other words, determines if a version increment should occur when this property is dirty.

`unique` (optional - defaults to `false`): Specifies that a unique constraint exists upon all mapped columns of the component.

The child `<property>` tags map properties of the child class to table columns.

The `<component>` element allows a `<parent>` subelement that maps a property of the component class as a reference back to the containing entity.

The `<dynamic-component>` element allows a `Map` to be mapped as a component, where the property names refer to keys of the map, see *Section 5, “Dynamic components”*.

1.14. properties

The `<properties>` element allows the definition of a named, logical grouping of properties of a class. The most important use of the construct is that it allows a combination of properties to be the target of a `property-ref`. It is also a convenient way to define a multi-column unique constraint.

```
<properties
        name="logicalName"
        insert="true|false"
        update="true|false"
        optimistic-lock="true|false"
        unique="true|false"
>

        <property ...../>
        <many-to-one .... />
        ........
</properties>
```

`name`: The logical name of the grouping - *not* an actual property name.

`insert`: Do the mapped columns appear in SQL `INSERT` s?

`update`: Do the mapped columns appear in SQL `UPDATE` s?

`optimistic-lock` (optional - defaults to `true`): Specifies that updates to these properties do or do not require acquisition of the optimistic lock. In other words, determines if a version increment should occur when these properties are dirty.

`unique` (optional - defaults to `false`): Specifies that a unique constraint exists upon all mapped columns of the component.

For example, if we have the following `<properties>` mapping:

```
<class name="Person">
    <id name="personNumber"/>
    ...
    <properties name="name"
            unique="true" update="false">
        <property name="firstName"/>
        <property name="initial"/>
        <property name="lastName"/>
    </properties>
</class>
```

Then we might have some legacy data association which refers to this unique key of the `Person` table, instead of to the primary key:

```
<many-to-one name="person"
         class="Person" property-ref="name">
    <column name="firstName"/>
    <column name="initial"/>
    <column name="lastName"/>
</many-to-one>
```

We don't recommend the use of this kind of thing outside the context of mapping legacy data.

1.15. subclass

Finally, polymorphic persistence requires the declaration of each subclass of the root persistent class. For the table-per-class-hierarchy mapping strategy, the `<subclass>` declaration is used.

```
<subclass
        name="ClassName"
        discriminator-value="discriminator_value"
        proxy="ProxyInterface"
        lazy="true|false"
        dynamic-update="true|false"
        dynamic-insert="true|false"
        entity-name="EntityName"
        node="element-name"
        extends="SuperclassName">
        <property .... />
        .....
</subclass>
```

`name`: The fully qualified class name of the subclass.

`discriminator-value` (optional - defaults to the class name): A value that distiguishes individual subclasses.

`proxy` (optional): Specifies a class or interface to use for lazy initializing proxies.

`lazy` (optional, defaults to `true`): Setting `lazy="false"` disables the use of lazy fetching.

Each subclass should declare its own persistent properties and subclasses. `<version>` and `<id>` properties are assumed to be inherited from the root class. Each subclass in a heirarchy must define a unique `discriminator-value`. If none is specified, the fully qualified Java class name is used.

For information about inheritance mappings, see *Chapter 10, Inheritance Mapping*.

1.16. joined-subclass

Alternatively, each subclass may be mapped to its own table (table-per-subclass mapping strategy). Inherited state is retrieved by joining with the table of the superclass. We use the `<joined-subclass>` element.

```
<joined-subclass
        name="ClassName"
        table="tablename"
        proxy="ProxyInterface"
        lazy="true|false"
        dynamic-update="true|false"
        dynamic-insert="true|false"
        schema="schema"
        catalog="catalog"
        extends="SuperclassName"
        persister="ClassName"
        subselect="SQL expression"
        entity-name="EntityName"
        node="element-name">

        <key .... >

        <property .... />
        .....
</joined-subclass>
```

`name`: The fully qualified class name of the subclass.

`table`: The name of the subclass table.

`proxy` (optional): Specifies a class or interface to use for lazy initializing proxies.

`lazy` (optional, defaults to `true`): Setting `lazy="false"` disables the use of lazy fetching.

No discriminator column is required for this mapping strategy. Each subclass must, however, declare a table column holding the object identifier using the `<key>` element. The mapping at the start of the chapter would be re-written as:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat" table="CATS">
                <id name="id" column="uid" type="long">
                        <generator class="hilo"/>
                </id>
                <property name="birthdate" type="date"/>
                <property name="color" not-null="true"/>
                <property name="sex" not-null="true"/>
                <property name="weight"/>
                <many-to-one name="mate"/>
                <set name="kittens">
                        <key column="MOTHER"/>
                        <one-to-many class="Cat"/>
                </set>
                <joined-subclass name="DomesticCat" table="DOMESTIC_CATS">
                    <key column="CAT"/>
                    <property name="name" type="string"/>
                </joined-subclass>
        </class>

        <class name="eg.Dog">
                <!-- mapping for Dog could go here -->
        </class>

</hibernate-mapping>
```

For information about inheritance mappings, see *Chapter 10, Inheritance Mapping*.

1.17. union-subclass

A third option is to map only the concrete classes of an inheritance hierarchy to tables, (the table-per-concrete-class strategy) where each table defines all persistent state of the class, including inherited state. In Hibernate, it is not absolutely necessary to explicitly map such inheritance hierarchies. You can simply map each class with a separate `<class>` declaration. However, if you wish use polymorphic associations (e.g. an association to the superclass of your hierarchy), you need to use the `<union-subclass>` mapping.

```
<union-subclass
        name="ClassName"
        table="tablename"
        proxy="ProxyInterface"
        lazy="true|false"
        dynamic-update="true|false"
        dynamic-insert="true|false"
        schema="schema"
        catalog="catalog"
        extends="SuperclassName"
        abstract="true|false"
        persister="ClassName"
        subselect="SQL expression"
        entity-name="EntityName"
        node="element-name">

        <property .... />
        .....
</union-subclass>
```

`name`: The fully qualified class name of the subclass.

`table`: The name of the subclass table.

`proxy` (optional): Specifies a class or interface to use for lazy initializing proxies.

`lazy` (optional, defaults to `true`): Setting `lazy="false"` disables the use of lazy fetching.

No discriminator column or key column is required for this mapping strategy.

For information about inheritance mappings, see *Chapter 10, Inheritance Mapping*.

1.18. join

Using the `<join>` element, it is possible to map properties of one class to several tables, when there's a 1-to-1 relationship between the tables.

```
<join
        table="tablename"
        schema="owner"
        catalog="catalog"
        fetch="join|select"
        inverse="true|false"
        optional="true|false">

        <key ... />

        <property ... />
        ...
</join>
```

`table`: The name of the joined table.

`schema` (optional): Override the schema name specified by the root `<hibernate-mapping>` element.

`catalog` (optional): Override the catalog name specified by the root `<hibernate-mapping>` element.

`fetch` (optional - defaults to `join`): If set to `join`, the default, Hibernate will use an inner join to retrieve a `<join>` defined by a class or its superclasses and an outer join for a `<join>` defined by a subclass. If set to `select` then Hibernate will use a sequential select for a `<join>` defined on a subclass, which will be issued only if a row turns out to represent an instance of the subclass. Inner joins will still be used to retrieve a `<join>` defined by the class and its superclasses.

`inverse` (optional - defaults to `false`): If enabled, Hibernate will not try to insert or update the properties defined by this join.

`optional` (optional - defaults to `false`): If enabled, Hibernate will insert a row only if the properties defined by this join are non-null and will always use an outer join to retrieve the properties.

For example, the address information for a person can be mapped to a separate table (while preserving value type semantics for all properties):

```
<class name="Person"
    table="PERSON">

    <id name="id" column="PERSON_ID">...</id>

    <join table="ADDRESS">
        <key column="ADDRESS_ID"/>
        <property name="address"/>
        <property name="zip"/>
        <property name="country"/>
    </join>
    ...
```

This feature is often only useful for legacy data models, we recommend fewer tables than classes and a fine-grained domain model. However, it is useful for switching between inheritance mapping strategies in a single hierarchy, as explained later.

1.19. key

We've seen the `<key>` element crop up a few times now. It appears anywhere the parent mapping element defines a join to a new table, and defines the foreign key in the joined table, that references the primary key of the original table.

```
<key
        column="columnname"
        on-delete="noaction|cascade"
        property-ref="propertyName"
        not-null="true|false"
        update="true|false"
        unique="true|false"
/>
```

`column` (optional): The name of the foreign key column. This may also be specified by nested `<column>` element(s).

`on-delete` (optional, defaults to `noaction`): Specifies whether the foreign key constraint has database-level cascade delete enabled.

`property-ref` (optional): Specifies that the foreign key refers to columns that are not the primary key of the orginal table. (Provided for legacy data.)

`not-null` (optional): Specifies that the foreign key columns are not nullable (this is implied whenever the foreign key is also part of the primary key).

`update` (optional): Specifies that the foreign key should never be updated (this is implied whenever the foreign key is also part of the primary key).

`unique` (optional): Specifies that the foreign key should have a unique constraint (this is implied whenever the foreign key is also the primary key).

We recommend that for systems where delete performance is important, all keys should be defined `on-delete="cascade"`, and Hibernate will use a database-level `ON CASCADE DELETE` constraint, instead of many individual `DELETE` statements. Be aware that this feature bypasses Hibernate's usual optimistic locking strategy for versioned data.

The `not-null` and `update` attributes are useful when mapping a unidirectional one to many association. If you map a unidirectional one to many to a non-nullable foreign key, you *must* declare the key column using `<key not-null="true">`.

1.20. column and formula elements

Any mapping element which accepts a `column` attribute will alternatively accept a `<column>` subelement. Likewise, `<formula>` is an alternative to the `formula` attribute.

```
<column
        name="column_name"
        length="N"
        precision="N"
        scale="N"
        not-null="true|false"
        unique="true|false"
        unique-key="multicolumn_unique_key_name"
        index="index_name"
        sql-type="sql_type_name"
        check="SQL expression"
        default="SQL expression"/>
```

```
<formula>SQL expression</formula>
```

`column` and `formula` attributes may even be combined within the same property or association mapping to express, for example, exotic join conditions.

```
<many-to-one name="homeAddress" class="Address"
        insert="false" update="false">
    <column name="person_id" not-null="true" length="10"/>
    <formula>'MAILING'</formula>
</many-to-one>
```

1.21. import

Suppose your application has two persistent classes with the same name, and you don't want to specify the fully qualified (package) name in Hibernate queries. Classes may be "imported" explicitly, rather than relying upon `auto-import="true"`. You may even import classes and interfaces that are not explicitly mapped.

```
<import class="java.lang.Object" rename="Universe"/>
```

```
<import
        class="ClassName"
        rename="ShortName"
/>
```

`class`: The fully qualified class name of of any Java class.

`rename` (optional - defaults to the unqualified class name): A name that may be used in the query language.

1.22. any

There is one further type of property mapping. The `<any>` mapping element defines a polymorphic association to classes from multiple tables. This type of mapping always requires more than one column. The first column holds the type of the associated entity. The remaining columns hold the identifier. It is impossible to specify a foreign key constraint for this kind of association, so this is most certainly not meant as the usual way of mapping (polymorphic) associations. You should use this only in very special cases (eg. audit logs, user session data, etc).

The `meta-type` attribute lets the application specify a custom type that maps database column values to persistent classes which have identifier properties of the type specified by `id-type`. You must specify the mapping from values of the meta-type to class names.

```
<any name="being" id-type="long" meta-type="string">
```

```
    <meta-value value="TBL_ANIMAL" class="Animal"/>
    <meta-value value="TBL_HUMAN" class="Human"/>
    <meta-value value="TBL_ALIEN" class="Alien"/>
    <column name="table_name"/>
    <column name="id"/>
</any>
```

```
<any
        name="propertyName"
        id-type="idtypename"
        meta-type="metatypename"
        cascade="cascade_style"
        access="field|property|ClassName"
        optimistic-lock="true|false"
>
        <meta-value ... />
        <meta-value ... />
        .....
        <column .... />
        <column .... />
        .....
</any>
```

`name`: the property name.

`id-type`: the identifier type.

`meta-type` (optional - defaults to `string`): Any type that is allowed for a discriminator mapping.

`cascade` (optional- defaults to `none`): the cascade style.

`access` (optional - defaults to `property`): The strategy Hibernate should use for accessing the property value.

`optimistic-lock` (optional - defaults to `true`): Specifies that updates to this property do or do not require acquisition of the optimistic lock. In other words, define if a version increment should occur if this property is dirty.

2. Hibernate Types

2.1. Entities and values

To understand the behaviour of various Java language-level objects with respect to the persistence service, we need to classify them into two groups:

An *entity* exists independently of any other objects holding references to the entity. Contrast this with the usual Java model where an unreferenced object is garbage collected. Entities must be explicitly saved and deleted (except that saves and deletions may be *cascaded* from a parent entity to its children). This is different from the ODMG model of object persistence by reachablity

- and corresponds more closely to how application objects are usually used in large systems. Entities support circular and shared references. They may also be versioned.

An entity's persistent state consists of references to other entities and instances of *value* types. Values are primitives, collections (not what's inside a collection), components and certain immutable objects. Unlike entities, values (in particular collections and components) *are* persisted and deleted by reachability. Since value objects (and primitives) are persisted and deleted along with their containing entity they may not be independently versioned. Values have no independent identity, so they cannot be shared by two entities or collections.

Up until now, we've been using the term "persistent class" to refer to entities. We will continue to do that. Strictly speaking, however, not all user-defined classes with persistent state are entities. A *component* is a user defined class with value semantics. A Java property of type `java.lang.String` also has value semantics. Given this definition, we can say that all types (classes) provided by the JDK have value type semantics in Java, while user-defined types may be mapped with entity or value type semantics. This decision is up to the application developer. A good hint for an entity class in a domain model are shared references to a single instance of that class, while composition or aggregation usually translates to a value type.

We'll revisit both concepts throughout the documentation.

The challenge is to map the Java type system (and the developers' definition of entities and value types) to the SQL/database type system. The bridge between both systems is provided by Hibernate: for entities we use `<class>`, `<subclass>` and so on. For value types we use `<property>`, `<component>`, etc, usually with a `type` attribute. The value of this attribute is the name of a Hibernate *mapping type*. Hibernate provides many mappings (for standard JDK value types) out of the box. You can write your own mapping types and implement your custom conversion strategies as well, as you'll see later.

All built-in Hibernate types except collections support null semantics.

2.2. Basic value types

The built-in *basic mapping types* may be roughly categorized into

`integer, long, short, float, double, character, byte, boolean, yes_no, true_false`

Type mappings from Java primitives or wrapper classes to appropriate (vendor-specific) SQL column types. `boolean, yes_no` and `true_false` are all alternative encodings for a Java `boolean` or `java.lang.Boolean`.

`string`

A type mapping from `java.lang.String` to `VARCHAR` (or Oracle `VARCHAR2`).

`date, time, timestamp`

Type mappings from `java.util.Date` and its subclasses to SQL types `DATE`, `TIME` and `TIMESTAMP` (or equivalent).

`calendar, calendar_date`

Type mappings from `java.util.Calendar` to SQL types `TIMESTAMP` and `DATE` (or equivalent).

`big_decimal, big_integer`

Type mappings from `java.math.BigDecimal` and `java.math.BigInteger` to `NUMERIC` (or Oracle `NUMBER`).

`locale, timezone, currency`

Type mappings from `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`

A type mapping from `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 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.Blob`. These 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.)

`imm_date, imm_time, imm_timestamp, imm_calendar, imm_calendar_date, imm_serializable, imm_binary`

Type mappings for what are usually considered mutable Java types, where Hibernate makes certain optimizations appropriate only for immutable Java types, and the application treats the object as immutable. For example, you should not call `Date.setTime()` for an instance mapped as `imm_timestamp`. To change the value of the property, and have that change made persistent, the application must assign a new (nonidentical) object to the property.

Unique identifiers of entities and collections may be of any basic type except `binary`, `blob` and `clob`. (Composite identifiers are also allowed, see below.)

The basic value types have corresponding `Type` constants defined on `org.hibernate.Hibernate`. For example, `Hibernate.STRING` represents the `string` type.

2.3. Custom value types

It is relatively easy for developers to create their own value types. For example, you might want to persist properties of type `java.lang.BigInteger` to `VARCHAR` columns. Hibernate does not provide a built-in type for this. But custom types are not limited to mapping a property (or collection element) to a single table column. So, for example, you might have a Java property `getName()`/`setName()` of type `java.lang.String` that is persisted to the columns `FIRST_NAME`, `INITIAL`, `SURNAME`.

To implement a custom type, implement either `org.hibernate.UserType` or `org.hibernate.CompositeUserType` and declare properties using the fully qualified classname of the type. Check out `org.hibernate.test.DoubleStringType` to see the kind of things that are possible.

```
<property name="twoStrings" type="org.hibernate.test.DoubleStringType">
    <column name="first_string"/>
    <column name="second_string"/>
</property>
```

Notice the use of `<column>` tags to map a property to multiple columns.

The `CompositeUserType`, `EnhancedUserType`, `UserCollectionType`, and `UserVersionType` interfaces provide support for more specialized uses.

You may even supply parameters to a `UserType` in the mapping file. To do this, your `UserType` must implement the `org.hibernate.usertype.ParameterizedType` interface. To supply parameters to your custom type, you can use the `<type>` element in your mapping files.

```
<property name="priority">
    <type name="com.mycompany.usertypes.DefaultValueIntegerType">
        <param name="default">0</param>
    </type>
</property>
```

The `UserType` can now retrieve the value for the parameter named `default` from the `Properties` object passed to it.

If you use a certain `UserType` very often, it may be useful to define a shorter name for it. You can do this using the `<typedef>` element. Typedefs assign a name to a custom type, and may also contain a list of default parameter values if the type is parameterized.

```
<typedef class="com.mycompany.usertypes.DefaultValueIntegerType"
name="default_zero">
    <param name="default">0</param>
</typedef>
```

```
<property name="priority" type="default_zero"/>
```

It is also possible to override the parameters supplied in a typedef on a case-by-case basis by using type parameters on the property mapping.

Even though Hibernate's rich range of built-in types and support for components means you will very rarely *need* to use a custom type, it is nevertheless considered good form to use custom types for (non-entity) classes that occur frequently in your application. For example, a `MonetaryAmount` class is a good candidate for a `CompositeUserType`, even though it could easily be mapped as a component. One motivation for this is abstraction. With a custom type, your mapping documents would be future-proofed against possible changes in your way of representing monetary values.

3. Mapping a class more than once

It is possible to provide more than one mapping for a particular persistent class. In this case you must specify an *entity name* do disambiguate between instances of the two mapped entities. (By default, the entity name is the same as the class name.) Hibernate lets you specify the entity name when working with persistent objects, when writing queries, or when mapping associations to the named entity.

```
<class name="Contract" table="Contracts"
        entity-name="CurrentContract">
    ...
    <set name="history" inverse="true"
            order-by="effectiveEndDate desc">
        <key column="currentContractId"/>
        <one-to-many entity-name="HistoricalContract"/>
    </set>
</class>

<class name="Contract" table="ContractHistory"
        entity-name="HistoricalContract">
    ...
    <many-to-one name="currentContract"
            column="currentContractId"
            entity-name="CurrentContract"/>
</class>
```

Notice how associations are now specified using `entity-name` instead of `class`.

4. SQL quoted identifiers

You may force Hibernate to quote an identifier in the generated SQL by enclosing the table or column name in backticks in the mapping document. Hibernate will use the correct quotation style for the SQL `Dialect` (usually double quotes, but brackets for SQL Server and backticks for MySQL).

```
<class name="LineItem" table="`Line Item`">
    <id name="id" column="`Item Id`"/><generator class="assigned"/></id>
    <property name="itemNumber" column="`Item #`"/>
    ...
```

```
</class>
```

5. Metadata alternatives

XML isn't for everyone, and so there are some alternative ways to define O/R mapping metadata in Hibernate.

5.1. Using XDoclet markup

Many Hibernate users prefer to embed mapping information directly in sourcecode using XDoclet `@hibernate.tags`. We will not cover this approach in this document, since strictly it is considered part of XDoclet. However, we include the following example of the `Cat` class with XDoclet mappings.

```
package eg;
import java.util.Set;
import java.util.Date;

/**
 * @hibernate.class
 *  table="CATS"
 */
public class Cat {
    private Long id; // identifier
    private Date birthdate;
    private Cat mother;
    private Set kittens
    private Color color;
    private char sex;
    private float weight;

    /*
     * @hibernate.id
     *  generator-class="native"
     *  column="CAT_ID"
     */
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id=id;
    }

    /**
     * @hibernate.many-to-one
     *  column="PARENT_ID"
     */
    public Cat getMother() {
        return mother;
    }
    void setMother(Cat mother) {
        this.mother = mother;
    }

    /**
     * @hibernate.property
     *  column="BIRTH_DATE"
     */
    public Date getBirthdate() {
        return birthdate;
    }
    void setBirthdate(Date date) {
        birthdate = date;
    }
    /**
     * @hibernate.property
     *  column="WEIGHT"
     */
    public float getWeight() {
        return weight;
    }
    void setWeight(float weight) {
        this.weight = weight;
    }

    /**
     * @hibernate.property
     *  column="COLOR"
     *  not-null="true"
     */
    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }
    /**
     * @hibernate.set
     *  inverse="true"
     *  order-by="BIRTH_DATE"
     * @hibernate.collection-key
     *  column="PARENT_ID"
     * @hibernate.collection-one-to-many
     */
    public Set getKittens() {
        return kittens;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
        kittens.add(kitten);
    }

    /**
     * @hibernate.property
     *  column="SEX"
     *  not-null="true"
     *  update="false"
     */
    public char getSex() {
        return sex;
    }
    void setSex(char sex) {
        this.sex=sex;
    }
}
```

See the Hibernate web site for more examples of XDoclet and Hibernate.

5.2. Using JDK 5.0 Annotations

JDK 5.0 introduced XDoclet-style annotations at the language level, type-safe and checked at compile time. This mechnism is more powerful than XDoclet annotations and better supported by tools and IDEs. IntelliJ IDEA, for example, supports auto-completion and syntax highlighting of JDK 5.0 annotations. The new revision of the EJB specification (JSR-220) uses JDK 5.0 annotations as the primary metadata mechanism for entity beans. Hibernate3 implements the `EntityManager` of JSR-220 (the persistence API), support for mapping metadata is available via the *Hibernate Annotations* package, as a separate download. Both EJB3 (JSR-220) and Hibernate3 metadata is supported.

This is an example of a POJO class annotated as an EJB entity bean:

```
@Entity(access = AccessType.FIELD)
public class Customer implements Serializable {

    @Id;
    Long id;

    String firstName;
    String lastName;
    Date birthday;

    @Transient
    Integer age;

    @Embedded
    private Address homeAddress;

    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumn(name="CUSTOMER_ID")
    Set<Order> orders;

    // Getter/setter and business methods
}
```

Note that support for JDK 5.0 Annotations (and JSR-220) is still work in progress and not completed. Please refer to the Hibernate Annotations module for more details.

6. Generated Properties

Generated properties are properties which have their values generated by the database. Typically, Hibernate applications needed to `refresh` objects which contain any properties for which the database was generating values. Marking properties as generated, however, lets the application delegate this responsibility to Hibernate. Essentially, whenever Hibernate issues an SQL INSERT or UPDATE for an entity which has defined generated properties, it immediately issues a select afterwards to retrieve the generated values.

Properties marked as generated must additionally be non-insertable and non-updateable. Only *Section 1.7, “version (optional)”* versions, *Section 1.8, “timestamp (optional)”* timestamps, and *Section 1.9, “property”* simple properties can be marked as generated.

`never` (the default) - means that the given property value is not generated within the database.

`insert` - states that the given property value is generated on insert, but is not regenerated on subsequent updates. Things like created-date would fall into this category. Note that even thought *Section 1.7, “version (optional)”* version and *Section 1.8, “timestamp (optional)”* timestamp properties can be marked as generated, this option is not available there...

`always` - states that the property value is generated both on insert and on update.

7. Auxiliary Database Objects

Allows CREATE and DROP of arbitrary database objects, in conjunction with Hibernate's schema evolution tools, to provide the ability to fully define a user schema within the Hibernate mapping files. Although designed specifically for creating and dropping things like triggers or stored procedures, really any SQL command that can be run via a `java.sql.Statement.execute()` method is valid here (ALTERs, INSERTS, etc). There are essentially two modes for defining auxiliary database objects...

The first mode is to explicitly list the CREATE and DROP commands out in the mapping file:

```
<hibernate-mapping>
    ...
    <database-object>
        <create>CREATE TRIGGER my_trigger ...</create>
        <drop>DROP TRIGGER my_trigger</drop>
    </database-object>
</hibernate-mapping>
```

The second mode is to supply a custom class which knows how to construct the CREATE and DROP commands. This custom class must implement the `org.hibernate.mapping.AuxiliaryDatabaseObject` interface.

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
    </database-object>
```

```
</hibernate-mapping>
```

Additionally, these database objects can be optionally scoped such that they only apply when certain dialects are used.

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
        <dialect-scope name="org.hibernate.dialect.Oracle9Dialect"/>
        <dialect-scope name="org.hibernate.dialect.OracleDialect"/>
    </database-object>
</hibernate-mapping>
```

Collection Mapping

1. Persistent collections

Hibernate requires that persistent collection-valued fields be declared as an interface type, for example:

```
public class Product {
    private String serialNumber;
    private Set parts = new HashSet();

    public Set getParts() { return parts; }
    void setParts(Set parts) { this.parts = parts; }
    public String getSerialNumber() { return serialNumber; }
    void setSerialNumber(String sn) { serialNumber = sn; }
}
```

The actual interface might be `java.util.Set`, `java.util.Collection`, `java.util.List`, `java.util.Map`, `java.util.SortedSet`, `java.util.SortedMap` or ... anything you like! (Where "anything you like" means you will have to write an implementation of `org.hibernate.usertype.UserCollectionType`.)

Notice how we initialized the instance variable with an instance of `HashSet`. This is the best way to initialize collection valued properties of newly instantiated (non-persistent) instances. When you make the instance persistent - by calling `persist()`, for example - Hibernate will actually replace the `HashSet` with an instance of Hibernate's own implementation of `Set`. Watch out for errors like this:

```
Cat cat = new DomesticCat();
Cat kitten = new DomesticCat();
....
Set kittens = new HashSet();
kittens.add(kitten);
cat.setKittens(kittens);
session.persist(cat);
kittens = cat.getKittens(); // Okay, kittens collection is a Set
(HashSet) cat.getKittens(); // Error!
```

The persistent collections injected by Hibernate behave like `HashMap`, `HashSet`, `TreeMap`, `TreeSet` or `ArrayList`, depending upon the interface type.

Collections instances have the usual behavior of value types. They are automatically persisted when referenced by a persistent object and automatically deleted when unreferenced. If a collection is passed from one persistent object to another, its elements might be moved from one table to another. Two entities may not share a reference to the same collection instance. Due to the underlying relational model, collection-valued properties do not support null value semantics; Hibernate does not distinguish between a null collection reference and an empty collection.

You shouldn't have to worry much about any of this. Use persistent collections the same way you use ordinary Java collections. Just make sure you understand the semantics of bidirectional associations (discussed later).

2. Collection mappings

The Hibernate mapping element used for mapping a collection depends upon the type of the interface. For example, a `<set>` element is used for mapping properties of type `Set`.

```
<class name="Product">
    <id name="serialNumber" column="productSerialNumber"/>
    <set name="parts">
        <key column="productSerialNumber" not-null="true"/>
        <one-to-many class="Part"/>
    </set>
</class>
```

Apart from `<set>`, there is also `<list>`, `<map>`, `<bag>`, `<array>` and `<primitive-array>` mapping elements. The `<map>` element is representative:

```
<map
    name="propertyName"
    table="table_name"
    schema="schema_name"
    lazy="true|extra|false"
    inverse="true|false"
    cascade="all|none|save-update|delete|all-delete-orphan|delete-orphan"
    sort="unsorted|natural|comparatorClass"
    order-by="column_name asc|desc"
    where="arbitrary sql where condition"
    fetch="join|select|subselect"
    batch-size="N"
    access="field|property|ClassName"
    optimistic-lock="true|false"
    mutable="true|false"
    node="element-name|."
    embed-xml="true|false"
>

    <key .... />
    <map-key .... />
    <element .... />
</map>
```

`name` the collection property name

`table` (optional - defaults to property name) the name of the collection table (not used for one-to-many associations)

`schema` (optional) the name of a table schema to override the schema declared on the root element `lazy` (optional - defaults to `true`) may be used to disable lazy fetching and specify that the association is always eagerly fetched, or to enable "extra-lazy" fetching where most operations do not initialize the collection (suitable for very large collections)

`inverse` (optional - defaults to `false`) mark this collection as the "inverse" end of a bidirectional association

`cascade` (optional - defaults to `none`) enable operations to cascade to child entities

`sort` (optional) specify a sorted collection with `natural` sort order, or a given comparator class

`order-by` (optional, JDK1.4 only) specify a table column (or columns) that define the iteration order of the `Map`, `Set` or bag, together with an optional `asc` or `desc`

`where` (optional) specify an arbitrary SQL `WHERE` condition to be used when retrieving or removing the collection (useful if the collection should contain only a subset of the available data)

`fetch` (optional, defaults to `select`) Choose between outer-join fetching, fetching by sequential select, and fetching by sequential subselect.

`batch-size` (optional, defaults to `1`) specify a "batch size" for lazily fetching instances of this collection.

`access` (optional - defaults to `property`): The strategy Hibernate should use for accessing the collection property value.

`optimistic-lock` (optional - defaults to `true`): Species that changes to the state of the collection results in increment of the owning entity's version. (For one to many associations, it is often reasonable to disable this setting.)

`mutable` (optional - defaults to `true`): A value of `false` specifies that the elements of the collection never change (a minor performance optimization in some cases).

2.1. Collection foreign keys

Collection instances are distinguished in the database by the foreign key of the entity that owns the collection. This foreign key is referred to as the *collection key column* (or columns) of the collection table. The collection key column is mapped by the `<key>` element.

There may be a nullability constraint on the foreign key column. For most collections, this is implied. For unidirectional one to many associations, the foreign key column is nullable by default, so you might need to specify `not-null="true"`.

```
<key column="productSerialNumber" not-null="true"/>
```

The foreign key constraint may use `ON DELETE CASCADE`.

```
<key column="productSerialNumber" on-delete="cascade"/>
```

See the previous chapter for a full definition of the `<key>` element.

2.2. Collection elements

Collections may contain almost any other Hibernate type, including all basic types, custom types, components, and of course, references to other entities. This is an important distinction: an object in a collection might be handled with "value" semantics (its lifecycle fully depends on the collection owner) or it might be a reference to another entity, with its own lifecycle. In the latter case, only the "link" between the two objects is considered to be state held by the collection.

The contained type is referred to as the *collection element type*. Collection elements are mapped by `<element>` or `<composite-element>`, or in the case of entity references, with `<one-to-many>` or `<many-to-many>`. The first two map elements with value semantics, the next two are used to map entity associations.

2.3. Indexed collections

All collection mappings, except those with set and bag semantics, need an *index column* in the collection table - a column that maps to an array index, or `List` index, or `Map` key. The index of a `Map` may be of any basic type, mapped with `<map-key>`, it may be an entity reference mapped with `<map-key-many-to-many>`, or it may be a composite type, mapped with `<composite-map-key>`. The index of an array or list is always of type `integer` and is mapped using the `<list-index>` element. The mapped column contains sequential integers (numbered from zero, by default).

```
<list-index
        column="column_name"
        base="0|1|..."/>
```

`column_name` (required): The name of the column holding the collection index values.

`base` (optional, defaults to `0`): The value of the index column that corresponds to the first element of the list or array.

```
<map-key
        column="column_name"
        formula="any SQL expression"
        type="type_name"
        node="@attribute-name"
        length="N"/>
```

`column` (optional): The name of the column holding the collection index values.

`formula` (optional): A SQL formula used to evaluate the key of the map.

`type` (reguired): The type of the map keys.

```
<map-key-many-to-many
        column="column_name"
        formula="any SQL expression"
        class="ClassName"
/>
```

`column` (optional): The name of the foreign key column for the collection index values.

`formula` (optional): A SQL formula used to evaluate the foreign key of the map key.

`class` (required): The entity class used as the map key.

If your table doesn't have an index column, and you still wish to use `List` as the property type, you should map the property as a Hibernate *<bag>*. A bag does not retain its order when it is retrieved from the database, but it may be optionally sorted or ordered.

There are quite a range of mappings that can be generated for collections, covering many common relational models. We suggest you experiment with the schema generation tool to get a feeling for how various mapping declarations translate to database tables.

2.4. Collections of values and many-to-many associations

Any collection of values or many-to-many association requires a dedicated *collection table* with a foreign key column or columns, *collection element column* or columns and possibly an index column or columns.

For a collection of values, we use the `<element>` tag.

```
<element
        column="column_name"
        formula="any SQL expression"
        type="typename"
        length="L"
        precision="P"
        scale="S"
        not-null="true|false"
        unique="true|false"
        node="element-name"
/>
```

`column` (optional): The name of the column holding the collection element values.

`formula` (optional): An SQL formula used to evaluate the element.

`type` (required): The type of the collection element.

A *many-to-many association* is specified using the `<many-to-many>` element.

```
<many-to-many
        column="column_name"
        formula="any SQL expression"
        class="ClassName"
        fetch="select|join"
        unique="true|false"
        not-found="ignore|exception"
        entity-name="EntityName"
        property-ref="propertyNameFromAssociatedClass"
        node="element-name"
        embed-xml="true|false"
    />
```

`column` (optional): The name of the element foreign key column.

`formula` (optional): An SQL formula used to evaluate the element foreign key value.

`class` (required): The name of the associated class.

`fetch` (optional - defaults to `join`): enables outer-join or sequential select fetching for this association. This is a special case; for full eager fetching (in a single `SELECT`) of an entity and its many-to-many relationships to other entities, you would enable `join` fetching not only of the collection itself, but also with this attribute on the `<many-to-many>` nested element.

`unique` (optional): Enable the DDL generation of a unique constraint for the foreign-key column. This makes the association multiplicity effectively one to many.

`not-found` (optional - defaults to `exception`): Specifies how foreign keys that reference missing rows will be handled: `ignore` will treat a missing row as a null association.

`entity-name` (optional): The entity name of the associated class, as an alternative to `class`.

`property-ref`: (optional) The name of a property of the associated class that is joined to this foreign key. If not specified, the primary key of the associated class is used.

Some examples, first, a set of strings:

```
<set name="names" table="person_names">
    <key column="person_id"/>
    <element column="person_name" type="string"/>
</set>
```

A bag containing integers (with an iteration order determined by the `order-by` attribute):

```
<bag name="sizes"
        table="item_sizes"
        order-by="size asc">
    <key column="item_id"/>
    <element column="size" type="integer"/>
</bag>
```

An array of entities - in this case, a many to many association:

```
<array name="addresses"
        table="PersonAddress"
        cascade="persist">
    <key column="personId"/>
    <list-index column="sortOrder"/>
    <many-to-many column="addressId" class="Address"/>
</array>
```

A map from string indices to dates:

```
<map name="holidays"
        table="holidays"
        schema="dbo"
        order-by="hol_name asc">
    <key column="id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>
```

A list of components (discussed in the next chapter):

```
<list name="carComponents"
        table="CarComponents">
    <key column="carId"/>
    <list-index column="sortOrder"/>
    <composite-element class="CarComponent">
        <property name="price"/>
        <property name="type"/>
        <property name="serialNumber" column="serialNum"/>
    </composite-element>
</list>
```

2.5. One-to-many associations

A *one to many association* links the tables of two classes via a foreign key, with no intervening collection table. This mapping loses certain semantics of normal Java collections:

- An instance of the contained entity class may not belong to more than one instance of the collection
- An instance of the contained entity class may not appear at more than one value of the collection index

An association from `Product` to `Part` requires existence of a foreign key column and possibly an index column to the `Part` table. A `<one-to-many>` tag indicates that this is a one to many association.

```
<one-to-many
        class="ClassName"
        not-found="ignore|exception"
        entity-name="EntityName"
        node="element-name"
        embed-xml="true|false"
    />
```

`class` (required): The name of the associated class.

`not-found` (optional - defaults to `exception`): Specifies how cached identifiers that reference missing rows will be handled: `ignore` will treat a missing row as a null association.

`entity-name` (optional): The entity name of the associated class, as an alternative to `class`.

Notice that the `<one-to-many>` element does not need to declare any columns. Nor is it necessary to specify the `table` name anywhere.

Very important note: If the foreign key column of a `<one-to-many>` association is declared `NOT NULL`, you must declare the `<key>` mapping `not-null="true"` or *use a bidirectional association* with the collection mapping marked `inverse="true"`. See the discussion of bidirectional associations later in this chapter.

This example shows a map of `Part` entities by name (where `partName` is a persistent property of `Part`). Notice the use of a formula-based index.

```
<map name="parts"
        cascade="all">
    <key column="productId" not-null="true"/>
    <map-key formula="partName"/>
    <one-to-many class="Part"/>
</map>
```

3. Advanced collection mappings

3.1. Sorted collections

Hibernate supports collections implementing `java.util.SortedMap` and `java.util.SortedSet`. You must specify a comparator in the mapping file:

```
<set name="aliases"
            table="person_aliases"
            sort="natural">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" sort="my.custom.HolidayComparator">
    <key column="year_id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>
```

Allowed values of the `sort` attribute are `unsorted`, `natural` and the name of a class implementing `java.util.Comparator`.

Sorted collections actually behave like `java.util.TreeSet` or `java.util.TreeMap`.

If you want the database itself to order the collection elements use the `order-by` attribute of `set`, `bag` or `map` mappings. This solution is only available under JDK 1.4 or higher (it is implemented using `LinkedHashSet` or `LinkedHashMap`). This performs the ordering in the SQL query, not in memory.

```
<set name="aliases" table="person_aliases" order-by="lower(name) asc">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" order-by="hol_date, hol_name">
    <key column="year_id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date type="date"/>
</map>
```

Note that the value of the `order-by` attribute is an SQL ordering, not a HQL ordering!

Associations may even be sorted by some arbitrary criteria at runtime using a collection `filter()`.

```
sortedUsers = s.createFilter( group.getUsers(), "order by this.name"
).list();
```

3.2. Bidirectional associations

A *bidirectional association* allows navigation from both "ends" of the association. Two kinds of bidirectional association are supported:

one-to-many
: set or bag valued at one end, single-valued at the other

many-to-many
: set or bag valued at both ends

You may specify a bidirectional many-to-many association simply by mapping two many-to-many associations to the same database table and declaring one end as *inverse* (which one is your choice, but it can not be an indexed collection).

Here's an example of a bidirectional many-to-many association; each category can have many items and each item can be in many categories:

```
<class name="Category">
    <id name="id" column="CATEGORY_ID"/>
    ...
    <bag name="items" table="CATEGORY_ITEM">
        <key column="CATEGORY_ID"/>
        <many-to-many class="Item" column="ITEM_ID"/>
    </bag>
</class>

<class name="Item">
    <id name="id" column="CATEGORY_ID"/>
    ...

    <!-- inverse end -->
    <bag name="categories" table="CATEGORY_ITEM" inverse="true">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </bag>
</class>
```

Changes made only to the inverse end of the association are *not* persisted. This means that Hibernate has two representations in memory for every bidirectional association, one link from A to B and another link from B to A. This is easier to understand if you think about the Java object model and how we create a many-to-many relationship in Java:

```

category.getItems().add(item);          // The category now "knows" about
the relationship
item.getCategories().add(category);     // The item now "knows" about the
relationship

session.persist(item);                   // The relationship won't be saved!
session.persist(category);               // The relationship will be saved
```

The non-inverse side is used to save the in-memory representation to the database.

You may define a bidirectional one-to-many association by mapping a one-to-many association to the same table column(s) as a many-to-one association and declaring the many-valued end `inverse="true"`.

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <set name="children" inverse="true">
        <key column="parent_id"/>
        <one-to-many class="Child"/>
    </set>
</class>

```

```
<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        not-null="true"/>
</class>
```

Mapping one end of an association with `inverse="true"` doesn't affect the operation of cascades, these are orthogonal concepts!

3.3. Bidirectional associations with indexed collections

A bidirectional association where one end is represented as a `<list>` or `<map>` requires special consideration. If there is a property of the child class which maps to the index column, no problem, we can continue using `inverse="true"` on the collection mapping:

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <map name="children" inverse="true">
        <key column="parent_id"/>
        <map-key column="name"
            type="string"/>
        <one-to-many class="Child"/>
    </map>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <property name="name"
        not-null="true"/>
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        not-null="true"/>
</class>
```

But, if there is no such property on the child class, we can't think of the association as truly bidirectional (there is information available at one end of the association that is not available at the other end). In this case, we can't map the collection `inverse="true"`. Instead, we could use the following mapping:

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <map name="children">
        <key column="parent_id"
            not-null="true"/>
        <map-key column="name"
            type="string"/>
        <one-to-many class="Child"/>
    </map>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        insert="false"
        update="false"
        not-null="true"/>
</class>
```

Note that in this mapping, the collection-valued end of the association is responsible for updates to the foreign key. TODO: Does this really result in some unnecessary update statements?

3.4. Ternary associations

There are three possible approaches to mapping a ternary association. One is to use a `Map` with an association as its index:

```
<map name="contracts">
    <key column="employer_id" not-null="true"/>
    <map-key-many-to-many column="employee_id" class="Employee"/>
    <one-to-many class="Contract"/>
</map>
```

```
<map name="connections">
    <key column="incoming_node_id"/>
    <map-key-many-to-many column="outgoing_node_id" class="Node"/>
    <many-to-many column="connection_id" class="Connection"/>
</map>
```

A second approach is to simply remodel the association as an entity class. This is the approach we use most commonly.

A final alternative is to use composite elements, which we will discuss later.

3.5. Using an <idbag>

If you've fully embraced our view that composite keys are a bad thing and that entities should have synthetic identifiers (surrogate keys), then you might find it a bit odd that the many to many associations and collections of values that we've shown so far all map to tables with composite keys! Now, this point is quite arguable; a pure association table doesn't seem to benefit much from a surrogate key (though a collection of composite values *might*). Nevertheless, Hibernate provides a feature that allows you to map many to many associations and collections of values to a table with a surrogate key.

The `<idbag>` element lets you map a `List` (or `Collection`) with bag semantics.

```
<idbag name="lovers" table="LOVERS">
    <collection-id column="ID" type="long">
        <generator class="sequence"/>
    </collection-id>
    <key column="PERSON1"/>
    <many-to-many column="PERSON2" class="Person" fetch="join"/>
</idbag>
```

As you can see, an `<idbag>` has a synthetic id generator, just like an entity class! A different surrogate key is assigned to each collection row. Hibernate does not provide any mechanism to discover the surrogate key value of a particular row, however.

Note that the update performance of an `<idbag>` is *much* better than a regular `<bag>`! Hibernate can locate individual rows efficiently and update or delete them individually, just like a list, map or set.

In the current implementation, the `native` identifier generation strategy is not supported for `<idbag>` collection identifiers.

4. Collection examples

The previous sections are pretty confusing. So lets look at an example. This class:

```
package eg;
import java.util.Set;

public class Parent {
    private long id;
    private Set children;

    public long getId() { return id; }
    private void setId(long id) { this.id=id; }

    private Set getChildren() { return children; }
    private void setChildren(Set children) { this.children=children; }

    ....
    ....
}
```

has a collection of `Child` instances. If each child has at most one parent, the most natural mapping is a one-to-many association:

```
<hibernate-mapping>
```

```

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children">
            <key column="parent_id"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

This maps to the following table definitions:

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255),
parent_id bigint )
alter table child add constraint childfk0 (parent_id) references parent
```

If the parent is *required*, use a bidirectional one-to-many association:

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" inverse="true">
            <key column="parent_id"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
        <many-to-one name="parent" class="Parent" column="parent_id"
not-null="true"/>
    </class>

</hibernate-mapping>
```

Notice the `NOT NULL` constraint:

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null
                     primary key,
                     name varchar(255),
                     parent_id bigint not null )
alter table child add constraint childfk0 (parent_id) references parent
```

Alternatively, if you absolutely insist that this association should be unidirectional, you can declare the `NOT NULL` constraint on the `<key>` mapping:

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children">
            <key column="parent_id" not-null="true"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

On the other hand, if a child might have multiple parents, a many-to-many association is appropriate:

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" table="childset">
            <key column="parent_id"/>
            <many-to-many class="Child" column="child_id"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

Table definitions:

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255) )
create table childset ( parent_id bigint not null,
                        child_id bigint not null,
                        primary key ( parent_id, child_id ) )
alter table childset add constraint childsetfk0 (parent_id) references
parent
alter table childset add constraint childsetfk1 (child_id) references child
```

For more examples and a complete walk-through a parent/child relationship mapping, see *Chapter 22, Example: Parent/Child*.

Even more exotic association mappings are possible, we will catalog all possibilities in the next chapter.

Association Mappings

1. Introduction

Association mappings are the often most difficult thing to get right. In this section we'll go through the canonical cases one by one, starting with unidirectional mappings, and then considering the bidirectional cases. We'll use `Person` and `Address` in all the examples.

We'll classify associations by whether or not they map to an intervening join table, and by multiplicity.

Nullable foreign keys are not considered good practice in traditional data modelling, so all our examples use not null foreign keys. This is not a requirement of Hibernate, and the mappings will all work if you drop the nullability constraints.

2. Unidirectional associations

2.1. many to one

A *unidirectional many-to-one association* is the most common kind of unidirectional association.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint
not null )
create table Address ( addressId bigint not null primary key )
```

2.2. one to one

A *unidirectional one-to-one association on a foreign key* is almost identical. The only difference is the column unique constraint.

```
<class name="Person">
```

```
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint
not
    null unique )
create table Address ( addressId bigint not null primary key )
```

A *unidirectional one-to-one association on a primary key* usually uses a special id generator. (Notice that we've reversed the direction of the association in this example.)

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property">person</param>
        </generator>
    </id>
    <one-to-one name="person" constrained="true"/>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

2.3. one to many

A *unidirectional one-to-many association on a foreign key* is a very unusual case, and is not really recommended.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses">
        <key column="personId"
            not-null="true"/>
        <one-to-many class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( addressId bigint not null primary key, personId
bigint not null )
```

We think it's better to use a join table for this kind of association.

3. Unidirectional associations with join tables

3.1. one to many

A *unidirectional one-to-many association on a join table* is much preferred. Notice that by specifying `unique="true"`, we have changed the multiplicity from many-to-many to one-to-many.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            unique="true"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId not null, addressId bigint
    not null primary key )
create table Address ( addressId bigint not null primary key )
```

3.2. many to one

A *unidirectional many-to-one association on a join table* is quite common when the association is optional.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId" unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId
bigint not null )
create table Address ( addressId bigint not null primary key )
```

3.3. one to one

A *unidirectional one-to-one association on a join table* is extremely unusual, but possible.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId"
            unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"
            unique="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId
bigint
    not null unique )
create table Address ( addressId bigint not null primary key )
```

3.4. many to many

Finally, we have a *unidirectional many-to-many association*.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not
null,
    primary key (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```

4. Bidirectional associations

4.1. one to many / many to one

A *bidirectional many-to-one association* is the most common kind of association. (This is the standard parent/child relationship.)

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true">
        <key column="addressId"/>
        <one-to-many class="Person"/>
    </set>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint
not null )
create table Address ( addressId bigint not null primary key )
```

If you use a `List` (or other indexed collection) you need to set the `key` column of the foreign key to `not null`, and let Hibernate manage the association from the collections side to maintain the index of each element (making the other side virtually inverse by setting `update="false"` and `insert="false"`):

```
<class name="Person">
   <id name="id"/>
   ...
   <many-to-one name="address"
      column="addressId"
      not-null="true"
      insert="false"
      update="false"/>
</class>

<class name="Address">
   <id name="id"/>
   ...
   <list name="people">
      <key column="addressId" not-null="true"/>
```

```
        <list-index column="peopleIdx"/>
        <one-to-many class="Person"/>
    </list>
</class>
```

It is important that you define `not-null="true"` on the `<key>` element of the collection mapping if the underlying foreign key column is `NOT NULL`. Don't only declare `not-null="true"` on a possible nested `<column>` element, but on the `<key>` element.

4.2. one to one

A *bidirectional one-to-one association on a foreign key* is quite common.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
   <one-to-one name="person"
        property-ref="address"/>
</class>
```

```

create table Person ( personId bigint not null primary key, addressId bigint
not null unique )
create table Address ( addressId bigint not null primary key )

```

A *bidirectional one-to-one association on a primary key* uses the special id generator.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <one-to-one name="address"/>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property">person</param>
        </generator>
    </id>
    <one-to-one name="person"
        constrained="true"/>
</class>
```

```

create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )

```

5. Bidirectional associations with join tables

5.1. one to many / many to one

A *bidirectional one-to-many association on a join table*. Note that the `inverse="true"` can go on either end of the association, on the collection, or on the join.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses"
        table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            unique="true"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        inverse="true"
        optional="true">
        <key column="addressId"/>
        <many-to-one name="person"
            column="personId"
            not-null="true"/>
    </join>
</class>
```

```

create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not
null primary key )
create table Address ( addressId bigint not null primary key )
```

5.2. one to one

A *bidirectional one-to-one association on a join table* is extremely unusual, but possible.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId"
            unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"
            unique="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true"
        inverse="true">
        <key column="addressId"
            unique="true"/>
        <many-to-one name="person"
            column="personId"
            not-null="true"
            unique="true"/>
    </join>
</class>
```

```

create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId
bigint
    not null unique )
create table Address ( addressId bigint not null primary key )

```

5.3. many to many

Finally, we have a *bidirectional many-to-many association*.

```
<class name="Person">
```

```
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true" table="PersonAddress">
        <key column="addressId"/>
        <many-to-many column="personId"
            class="Person"/>
    </set>
</class>
```

```

create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not
null, primary key
    (personId, addressId) )
create table Address ( addressId bigint not null primary key )

```

6. More complex association mappings

More complex association joins are *extremely* rare. Hibernate makes it possible to handle more complex situations using SQL fragments embedded in the mapping document. For example, if a table with historical account information data defines `accountNumber`, `effectiveEndDate` and `effectiveStartDate` columns, mapped as follows:

```
<properties name="currentAccountKey">
    <property name="accountNumber" type="string" not-null="true"/>
    <property name="currentAccount" type="boolean">
        <formula>case when effectiveEndDate is null then 1 else 0
end</formula>
    </property>
</properties>
<property name="effectiveEndDate" type="date"/>
<property name="effectiveStateDate" type="date" not-null="true"/>
```

Then we can map an association to the *current* instance (the one with null `effectiveEndDate`) using:

```
<many-to-one name="currentAccountInfo"
```

```
        property-ref="currentAccountKey"
        class="AccountInfo">
    <column name="accountNumber"/>
    <formula>'1'</formula>
</many-to-one>
```

In a more complex example, imagine that the association between `Employee` and `Organization` is maintained in an `Employment` table full of historical employment data. Then an association to the employee's *most recent* employer (the one with the most recent `startDate`) might be mapped this way:

```
<join>
    <key column="employeeId"/>
    <subselect>
        select employeeId, orgId
        from Employments
        group by orgId
        having startDate = max(startDate)
    </subselect>
    <many-to-one name="mostRecentEmployer"
            class="Organization"
            column="orgId"/>
</join>
```

You can get quite creative with this functionality, but it is usually more practical to handle these kinds of cases using HQL or a criteria query.

Component Mapping

The notion of a *component* is re-used in several different contexts, for different purposes, throughout Hibernate.

1. Dependent objects

A component is a contained object that is persisted as a value type, not an entity reference. The term "component" refers to the object-oriented notion of composition (not to architecture-level components). For example, you might model a person like this:

```
public class Person {
    private java.util.Date birthday;
    private Name name;
    private String key;
    public String getKey() {
        return key;
    }
    private void setKey(String key) {
        this.key=key;
    }
    public java.util.Date getBirthday() {
        return birthday;
    }
    public void setBirthday(java.util.Date birthday) {
        this.birthday = birthday;
    }
    public Name getName() {
        return name;
    }
    public void setName(Name name) {
        this.name = name;
    }
    ......
    ......
}
```

```
public class Name {
    char initial;
    String first;
    String last;
    public String getFirst() {
        return first;
    }
    void setFirst(String first) {
        this.first = first;
    }
    public String getLast() {
        return last;
    }
    void setLast(String last) {
        this.last = last;
    }
    public char getInitial() {
        return initial;
    }
    void setInitial(char initial) {
        this.initial = initial;
    }
}
```

Now `Name` may be persisted as a component of `Person`. Notice that `Name` defines getter and setter methods for its persistent properties, but doesn't need to declare any interfaces or identifier properties.

Our Hibernate mapping would look like:

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name"> <!-- class attribute optional
-->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class>
```

The person table would have the columns `pid`, `birthday`, `initial`, `first` and `last`.

Like all value types, components do not support shared references. In other words, two persons could have the same name, but the two person objects would contain two independent name ojects, only "the same" by value. The null value semantics of a component are *ad hoc*. When reloading the containing object, Hibernate will assume that if all component columns are null, then the entire component is null. This should be okay for most purposes.

The properties of a component may be of any Hibernate type (collections, many-to-one associations, other components, etc). Nested components should *not* be considered an exotic usage. Hibernate is intended to support a very fine-grained object model.

The `<component>` element allows a `<parent>` subelement that maps a property of the component class as a reference back to the containing entity.

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name" unique="true">
        <parent name="namedPerson"/> <!-- reference back to the Person -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class>
```

2. Collections of dependent objects

Collections of components are supported (eg. an array of type `Name`). Declare your component collection by replacing the `<element>` tag with a `<composite-element>` tag.

```
<set name="someNames" table="some_names" lazy="true">
    <key column="id"/>
    <composite-element class="eg.Name"> <!-- class attribute required -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </composite-element>
</set>
```

Note: if you define a `Set` of composite elements, it is very important to implement `equals()` and `hashCode()` correctly.

Composite elements may contain components but not collections. If your composite element itself contains components, use the `<nested-composite-element>` tag. This is a pretty exotic case - a collection of components which themselves have components. By this stage you should be asking yourself if a one-to-many association is more appropriate. Try remodelling the composite element as an entity - but note that even though the Java model is the same, the relational model and persistence semantics are still slightly different.

Please note that a composite element mapping doesn't support null-able properties if you're using a `<set>`. Hibernate has to use each columns value to identify a record when deleting objects (there is no separate primary key column in the composite element table), which is not possible with null values. You have to either use only not-null properties in a composite-element or choose a `<list>`, `<map>`, `<bag>` or `<idbag>`.

A special case of a composite element is a composite element with a nested `<many-to-one>` element. A mapping like this allows you to map extra columns of a many-to-many association table to the composite element class. The following is a many-to-many association from `Order` to `Item` where `purchaseDate`, `price` and `quantity` are properties of the association:

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="eg.Purchase">
            <property name="purchaseDate"/>
            <property name="price"/>
            <property name="quantity"/>
            <many-to-one name="item" class="eg.Item"/> <!-- class attribute
```

```
is optional -->
        </composite-element>
    </set>
</class>
```

Of course, there can't be a reference to the purchae on the other side, for bidirectional association navigation. Remember that components are value types and don't allow shared references. A single `Purchase` can be in the set of an `Order`, but it can't be referenced by the `Item` at the same time.

Even ternary (or quaternary, etc) associations are possible:

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="eg.OrderLine">
            <many-to-one name="purchaseDetails class="eg.Purchase"/>
            <many-to-one name="item" class="eg.Item"/>
        </composite-element>
    </set>
</class>
```

Composite elements may appear in queries using the same syntax as associations to other entities.

3. Components as Map indices

The `<composite-map-key>` element lets you map a component class as the key of a `Map`. Make sure you override `hashCode()` and `equals()` correctly on the component class.

4. Components as composite identifiers

You may use a component as an identifier of an entity class. Your component class must satisfy certain requirements:

- It must implement `java.io.Serializable`.
- It must re-implement `equals()` and `hashCode()`, consistently with the database's notion of composite key equality.

Note: in Hibernate3, the second requirement is not an absolutely hard requirement of Hibernate. But do it anyway.

You can't use an `IdentifierGenerator` to generate composite keys. Instead the application must assign its own identifiers.

Use the `<composite-id>` tag (with nested `<key-property>` elements) in place of the usual

`<id>` declaration. For example, the `OrderLine` class has a primary key that depends upon the (composite) primary key of `Order`.

```
<class name="OrderLine">

    <composite-id name="id" class="OrderLineId">
        <key-property name="lineId"/>
        <key-property name="orderId"/>
        <key-property name="customerId"/>
    </composite-id>

    <property name="name"/>

    <many-to-one name="order" class="Order"
            insert="false" update="false">
        <column name="orderId"/>
        <column name="customerId"/>
    </many-to-one>
    ....

</class>
```

Now, any foreign keys referencing the `OrderLine` table are also composite. You must declare this in your mappings for other classes. An association to `OrderLine` would be mapped like this:

```
<many-to-one name="orderLine" class="OrderLine">
<!-- the "class" attribute is optional, as usual -->
    <column name="lineId"/>
    <column name="orderId"/>
    <column name="customerId"/>
</many-to-one>
```

(Note that the `<column>` tag is an alternative to the `column` attribute everywhere.)

A `many-to-many` association to `OrderLine` also uses the composite foreign key:

```
<set name="undeliveredOrderLines">
    <key column name="warehouseId"/>
    <many-to-many class="OrderLine">
        <column name="lineId"/>
        <column name="orderId"/>
        <column name="customerId"/>
    </many-to-many>
</set>
```

The collection of `OrderLine` s in `Order` would use:

```
<set name="orderLines" inverse="true">
    <key>
        <column name="orderId"/>
        <column name="customerId"/>
    </key>
    <one-to-many class="OrderLine"/>
</set>
```

(The `<one-to-many>` element, as usual, declares no columns.)

If `OrderLine` itself owns a collection, it also has a composite foreign key.

```
<class name="OrderLine">
    ....
    ....
    <list name="deliveryAttempts">
        <key>   <!-- a collection inherits the composite key type -->
            <column name="lineId"/>
            <column name="orderId"/>
            <column name="customerId"/>
        </key>
        <list-index column="attemptId" base="1"/>
        <composite-element class="DeliveryAttempt">
            ...
        </composite-element>
    </set>
</class>
```

5. Dynamic components

You may even map a property of type `Map`:

```
<dynamic-component name="userAttributes">
    <property name="foo" column="FOO" type="string"/>
    <property name="bar" column="BAR" type="integer"/>
    <many-to-one name="baz" class="Baz" column="BAZ_ID"/>
</dynamic-component>
```

The semantics of a `<dynamic-component>` mapping are identical to `<component>`. The advantage of this kind of mapping is the ability to determine the actual properties of the bean at deployment time, just by editing the mapping document. Runtime manipulation of the mapping document is also possible, using a DOM parser. Even better, you can access (and change) Hibernate's configuration-time metamodel via the `Configuration` object.

Inheritance Mapping

1. The Three Strategies

Hibernate supports the three basic inheritance mapping strategies:

- table per class hierarchy
- table per subclass
- table per concrete class

In addition, Hibernate supports a fourth, slightly different kind of polymorphism:

- implicit polymorphism

It is possible to use different mapping strategies for different branches of the same inheritance hierarchy, and then make use of implicit polymorphism to achieve polymorphism across the whole hierarchy. However, Hibernate does not support mixing `<subclass>`, and `<joined-subclass>` and `<union-subclass>` mappings under the same root `<class>` element. It is possible to mix together the table per hierarchy and table per subclass strategies, under the the same `<class>` element, by combining the `<subclass>` and `<join>` elements (see below).

It is possible to define `subclass`, `union-subclass`, and `joined-subclass` mappings in separate mapping documents, directly beneath `hibernate-mapping`. This allows you to extend a class hierachy just by adding a new mapping file. You must specify an `extends` attribute in the subclass mapping, naming a previously mapped superclass. Note: Previously this feature made the ordering of the mapping documents important. Since Hibernate3, the ordering of mapping files does not matter when using the extends keyword. The ordering inside a single mapping file still needs to be defined as superclasses before subclasses.

```

 <hibernate-mapping>
     <subclass name="DomesticCat" extends="Cat" discriminator-value="D">
          <property name="name" type="string"/>
     </subclass>
 </hibernate-mapping>
```

1.1. Table per class hierarchy

Suppose we have an interface `Payment`, with implementors `CreditCardPayment`, `CashPayment`, `ChequePayment`. The table per hierarchy mapping would look like:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class>
```

Exactly one table is required. There is one big limitation of this mapping strategy: columns declared by the subclasses, such as `CCTYPE`, may not have `NOT NULL` constraints.

1.2. Table per subclass

A table per subclass mapping would look like:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <joined-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </joined-subclass>
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
</class>
```

Four tables are required. The three subclass tables have primary key associations to the superclass table (so the relational model is actually a one-to-one association).

1.3. Table per subclass, using a discriminator

Note that Hibernate's implementation of table per subclass requires no discriminator column. Other object/relational mappers use a different implementation of table per subclass which requires a type discriminator column in the superclass table. The approach taken by Hibernate is much more difficult to implement but arguably more correct from a relational point of view. If you would like to use a discriminator column with the table per subclass strategy, you may combine the use of `<subclass>` and `<join>`, as follow:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <join table="CREDIT_PAYMENT">
            <key column="PAYMENT_ID"/>
            <property name="creditCardType" column="CCTYPE"/>
            ...
        </join>
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        <join table="CASH_PAYMENT">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        <join table="CHEQUE_PAYMENT" fetch="select">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
</class>
```

The optional `fetch="select"` declaration tells Hibernate not to fetch the `ChequePayment` subclass data using an outer join when querying the superclass.

1.4. Mixing table per class hierarchy with table per subclass

You may even mix the table per hierarchy and table per subclass strategies using this approach:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <join table="CREDIT_PAYMENT">
            <property name="creditCardType" column="CCTYPE"/>
            ...
        </join>
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class>
```

For any of these mapping strategies, a polymorphic association to the root `Payment` class is mapped using `<many-to-one>`.

```
<many-to-one name="payment" column="PAYMENT_ID" class="Payment"/>
```

1.5. Table per concrete class

There are two ways we could go about mapping the table per concrete class strategy. The first is to use `<union-subclass>`.

```
<class name="Payment">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="sequence"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <union-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </union-subclass>
    <union-subclass name="CashPayment" table="CASH_PAYMENT">
        ...
    </union-subclass>
    <union-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        ...
    </union-subclass>
</class>
```

Three tables are involved for the subclasses. Each table defines columns for all properties of the class, including inherited properties.

The limitation of this approach is that if a property is mapped on the superclass, the column name must be the same on all subclass tables. (We might relax this in a future release of Hibernate.) The identity generator strategy is not allowed in union subclass inheritance, indeed the primary key seed has to be shared accross all unioned subclasses of a hierarchy.

If your superclass is abstract, map it with `abstract="true"`. Of course, if it is not abstract, an additional table (defaults to `PAYMENT` in the example above) is needed to hold instances of the superclass.

1.6. Table per concrete class, using implicit polymorphism

An alternative approach is to make use of implicit polymorphism:

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
</class>

<class name="CashPayment" table="CASH_PAYMENT">
    <id name="id" type="long" column="CASH_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CASH_AMOUNT"/>
    ...
</class>

<class name="ChequePayment" table="CHEQUE_PAYMENT">
    <id name="id" type="long" column="CHEQUE_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CHEQUE_AMOUNT"/>
    ...
</class>
```

Notice that nowhere do we mention the `Payment` interface explicitly. Also notice that properties of `Payment` are mapped in each of the subclasses. If you want to avoid duplication, consider using XML entities (e.g. `[ <!ENTITY allproperties SYSTEM "allproperties.xml"> ]` in the `DOCTYPE` declartion and `& allproperties;` in the mapping).

The disadvantage of this approach is that Hibernate does not generate SQL `UNION` s when performing polymorphic queries.

For this mapping strategy, a polymorphic association to `Payment` is usually mapped using `<any>`.

```
<any name="payment" meta-type="string" id-type="long">
    <meta-value value="CREDIT" class="CreditCardPayment"/>
    <meta-value value="CASH" class="CashPayment"/>
    <meta-value value="CHEQUE" class="ChequePayment"/>
    <column name="PAYMENT_CLASS"/>
    <column name="PAYMENT_ID"/>
</any>
```

1.7. Mixing implicit polymorphism with other inheritance mappings

There is one further thing to notice about this mapping. Since the subclasses are each mapped in their own `<class>` element (and since `Payment` is just an interface), each of the subclasses could easily be part of another inheritance hierarchy! (And you can still use polymorphic queries against the `Payment` interface.)

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="CREDIT_CARD" type="string"/>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
    <subclass name="MasterCardPayment" discriminator-value="MDC"/>
    <subclass name="VisaPayment" discriminator-value="VISA"/>
</class>

<class name="NonelectronicTransaction" table="NONELECTRONIC_TXN">
    <id name="id" type="long" column="TXN_ID">
        <generator class="native"/>
    </id>
    ...
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CASH_AMOUNT"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CHEQUE_AMOUNT"/>
        ...
    </joined-subclass>
</class>
```

Once again, we don't mention `Payment` explicitly. If we execute a query against the `Payment` interface - for example, `from Payment` - Hibernate automatically returns instances of `CreditCardPayment` (and its subclasses, since they also implement `Payment`), `CashPayment` and `ChequePayment` but not instances of `NonelectronicTransaction`.

2. Limitations

There are certain limitations to the "implicit polymorphism" approach to the table per concrete-class mapping strategy. There are somewhat less restrictive limitations to `<union-subclass>` mappings.

The following table shows the limitations of table per concrete-class mappings, and of implicit polymorphism, in Hibernate.

Inheritance strategy	Polymorph many-to-on	Polymorph one-to-one	Polymorph one-to-mar	Polymorph many-to-m	Polymorph `load()/get`	Polymorph queries	Polymorph joins
table per class-hierarchy	`<many-to-one>`	`<one-to-one>`	`<one-to-many>`	`<many-to-many>`	`s.get(Payment.class, id)`	`from Payment`	`from Order o`

Inheritance strategy	Polymorph many-to-on	Polymorph one-to-one	Polymorph one-to-man	Polymorph many-to-m	Polymorph `load()/get`	Polymorph queries	Polymorph joins
						`p`	`join o.payment p`
table per subclass	`<many-to-one>`	`<one-to-one>`	`<one-to-many>`	`<many-to-many>`	`s.get(Payment.class, id)`	`from Payment p`	`from Order o join o.payment p`
table per concrete-class (union-subclass)	`<many-to-one>`	`<one-to-one>`	`<one-to-many>` (for `inverse="true"` only)	`<many-to-many>`	`s.get(Payment.class, id)`	`from Payment p`	`from Order o join o.payment p`
table per concrete class (implicit polymorphism)	`<any>`	*not supported*	*not supported*	`<many-to-any>`	`s.createCriteria(Payment.class).add( Restrictions.idEq(id) ).uniqueResult()`	`from Payment p`	*not supported*

Table 10.1. Features of inheritance mappings

Working with objects

Hibernate is a full object/relational mapping solution that not only shields the developer from the details of the underlying database management system, but also offers *state management* of objects. This is, contrary to the management of SQL `statements` in common JDBC/SQL persistence layers, a very natural object-oriented view of persistence in Java applications.

In other words, Hibernate application developers should always think about the *state* of their objects, and not necessarily about the execution of SQL statements. This part is taken care of by Hibernate and is only relevant for the application developer when tuning the performance of the system.

1. Hibernate object states

Hibernate defines and supports the following object states:

- *Transient* - an object is transient if it has just been instantiated using the `new` operator, and it is not associated with a Hibernate `Session`. It has no persistent representation in the database and no identifier value has been assigned. Transient instances will be destroyed by the garbage collector if the application doesn't hold a reference anymore. Use the Hibernate `Session` to make an object persistent (and let Hibernate take care of the SQL statements that need to be executed for this transition).
- *Persistent* - a persistent instance has a representation in the database and an identifier value. It might just have been saved or loaded, however, it is by definition in the scope of a `Session`. Hibernate will detect any changes made to an object in persistent state and synchronize the state with the database when the unit of work completes. Developers don't execute manual `UPDATE` statements, or `DELETE` statements when an object should be made transient.
- *Detached* - a detached instance is an object that has been persistent, but its `Session` has been closed. The reference to the object is still valid, of course, and the detached instance might even be modified in this state. A detached instance can be reattached to a new `Session` at a later point in time, making it (and all the modifications) persistent again. This feature enables a programming model for long running units of work that require user think-time. We call them *application transactions*, i.e. a unit of work from the point of view of the user.

We'll now discuss the states and state transitions (and the Hibernate methods that trigger a transition) in more detail.

2. Making objects persistent

Newly instantiated instances of a a persistent class are considered *transient* by Hibernate. We can make a transient instance *persistent* by associating it with a session:

```
DomesticCat fritz = new DomesticCat();
```

```
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
Long generatedId = (Long) sess.save(fritz);
```

If `Cat` has a generated identifier, the identifier is generated and assigned to the `cat` when `save()` is called. If `Cat` has an `assigned` identifier, or a composite key, the identifier should be assigned to the `cat` instance before calling `save()`. You may also use `persist()` instead of `save()`, with the semantics defined in the EJB3 early draft.

Alternatively, you may assign the identifier using an overloaded version of `save()`.

```
DomesticCat pk = new DomesticCat();
pk.setColor(Color.TABBY);
pk.setSex('F');
pk.setName("PK");
pk.setKittens( new HashSet() );
pk.addKitten(fritz);
sess.save( pk, new Long(1234) );
```

If the object you make persistent has associated objects (e.g. the `kittens` collection in the previous example), these objects may be made persistent in any order you like unless you have a `NOT NULL` constraint upon a foreign key column. There is never a risk of violating foreign key constraints. However, you might violate a `NOT NULL` constraint if you `save()` the objects in the wrong order.

Usually you don't bother with this detail, as you'll very likely use Hibernate's *transitive persistence* feature to save the associated objects automatically. Then, even `NOT NULL` constraint violations don't occur - Hibernate will take care of everything. Transitive persistence is discussed later in this chapter.

3. Loading an object

The `load()` methods of `Session` gives you a way to retrieve a persistent instance if you already know its identifier. `load()` takes a class object and will load the state into a newly instantiated instance of that class, in persistent state.

```
Cat fritz = (Cat) sess.load(Cat.class, generatedId);
```

```
// you need to wrap primitive identifiers
long id = 1234;
DomesticCat pk = (DomesticCat) sess.load( DomesticCat.class, new Long(id) );
```

Alternatively, you can load state into a given instance:

```
Cat cat = new DomesticCat();
```

```
// load pk's state into cat
sess.load( cat, new Long(pkId) );
Set kittens = cat.getKittens();
```

Note that `load()` will throw an unrecoverable exception if there is no matching database row. If the class is mapped with a proxy, `load()` just returns an uninitialized proxy and does not actually hit the database until you invoke a method of the proxy. This behaviour is very useful if you wish to create an association to an object without actually loading it from the database. It also allows multiple instances to be loaded as a batch if `batch-size` is defined for the class mapping.

If you are not certain that a matching row exists, you should use the `get()` method, which hits the database immediately and returns null if there is no matching row.

```
Cat cat = (Cat) sess.get(Cat.class, id);
if (cat==null) {
    cat = new Cat();
    sess.save(cat, id);
}
return cat;
```

You may even load an object using an SQL `SELECT ... FOR UPDATE`, using a `LockMode`. See the API documentation for more information.

```
Cat cat = (Cat) sess.get(Cat.class, id, LockMode.UPGRADE);
```

Note that any associated instances or contained collections are *not* selected `FOR UPDATE`, unless you decide to specify `lock` or `all` as a cascade style for the association.

It is possible to re-load an object and all its collections at any time, using the `refresh()` method. This is useful when database triggers are used to initialize some of the properties of the object.

```
sess.save(cat);
sess.flush(); //force the SQL INSERT
sess.refresh(cat); //re-read the state (after the trigger executes)
```

An important question usually appears at this point: How much does Hibernate load from the database and how many SQL `SELECT` s will it use? This depends on the *fetching strategy* and is explained in *Section 1, “Fetching strategies”*.

4. Querying

If you don't know the identifiers of the objects you are looking for, you need a query. Hibernate supports an easy-to-use but powerful object oriented query language (HQL). For programmatic query creation, Hibernate supports a sophisticated Criteria and Example query feature (QBC and QBE). You may also express your query in the native SQL of your database, with optional support from Hibernate for result set conversion into objects.

4.1. Executing queries

HQL and native SQL queries are represented with an instance of `org.hibernate.Query`. This interface offers methods for parameter binding, result set handling, and for the execution of the actual query. You always obtain a `Query` using the current `Session`:

```
List cats = session.createQuery(
    "from Cat as cat where cat.birthdate < ?")
    .setDate(0, date)
    .list();

List mothers = session.createQuery(
    "select mother from Cat as cat join cat.mother as mother where cat.name
= ?")
    .setString(0, name)
    .list();

List kittens = session.createQuery(
    "from Cat as cat where cat.mother = ?")
    .setEntity(0, pk)
    .list();

Cat mother = (Cat) session.createQuery(
    "select cat.mother from Cat as cat where cat = ?")
    .setEntity(0, izi)
    .uniqueResult();]]

Query mothersWithKittens = (Cat) session.createQuery(
    "select mother from Cat as mother left join fetch mother.kittens");
Set uniqueMothers = new HashSet(mothersWithKittens.list());
```

A query is usually executed by invoking `list()`, the result of the query will be loaded completely into a collection in memory. Entity instances retrieved by a query are in persistent state. The `uniqueResult()` method offers a shortcut if you know your query will only return a single object. Note that queries that make use of eager fetching of collections usually return duplicates of the root objects (but with their collections initialized). You can filter these duplicates simply through a `Set`.

4.1.1. Iterating results

Occasionally, you might be able to achieve better performance by executing the query using the `iterate()` method. This will only usually be the case if you expect that the actual entity instances returned by the query will already be in the session or second-level cache. If they are not already cached, `iterate()` will be slower than `list()` and might require many database hits for a simple query, usually *1* for the initial select which only returns identifiers, and *n* additional selects to initialize the actual instances.

```
// fetch ids
Iterator iter = sess.createQuery("from eg.Qux q order by
q.likeliness").iterate();
while ( iter.hasNext() ) {
    Qux qux = (Qux) iter.next();  // fetch the object
    // something we couldnt express in the query
    if ( qux.calculateComplicatedAlgorithm() ) {
        // delete the current instance
        iter.remove();
        // dont need to process the rest
        break;
    }
}
```

4.1.2. Queries that return tuples

Hibernate queries sometimes return tuples of objects, in which case each tuple is returned as an array:

```
Iterator kittensAndMothers = sess.createQuery(
            "select kitten, mother from Cat kitten join kitten.mother
mother")
            .list()
            .iterator();

while ( kittensAndMothers.hasNext() ) {
    Object[] tuple = (Object[]) kittensAndMothers.next();
    Cat kitten  = tuple[0];
    Cat mother  = tuple[1];
    ....
}
```

4.1.3. Scalar results

Queries may specify a property of a class in the `select` clause. They may even call SQL aggregate functions. Properties or aggregates are considered "scalar" results (and not entities in persistent state).

```
Iterator results = sess.createQuery(
        "select cat.color, min(cat.birthdate), count(cat) from Cat cat " +
        "group by cat.color")
        .list()
        .iterator();

while ( results.hasNext() ) {
    Object[] row = (Object[]) results.next();
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    .....
}
```

4.1.4. Bind parameters

Methods on `Query` are provided for binding values to named parameters or JDBC-style `?` parameters. *Contrary to JDBC, Hibernate numbers parameters from zero.* Named parameters are identifiers of the form `:name` in the query string. The advantages of named parameters are:

- named parameters are insensitive to the order they occur in the query string
- they may occur multiple times in the same query
- they are self-documenting

```
//named parameter (preferred)
Query q = sess.createQuery("from DomesticCat cat where cat.name = :name");
q.setString("name", "Fritz");
Iterator cats = q.iterate();
```

```
//positional parameter
Query q = sess.createQuery("from DomesticCat cat where cat.name = ?");
q.setString(0, "Izi");
Iterator cats = q.iterate();
```

```
//named parameter list
List names = new ArrayList();
names.add("Izi");
names.add("Fritz");
Query q = sess.createQuery("from DomesticCat cat where cat.name in
(:namesList)");
q.setParameterList("namesList", names);
List cats = q.list();
```

4.1.5. Pagination

If you need to specify bounds upon your result set (the maximum number of rows you want to retrieve and / or the first row you want to retrieve) you should use methods of the `Query` interface:

```
Query q = sess.createQuery("from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.list();
```

Hibernate knows how to translate this limit query into the native SQL of your DBMS.

4.1.6. Scrollable iteration

If your JDBC driver supports scrollable `ResultSet` s, the `Query` interface may be used to obtain a `ScrollableResults` object, which allows flexible navigation of the query results.

```
Query q = sess.createQuery("select cat.name, cat from DomesticCat cat " +
                            "order by cat.name");
ScrollableResults cats = q.scroll();
if ( cats.first() ) {

    // find the first name on each page of an alphabetical list of cats by
name
    firstNamesOfPages = new ArrayList();
    do {
        String name = cats.getString(0);
        firstNamesOfPages.add(name);
    }
    while ( cats.scroll(PAGE_SIZE) );

    // Now get the first page of cats
    pageOfCats = new ArrayList();
    cats.beforeFirst();
    int i=0;
    while( ( PAGE_SIZE > i++ ) && cats.next() ) pageOfCats.add( cats.get(1)
);

}
cats.close()
```

Note that an open database connection (and cursor) is required for this functionality, use `setMaxResult()`/`setFirstResult()` if you need offline pagination functionality.

4.1.7. Externalizing named queries

You may also define named queries in the mapping document. (Remember to use a `CDATA` section if your query contains characters that could be interpreted as markup.)

```
<query name="ByNameAndMaximumWeight"><![CDATA[
    from eg.DomesticCat as cat
        where cat.name = ?
        and cat.weight > ?
] ]></query>
```

Parameter binding and executing is done programatically:

```
Query q = sess.getNamedQuery("ByNameAndMaximumWeight");
q.setString(0, name);
q.setInt(1, minWeight);
List cats = q.list();
```

Note that the actual program code is independent of the query language that is used, you may also define native SQL queries in metadata, or migrate existing queries to Hibernate by placing them in mapping files.

Also note that a query declaration inside a `<hibernate-mapping>` element requires a global unique name for the query, while a query declaration inside a `<class>` element is made unique automatically by prepending the fully qualified name of the class, for example `eg.Cat.ByNameAndMaximumWeight`.

4.2. Filtering collections

A collection *filter* is a special type of query that may be applied to a persistent collection or array. The query string may refer to `this`, meaning the current collection element.

```
Collection blackKittens = session.createFilter(
    pk.getKittens(),
    "where this.color = ?")
    .setParameter( Color.BLACK, Hibernate.custom(ColorUserType.class) )
    .list()
);
```

The returned collection is considered a bag, and it's a copy of the given collection. The original collection is not modified (this is contrary to the implication of the name "filter", but consistent with expected behavior).

Observe that filters do not require a `from` clause (though they may have one if required). Filters are not limited to returning the collection elements themselves.

```
Collection blackKittenMates = session.createFilter(
    pk.getKittens(),
    "select this.mate where this.color = eg.Color.BLACK.intValue")
    .list();
```

Even an empty filter query is useful, e.g. to load a subset of elements in a huge collection:

```
Collection tenKittens = session.createFilter(
    mother.getKittens(), "")
    .setFirstResult(0).setMaxResults(10)
    .list();
```

4.3. Criteria queries

HQL is extremely powerful but some developers prefer to build queries dynamically, using an object-oriented API, rather than building query strings. Hibernate provides an intuitive `Criteria` query API for these cases:

```
Criteria crit = session.createCriteria(Cat.class);
crit.add( Expression.eq( "color", eg.Color.BLACK ) );
crit.setMaxResults(10);
List cats = crit.list();
```

The `Criteria` and the associated `Example` API are discussed in more detail in *Chapter 16,* *Criteria Queries*.

4.4. Queries in native SQL

You may express a query in SQL, using `createSQLQuery()` and let Hibernate take care of the mapping from result sets to objects. Note that you may at any time call `session.connection()` and use the JDBC `Connection` directly. If you chose to use the Hibernate API, you must enclose SQL aliases in braces:

```
List cats = session.createSQLQuery(
    "SELECT {cat.*} FROM CAT {cat} WHERE ROWNUM<10",
    "cat",
    Cat.class
).list();
```

```
List cats = session.createSQLQuery(
    "SELECT {cat}.ID AS {cat.id}, {cat}.SEX AS {cat.sex}, " +
           "{cat}.MATE AS {cat.mate}, {cat}.SUBCLASS AS {cat.class}, ... " +
    "FROM CAT {cat} WHERE ROWNUM<10",
    "cat",
    Cat.class
).list()
```

SQL queries may contain named and positional parameters, just like Hibernate queries. More information about native SQL queries in Hibernate can be found in *Chapter 17, Native SQL*.

5. Modifying persistent objects

Transactional persistent instances (ie. objects loaded, saved, created or queried by the `Session`) may be manipulated by the application and any changes to persistent state will be persisted when the `Session` is *flushed* (discussed later in this chapter). There is no need to call a particular method (like `update()`, which has a different purpose) to make your modifications persistent. So the most straightforward way to update the state of an object is to `load()` it, and then manipulate it directly, while the `Session` is open:

```
DomesticCat cat = (DomesticCat) sess.load( Cat.class, new Long(69) );
cat.setName("PK");
sess.flush();  // changes to cat are automatically detected and persisted
```

Sometimes this programming model is inefficient since it would require both an SQL `SELECT` (to load an object) and an SQL `UPDATE` (to persist its updated state) in the same session. Therefore Hibernate offers an alternate approach, using detached instances.

Note that Hibernate does not offer its own API for direct execution of `UPDATE` or `DELETE` statements. Hibernate is a state management *service, you don't have to think in* statements *to use it. JDBC is a perfect API for executing SQL statements, you can get a JDBC `Connection` at any time by calling `session.connection()`. Furthermore, the notion of mass operations* *conflicts with object/relational mapping for online transaction processing-oriented applications. Future versions of Hibernate may however provide special mass operation functions. See Chapter 14, Batch processing for some possible batch operation tricks.*

6. Modifying detached objects

Many applications need to retrieve an object in one transaction, send it to the UI layer for manipulation, then save the changes in a new transaction. Applications that use this kind of approach in a high-concurrency environment usually use versioned data to ensure isolation for the "long" unit of work.

Hibernate supports this model by providing for reattachment of detached instances using the `Session.update()` or `Session.merge()` methods:

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catId);
Cat potentialMate = new Cat();
firstSession.save(potentialMate);

// in a higher layer of the application
cat.setMate(potentialMate);

// later, in a new session
secondSession.update(cat);  // update cat
secondSession.update(mate); // update mate
```

If the `Cat` with identifier `catId` had already been loaded by `secondSession` when the application tried to reattach it, an exception would have been thrown.

Use `update()` if you are sure that the session does not contain an already persistent instance with the same identifier, and `merge()` if you want to merge your modifications at any time without consideration of the state of the session. In other words, `update()` is usually the first method you would call in a fresh session, ensuring that reattachment of your detached instances is the first operation that is executed.

The application should individually `update()` detached instances reachable from the given detached instance if and *only* if it wants their state also updated. This can be automated of course, using *transitive persistence*, see *Section 11, “Transitive persistence”*.

The `lock()` method also allows an application to reassociate an object with a new session. However, the detached instance has to be unmodified!

```
//just reassociate:
sess.lock(fritz, LockMode.NONE);
//do a version check, then reassociate:
sess.lock(izi, LockMode.READ);
//do a version check, using SELECT ... FOR UPDATE, then reassociate:
sess.lock(pk, LockMode.UPGRADE);
```

Note that `lock()` can be used with various `LockMode` s, see the API documentation and the chapter on transaction handling for more information. Reattachment is not the only usecase for `lock()`.

Other models for long units of work are discussed in *Section 3, “Optimistic concurrency control”*.

7. Automatic state detection

Hibernate users have requested a general purpose method that either saves a transient instance by generating a new identifier or updates/reattaches the detached instances associated with its current identifier. The `saveOrUpdate()` method implements this functionality.

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catID);

// in a higher tier of the application
Cat mate = new Cat();
cat.setMate(mate);

// later, in a new session
secondSession.saveOrUpdate(cat);   // update existing state (cat has a
non-null id)
secondSession.saveOrUpdate(mate);  // save the new instance (mate has a null
id)
```

The usage and semantics of `saveOrUpdate()` seems to be confusing for new users. Firstly, so long as you are not trying to use instances from one session in another new session, you should not need to use `update()`, `saveOrUpdate()`, or `merge()`. Some whole applications will never use either of these methods.

Usually `update()` or `saveOrUpdate()` are used in the following scenario:

- the application loads an object in the first session
- the object is passed up to the UI tier
- some modifications are made to the object
- the object is passed back down to the business logic tier
- the application persists these modifications by calling `update()` in a second session

`saveOrUpdate()` does the following:

- if the object is already persistent in this session, do nothing
- if another object associated with the session has the same identifier, throw an exception
- if the object has no identifier property, `save()` it
- if the object's identifier has the value assigned to a newly instantiated object, `save()` it
- if the object is versioned (by a `<version>` or `<timestamp>`), and the version property value is the same value assigned to a newly instantiated object, `save()` it
- otherwise `update()` the object

and `merge()` is very different:

- if there is a persistent instance with the same identifier currently associated with the session, copy the state of the given object onto the persistent instance
- if there is no persistent instance currently associated with the session, try to load it from the database, or create a new persistent instance
- the persistent instance is returned
- the given instance does not become associated with the session, it remains detached

8. Deleting persistent objects

`Session.delete()` will remove an object's state from the database. Of course, your application might still hold a reference to a deleted object. It's best to think of `delete()` as making a persistent instance transient.

```
sess.delete(cat);
```

You may delete objects in any order you like, without risk of foreign key constraint violations. It is still possible to violate a `NOT NULL` constraint on a foreign key column by deleting objects in the wrong order, e.g. if you delete the parent, but forget to delete the children.

9. Replicating object between two different datastores

It is occasionally useful to be able to take a graph of persistent instances and make them persistent in a different datastore, without regenerating identifier values.

```
//retrieve a cat from one database
Session session1 = factory1.openSession();
Transaction tx1 = session1.beginTransaction();
Cat cat = session1.get(Cat.class, catId);
tx1.commit();
session1.close();

//reconcile with a second database
Session session2 = factory2.openSession();
Transaction tx2 = session2.beginTransaction();
session2.replicate(cat, ReplicationMode.LATEST_VERSION);
tx2.commit();
session2.close();
```

The `ReplicationMode` determines how `replicate()` will deal with conflicts with existing rows in the database.

- `ReplicationMode.IGNORE` - ignore the object when there is an existing database row with the same identifier
- `ReplicationMode.OVERWRITE` - overwrite any existing database row with the same identifier
- `ReplicationMode.EXCEPTION` - throw an exception if there is an existing database row with

the same identifier

- `ReplicationMode.LATEST_VERSION` - overwrite the row if its version number is earlier than the version number of the object, or ignore the object otherwise

Usecases for this feature include reconciling data entered into different database instances, upgrading system configuration information during product upgrades, rolling back changes made during non-ACID transactions and more.

10. Flushing the Session

From time to time the `Session` will execute the SQL statements needed to synchronize the JDBC connection's state with the state of objects held in memory. This process, *flush*, occurs by default at the following points

- before some query executions
- from `org.hibernate.Transaction.commit()`
- from `Session.flush()`

The SQL statements are issued in the following order

1. all entity insertions, in the same order the corresponding objects were saved using `Session.save()`
2. all entity updates
3. all collection deletions
4. all collection element deletions, updates and insertions
5. all collection insertions
6. all entity deletions, in the same order the corresponding objects were deleted using `Session.delete()`

(An exception is that objects using `native` ID generation are inserted when they are saved.)

Except when you explicity `flush()`, there are absolutely no guarantees about *when* the `Session` executes the JDBC calls, only the *order* in which they are executed. However, Hibernate does guarantee that the `Query.list(..)` will never return stale data; nor will they return the wrong data.

It is possible to change the default behavior so that flush occurs less frequently. The `FlushMode` class defines three different modes: only flush at commit time (and only when the Hibernate `Transaction` API is used), flush automatically using the explained routine, or never flush unless `flush()` is called explicitly. The last mode is useful for long running units of work, where a `Session` is kept open and disconnected for a long time (see *Section 3.2, “Extended session and automatic versioning”*).

```
sess = sf.openSession();
Transaction tx = sess.beginTransaction();
sess.setFlushMode(FlushMode.COMMIT); // allow queries to return stale state

Cat izi = (Cat) sess.load(Cat.class, id);
```

```
izi.setName(iznizi);

// might return stale data
sess.find("from Cat as cat left outer join cat.kittens kitten");

// change to izi is not flushed!
...
tx.commit(); // flush occurs
sess.close();
```

During flush, an exception might occur (e.g. if a DML operation violates a constraint). Since handling exceptions involves some understanding of Hibernate's transactional behavior, we discuss it in *Chapter 12, Transactions And Concurrency*.

11. Transitive persistence

It is quite cumbersome to save, delete, or reattach individual objects, especially if you deal with a graph of associated objects. A common case is a parent/child relationship. Consider the following example:

If the children in a parent/child relationship would be value typed (e.g. a collection of addresses or strings), their lifecycle would depend on the parent and no further action would be required for convenient "cascading" of state changes. When the parent is saved, the value-typed child objects are saved as well, when the parent is deleted, the children will be deleted, etc. This even works for operations such as the removal of a child from the collection; Hibernate will detect this and, since value-typed objects can't have shared references, delete the child from the database.

Now consider the same scenario with parent and child objects being entities, not value-types (e.g. categories and items, or parent and child cats). Entities have their own lifecycle, support shared references (so removing an entity from the collection does not mean it can be deleted), and there is by default no cascading of state from one entity to any other associated entities. Hibernate does not implement *persistence by reachability* by default.

For each basic operation of the Hibernate session - including `persist()`, `merge()`, `saveOrUpdate()`, `delete()`, `lock()`, `refresh()`, `evict()`, `replicate()` - there is a corresponding cascade style. Respectively, the cascade styles are named `create`, `merge`, `save-update`, `delete`, `lock`, `refresh`, `evict`, `replicate`. If you want an operation to be cascaded along an association, you must indicate that in the mapping document. For example:

```
<one-to-one name="person" cascade="persist"/>
```

Cascade styles my be combined:

```
<one-to-one name="person" cascade="persist,delete,lock"/>
```

You may even use `cascade="all"` to specify that *all* operations should be cascaded along the association. The default `cascade="none"` specifies that no operations are to be cascaded.

A special cascade style, `delete-orphan`, applies only to one-to-many associations, and indicates that the `delete()` operation should be applied to any child object that is removed from the association.

Recommendations:

- It doesn't usually make sense to enable cascade on a `<many-to-one>` or `<many-to-many>` association. Cascade is often useful for `<one-to-one>` and `<one-to-many>` associations.
- If the child object's lifespan is bounded by the lifespan of the of the parent object make it a *lifecycle object* by specifying `cascade="all,delete-orphan"`.
- Otherwise, you might not need cascade at all. But if you think that you will often be working with the parent and children together in the same transaction, and you want to save yourself some typing, consider using `cascade="persist,merge,save-update"`.

Mapping an association (either a single valued association, or a collection) with `cascade="all"` marks the association as a *parent/child* style relationship where save/update/delete of the parent results in save/update/delete of the child or children.

Futhermore, a mere reference to a child from a persistent parent will result in save/update of the child. This metaphor is incomplete, however. A child which becomes unreferenced by its parent is *not* automatically deleted, except in the case of a `<one-to-many>` association mapped with `cascade="delete-orphan"`. The precise semantics of cascading operations for a parent/child relationship are as follows:

- If a parent is passed to `persist()`, all children are passed to `persist()`
- If a parent is passed to `merge()`, all children are passed to `merge()`
- If a parent is passed to `save()`, `update()` or `saveOrUpdate()`, all children are passed to `saveOrUpdate()`
- If a transient or detached child becomes referenced by a persistent parent, it is passed to `saveOrUpdate()`
- If a parent is deleted, all children are passed to `delete()`
- If a child is dereferenced by a persistent parent, *nothing special happens* - the application should explicitly delete the child if necessary - unless `cascade="delete-orphan"`, in which case the "orphaned" child is deleted.

Finally, note that cascading of operations can be applied to an object graph at *call time* or at *flush time*. All operations, if enabled, are cascaded to associated entities reachable when the operation is executed. However, `save-upate` and `delete-orphan` are transitive for all associated entities reachable during flush of the `Session`.

12. Using metadata

Hibernate requires a very rich meta-level model of all entity and value types. From time to time, this model is very useful to the application itself. For example, the application might use

Hibernate's metadata to implement a "smart" deep-copy algorithm that understands which objects should be copied (eg. mutable value types) and which should not (eg. immutable value types and, possibly, associated entities).

Hibernate exposes metadata via the `ClassMetadata` and `CollectionMetadata` interfaces and the `Type` hierarchy. Instances of the metadata interfaces may be obtained from the `SessionFactory`.

```
Cat fritz = ......;
ClassMetadata catMeta = sessionfactory.getClassMetadata(Cat.class);

Object[] propertyValues = catMeta.getPropertyValues(fritz);
String[] propertyNames = catMeta.getPropertyNames();
Type[] propertyTypes = catMeta.getPropertyTypes();

// get a Map of all properties which are not collections or associations
Map namedValues = new HashMap();
for ( int i=0; i<propertyNames.length; i++ ) {
    if ( !propertyTypes[i].isEntityType() &&
!propertyTypes[i].isCollectionType() ) {
        namedValues.put( propertyNames[i], propertyValues[i] );
    }
}
```

Chapter 12.

Transactions And Concurrency

The most important point about Hibernate and concurrency control is that it is very easy to understand. Hibernate directly uses JDBC connections and JTA resources without adding any additional locking behavior. We highly recommend you spend some time with the JDBC, ANSI, and transaction isolation specification of your database management system.

Hibernate does not lock objects in memory. Your application can expect the behavior as defined by the isolation level of your database transactions. Note that thanks to the `Session`, which is also a transaction-scoped cache, Hibernate provides repeatable reads for lookup by identifier and entity queries (not reporting queries that return scalar values).

In addition to versioning for automatic optimistic concurrency control, Hibernate also offers a (minor) API for pessimistic locking of rows, using the `SELECT FOR UPDATE` syntax. Optimistic concurrency control and this API are discussed later in this chapter.

We start the discussion of concurrency control in Hibernate with the granularity of `Configuration`, `SessionFactory`, and `Session`, as well as database transactions and long conversations.

1. Session and transaction scopes

A `SessionFactory` is an expensive-to-create, threadsafe object intended to be shared by all application threads. It is created once, usually on application startup, from a `Configuration` instance.

A `Session` is an inexpensive, non-threadsafe object that should be used once, for a single request, a conversation, single unit of work, and then discarded. A `Session` will not obtain a JDBC `Connection` (or a `Datasource`) unless it is needed, hence consume no resources until used.

To complete this picture you also have to think about database transactions. A database transaction has to be as short as possible, to reduce lock contention in the database. Long database transactions will prevent your application from scaling to highly concurrent load. Hence, it is almost never good design to hold a database transaction open during user think time, until the unit of work is complete.

What is the scope of a unit of work? Can a single Hibernate `Session` span several database transactions or is this a one-to-one relationship of scopes? When should you open and close a `Session` and how do you demarcate the database transaction boundaries?

1.1. Unit of work

First, don't use the *session-per-operation* antipattern, that is, don't open and close a `Session` for every simple database call in a single thread! Of course, the same is true for database transactions. Database calls in an application are made using a planned sequence, they are grouped into atomic units of work. (Note that this also means that auto-commit after every single

SQL statement is useless in an application, this mode is intended for ad-hoc SQL console work. Hibernate disables, or expects the application server to do so, auto-commit mode immediately.) Database transactions are never optional, all communication with a database has to occur inside a transaction, no matter if you read or write data. As explained, auto-commit behavior for reading data should be avoided, as many small transactions are unlikely to perform better than one clearly defined unit of work. The latter is also much more maintainable and extensible.

The most common pattern in a multi-user client/server application is *session-per-request*. In this model, a request from the client is send to the server (where the Hibernate persistence layer runs), a new Hibernate `Session` is opened, and all database operations are executed in this unit of work. Once the work has been completed (and the response for the client has been prepared), the session is flushed and closed. You would also use a single database transaction to serve the clients request, starting and committing it when you open and close the `Session`. The relationship between the two is one-to-one and this model is a perfect fit for many applications.

The challenge lies in the implementation. Hibernate provides built-in management of the "current session" to simplify this pattern. All you have to do is start a transaction when a server request has to be processed, and end the transaction before the response is send to the client. You can do this in any way you like, common solutions are `ServletFilter`, AOP interceptor with a pointcut on the service methods, or a proxy/interception container. An EJB container is a standardized way to implement cross-cutting aspects such as transaction demarcation on EJB session beans, declaratively with CMT. If you decide to use programmatic transaction demarcation, prefer the Hibernate `Transaction` API shown later in this chapter, for ease of use and code portability.

Your application code can access a "current session" to process the request by simply calling `sessionFactory.getCurrentSession()` anywhere and as often as needed. You will always get a `Session` scoped to the current database transaction. This has to be configured for either resource-local or JTA environments, see *Section 5, “Contextual Sessions”*.

Sometimes it is convenient to extend the scope of a `Session` and database transaction until the "view has been rendered". This is especially useful in servlet applications that utilize a separate rendering phase after the request has been processed. Extending the database transaction until view rendering is complete is easy to do if you implement your own interceptor. However, it is not easily doable if you rely on EJBs with container-managed transactions, as a transaction will be completed when an EJB method returns, before rendering of any view can start. See the Hibernate website and forum for tips and examples around this *Open Session in View* pattern.

1.2. Long conversations

The session-per-request pattern is not the only useful concept you can use to design units of work. Many business processes require a whole series of interactions with the user interleaved with database accesses. In web and enterprise applications it is not acceptable for a database transaction to span a user interaction. Consider the following example:

- The first screen of a dialog opens, the data seen by the user has been loaded in a particular `Session` and database transaction. The user is free to modify the objects.

- The user clicks "Save" after 5 minutes and expects his modifications to be made persistent; he also expects that he was the only person editing this information and that no conflicting modification can occur.

We call this unit of work, from the point of view of the user, a long running *conversation* (or *application transaction*). There are many ways how you can implement this in your application.

A first naive implementation might keep the `Session` and database transaction open during user think time, with locks held in the database to prevent concurrent modification, and to guarantee isolation and atomicity. This is of course an anti-pattern, since lock contention would not allow the application to scale with the number of concurrent users.

Clearly, we have to use several database transactions to implement the converastion. In this case, maintaining isolation of business processes becomes the partial responsibility of the application tier. A single conversation usually spans several database transactions. It will be atomic if only one of these database transactions (the last one) stores the updated data, all others simply read data (e.g. in a wizard-style dialog spanning several request/response cycles). This is easier to implement than it might sound, especially if you use Hibernate's features:

- *Automatic Versioning* - Hibernate can do automatic optimistic concurrency control for you, it can automatically detect if a concurrent modification occured during user think time. Usually we only check at the end of the conversation.
- *Detached Objects* - If you decide to use the already discussed *session-per-request* pattern, all loaded instances will be in detached state during user think time. Hibernate allows you to reattach the objects and persist the modifications, the pattern is called *session-per-request-with-detached-objects*. Automatic versioning is used to isolate concurrent modifications.
- *Extended (or Long) Session* - The Hibernate `Session` may be disconnected from the underlying JDBC connection after the database transaction has been committed, and reconnected when a new client request occurs. This pattern is known as *session-per-conversation* and makes even reattachment unnecessary. Automatic versioning is used to isolate concurrent modifications and the `Session` is usually not allowed to be flushed automatically, but explicitely.

Both *session-per-request-with-detached-objects* and *session-per-conversation* have advantages and disadvantages, we discuss them later in this chapter in the context of optimistic concurrency control.

1.3. Considering object identity

An application may concurrently access the same persistent state in two different `Session` s. However, an instance of a persistent class is never shared between two `Session` instances. Hence there are two different notions of identity:

Database Identity
: `foo.getId().equals( bar.getId() )`

JVM Identity
: `foo==bar`

Then for objects attached to a *particular*`Session` (i.e. in the scope of a `Session`) the two notions are equivalent, and JVM identity for database identity is guaranteed by Hibernate. However, while the application might concurrently access the "same" (persistent identity) business object in two different sessions, the two instances will actually be "different" (JVM identity). Conflicts are resolved using (automatic versioning) at flush/commit time, using an optimistic approach.

This approach leaves Hibernate and the database to worry about concurrency; it also provides the best scalability, since guaranteeing identity in single-threaded units of work only doesn't need expensive locking or other means of synchronization. The application never needs to synchronize on any business object, as long as it sticks to a single thread per `Session`. Within a `Session` the application may safely use == to compare objects.

However, an application that uses == outside of a `Session`, might see unexpected results. This might occur even in some unexpected places, for example, if you put two detached instances into the same `Set`. Both might have the same database identity (i.e. they represent the same row), but JVM identity is by definition not guaranteed for instances in detached state. The developer has to override the `equals()` and `hashCode()` methods in persistent classes and implement his own notion of object equality. There is one caveat: Never use the database identifier to implement equality, use a business key, a combination of unique, usually immutable, attributes. The database identifier will change if a transient object is made persistent. If the transient instance (usually together with detached instances) is held in a `Set`, changing the hashcode breaks the contract of the `Set`. Attributes for business keys don't have to be as stable as database primary keys, you only have to guarantee stability as long as the objects are in the same `Set`. See the Hibernate website for a more thorough discussion of this issue. Also note that this is not a Hibernate issue, but simply how Java object identity and equality has to be implemented.

1.4. Common issues

Never use the anti-patterns *session-per-user-session* or *session-per-application* (of course, there are rare exceptions to this rule). Note that some of the following issues might also appear with the recommended patterns, make sure you understand the implications before making a design decision:

- A `Session` is not thread-safe. Things which are supposed to work concurrently, like HTTP requests, session beans, or Swing workers, will cause race conditions if a `Session` instance would be shared. If you keep your Hibernate `Session` in your `HttpSession` (discussed later), you should consider synchronizing access to your Http session. Otherwise, a user that clicks reload fast enough may use the same `Session` in two concurrently running threads.

- An exception thrown by Hibernate means you have to rollback your database transaction and close the `Session` immediately (discussed later in more detail). If your `Session` is bound to the application, you have to stop the application. Rolling back the database transaction doesn't put your business objects back into the state they were at the start of the transaction. This means the database state and the business objects do get out of sync. Usually this is not a problem, because exceptions are not recoverable and you have to start over after rollback anyway.

- The `Session` caches every object that is in persistent state (watched and checked for dirty state by Hibernate). This means it grows endlessly until you get an OutOfMemoryException, if you keep it open for a long time or simply load too much data. One solution for this is to call `clear()` and `evict()` to manage the `Session` cache, but you most likely should consider a Stored Procedure if you need mass data operations. Some solutions are shown in *Chapter 14, Batch processing*. Keeping a `Session` open for the duration of a user session also means a high probability of stale data.

2. Database transaction demarcation

Datatabase (or system) transaction boundaries are always necessary. No communication with the database can occur outside of a database transaction (this seems to confuse many developers who are used to the auto-commit mode). Always use clear transaction boundaries, even for read-only operations. Depending on your isolation level and database capabilities this might not be required but there is no downside if you always demarcate transactions explicitly. Certainly, a single database transaction is going to perform better than many small transactions, even for reading data.

A Hibernate application can run in non-managed (i.e. standalone, simple Web- or Swing applications) and managed J2EE environments. In a non-managed environment, Hibernate is usually responsible for its own database connection pool. The application developer has to manually set transaction boundaries, in other words, begin, commit, or rollback database transactions himself. A managed environment usually provides container-managed transactions (CMT), with the transaction assembly defined declaratively in deployment descriptors of EJB session beans, for example. Programmatic transaction demarcation is then no longer necessary.

However, it is often desirable to keep your persistence layer portable between non-managed resource-local environments, and systems that can rely on JTA but use BMT instead of CMT. In both cases you'd use programmatic transaction demaracation. Hibernate offers a wrapper API called `Transaction` that translates into the native transaction system of your deployment environment. This API is actually optional, but we strongly encourage its use unless you are in a CMT session bean.

Usually, ending a `Session` involves four distinct phases:

- flush the session
- commit the transaction
- close the session

- handle exceptions

Flushing the session has been discussed earlier, we'll now have a closer look at transaction demarcation and exception handling in both managed- and non-managed environments.

2.1. Non-managed environment

If a Hibernate persistence layer runs in a non-managed environment, database connections are usually handled by simple (i.e. non-DataSource) connection pools from which Hibernate obtains connections as needed. The session/transaction handling idiom looks like this:

```
// Non-managed environment idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

You don't have to `flush()` the `Session` explicitly - the call to `commit()` automatically triggers the synchronization (depending upon the *Section 10, “Flushing the Session”* FlushMode for the session. A call to `close()` marks the end of a session. The main implication of `close()` is that the JDBC connection will be relinquished by the session. This Java code is portable and runs in both non-managed and JTA environments.

A much more flexible solution is Hibernate's built-in "current session" context management, as described earlier:

```
// Non-managed environment idiom with getCurrentSession()
try {
    factory.getCurrentSession().beginTransaction();

    // do some work
    ...

    factory.getCurrentSession().getTransaction().commit();
}
catch (RuntimeException e) {
    factory.getCurrentSession().getTransaction().rollback();
    throw e; // or display error message
}
```

You will very likely never see these code snippets in a regular application; fatal (system) exceptions should always be caught at the "top". In other words, the code that executes Hibernate calls (in the persistence layer) and the code that handles `RuntimeException` (and usually can only clean up and exit) are in different layers. The current context management by Hibernate can significantly simplify this design, as all you need is access to a `SessionFactory`. Exception handling is discussed later in this chapter.

Note that you should select `org.hibernate.transaction.JDBCTransactionFactory` (which is the default), and for the second example `"thread"` as your `hibernate.current_session_context_class`.

2.2. Using JTA

If your persistence layer runs in an application server (e.g. behind EJB session beans), every datasource connection obtained by Hibernate will automatically be part of the global JTA transaction. You can also install a standalone JTA implementation and use it without EJB. Hibernate offers two strategies for JTA integration.

If you use bean-managed transactions (BMT) Hibernate will tell the application server to start and end a BMT transaction if you use the `Transaction` API. So, the transaction management code is identical to the non-managed environment.

```
// BMT idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

If you want to use a transaction-bound `Session`, that is, the `getCurrentSession()` functionality for easy context propagation, you will have to use the JTA `UserTransaction` API directly:

```
// BMT idiom with getCurrentSession()
try {
    UserTransaction tx = (UserTransaction)new InitialContext()
                            .lookup("java:comp/UserTransaction");
    tx.begin();

    // Do some work on Session bound to transaction
    factory.getCurrentSession().load(...);
    factory.getCurrentSession().persist(...);

    tx.commit();
}
catch (RuntimeException e) {
    tx.rollback();
    throw e; // or display error message
}
```

With CMT, transaction demarcation is done in session bean deployment descriptors, not programatically, hence, the code is reduced to:

```
// CMT idiom
 Session sess = factory.getCurrentSession();

 // do some work
 ...
```

In a CMT/EJB even rollback happens automatically, since an unhandled `RuntimeException` thrown by a session bean method tells the container to set the global transaction to rollback. *This means you do not need to use the Hibernate `Transaction` API at all with BMT or CMT, and you get automatic propagation of the "current" Session bound to the transaction.*

Note that you should choose `org.hibernate.transaction.JTATransactionFactory` if you use JTA directly (BMT), and `org.hibernate.transaction.CMTTransactionFactory` in a CMT session bean, when you configure Hibernate's transaction factory. Remember to also set `hibernate.transaction.manager_lookup_class`. Furthermore, make sure that your `hibernate.current_session_context_class` is either unset (backwards compatiblity), or set to `"jta"`.

The `getCurrentSession()` operation has one downside in a JTA environment. There is one caveat to the use of `after_statement` connection release mode, which is then used by default. Due to a silly limitation of the JTA spec, it is not possible for Hibernate to automatically clean up any unclosed `ScrollableResults` or `Iterator` instances returned by `scroll()` or `iterate()`. You *must* release the underlying database cursor by calling `ScrollableResults.close()` or `Hibernate.close(Iterator)` explicity from a `finally` block. (Of course, most applications can easily avoid using `scroll()` or `iterate()` at all from the JTA or CMT code.)

2.3. Exception handling

If the `Session` throws an exception (including any `SQLException`), you should immediately rollback the database transaction, call `Session.close()` and discard the `Session` instance. Certain methods of `Session` will *not* leave the session in a consistent state. No exception thrown by Hibernate can be treated as recoverable. Ensure that the `Session` will be closed by calling `close()` in a `finally` block.

The `HibernateException`, which wraps most of the errors that can occur in a Hibernate persistence layer, is an unchecked exception (it wasn't in older versions of Hibernate). In our opinion, we shouldn't force the application developer to catch an unrecoverable exception at a low layer. In most systems, unchecked and fatal exceptions are handled in one of the first frames of the method call stack (i.e. in higher layers) and an error message is presented to the application user (or some other appropriate action is taken). Note that Hibernate might also throw other unchecked exceptions which are not a `HibernateException`. These are, again, not recoverable and appropriate action should be taken.

Hibernate wraps `SQLException` s thrown while interacting with the database in a `JDBCException`. In fact, Hibernate will attempt to convert the eexception into a more meningful subclass of `JDBCException`. The underlying `SQLException` is always available via `JDBCException.getCause()`. Hibernate converts the `SQLException` into an appropriate `JDBCException` subclass using the `SQLExceptionConverter` attached to the `SessionFactory`. By default, the `SQLExceptionConverter` is defined by the configured dialect; however, it is also possible to plug in a custom implementation (see the javadocs for the `SQLExceptionConverterFactory` class for details). The standard `JDBCException` subtypes are:

- `JDBCConnectionException` - indicates an error with the underlying JDBC communication.
- `SQLGrammarException` - indicates a grammar or syntax problem with the issued SQL.
- `ConstraintViolationException` - indicates some form of integrity constraint violation.
- `LockAcquisitionException` - indicates an error acquiring a lock level necessary to perform the requested operation.
- `GenericJDBCException` - a generic exception which did not fall into any of the other categories.

2.4. Transaction timeout

One extremely important feature provided by a managed environment like EJB that is never provided for non-managed code is transaction timeout. Transaction timeouts ensure that no misbehaving transaction can indefinitely tie up resources while returning no response to the user. Outside a managed (JTA) environment, Hibernate cannot fully provide this functionality. However, Hibernate can at least control data access operations, ensuring that database level deadlocks and queries with huge result sets are limited by a defined timeout. In a managed environment, Hibernate can delegate transaction timeout to JTA. This functioanlity is abstracted by the Hibernate `Transaction` object.

```
Session sess = factory.openSession();
try {
    //set transaction timeout to 3 seconds
    sess.getTransaction().setTimeout(3);
    sess.getTransaction().begin();

    // do some work
    ...

    sess.getTransaction().commit()
}
catch (RuntimeException e) {
    sess.getTransaction().rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

Note that `setTimeout()` may not be called in a CMT bean, where transaction timeouts must be defined declaratively.

3. Optimistic concurrency control

The only approach that is consistent with high concurrency and high scalability is optimistic concurrency control with versioning. Version checking uses version numbers, or timestamps, to detect conflicting updates (and to prevent lost updates). Hibernate provides for three possible approaches to writing application code that uses optimistic concurrency. The use cases we show are in the context of long conversations, but version checking also has the benefit of preventing lost updates in single database transactions.

3.1. Application version checking

In an implementation without much help from Hibernate, each interaction with the database occurs in a new `Session` and the developer is responsible for reloading all persistent instances from the database before manipulating them. This approach forces the application to carry out its own version checking to ensure conversation transaction isolation. This approach is the least efficient in terms of database access. It is the approach most similar to entity EJBs.

```
// foo is an instance loaded by a previous Session
session = factory.openSession();
Transaction t = session.beginTransaction();

int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() ); // load the current state
if ( oldVersion!=foo.getVersion ) throw new StaleObjectStateException();
foo.setProperty("bar");

t.commit();
session.close();
```

The `version` property is mapped using `<version>`, and Hibernate will automatically increment it during flush if the entity is dirty.

Of course, if you are operating in a low-data-concurrency environment and don't require version checking, you may use this approach and just skip the version check. In that case, *last commit* *wins* will be the default strategy for your long conversations. Keep in mind that this might confuse the users of the application, as they might experience lost updates without error messages or a chance to merge conflicting changes.

Clearly, manual version checking is only feasible in very trivial circumstances and not practical for most applications. Often not only single instances, but complete graphs of modified ojects have to be checked. Hibernate offers automatic version checking with either an extended `Session` or detached instances as the design paradigm.

3.2. Extended session and automatic versioning

A single `Session` instance and its persistent instances are used for the whole conversation, known as *session-per-conversation*. Hibernate checks instance versions at flush time, throwing an exception if concurrent modification is detected. It's up to the developer to catch and handle this exception (common options are the opportunity for the user to merge changes or to restart the business conversation with non-stale data).

The `Session` is disconnected from any underlying JDBC connection when waiting for user interaction. This approach is the most efficient in terms of database access. The application need not concern itself with version checking or with reattaching detached instances, nor does it have to reload instances in every database transaction.

```
// foo is an instance loaded earlier by the old session
Transaction t = session.beginTransaction(); // Obtain a new JDBC connection,
start transaction

foo.setProperty("bar");

session.flush();    // Only for last transaction in conversation
t.commit();         // Also return JDBC connection
session.close();    // Only for last transaction in conversation
```

The `foo` object still knows which `Session` it was loaded in. Beginning a new database transaction on an old session obtains a new connection and resumes the session. Committing a database transaction disconnects a session from the JDBC connection and returns the connection to the pool. After reconnection, to force a version check on data you aren't updating, you may call `Session.lock()` with `LockMode.READ` on any objects that might have been updated by another transaction. You don't need to lock any data that you *are* updating. Usually you would set `FlushMode.NEVER` on an extended `Session`, so that only the last database transaction cycle is allowed to actually persist all modifications made in this conversation. Hence, only this last database transaction would include the `flush()` operation, and then also `close()` the session to end the conversation.

This pattern is problematic if the `Session` is too big to be stored during user think time, e.g. an `HttpSession` should be kept as small as possible. As the `Session` is also the (mandatory) first-level cache and contains all loaded objects, we can probably use this strategy only for a few request/response cycles. You should use a `Session` only for a single conversation, as it will soon also have stale data.

(Note that earlier Hibernate versions required explicit disconnection and reconnection of a `Session`. These methods are deprecated, as beginning and ending a transaction has the same effect.)

Also note that you should keep the disconnected `Session` close to the persistence layer. In other words, use an EJB stateful session bean to hold the `Session` in a three-tier environment, and don't transfer it to the web layer (or even serialize it to a separate tier) to store it in the `HttpSession`.

The extended session pattern, or *session-per-conversation*, is more difficult to implement with automatic current session context management. You need to supply your own implementation of the `CurrentSessionContext` for this, see the Hibernate Wiki for examples.

3.3. Detached objects and automatic versioning

Each interaction with the persistent store occurs in a new `Session`. However, the same persistent instances are reused for each interaction with the database. The application manipulates the state of detached instances originally loaded in another `Session` and then reattaches them using `Session.update()`, `Session.saveOrUpdate()`, or `Session.merge()`.

```
// foo is an instance loaded by a previous Session
foo.setProperty("bar");
session = factory.openSession();
Transaction t = session.beginTransaction();
session.saveOrUpdate(foo); // Use merge() if "foo" might have been loaded
already
t.commit();
session.close();
```

Again, Hibernate will check instance versions during flush, throwing an exception if conflicting updates occured.

You may also call `lock()` instead of `update()` and use `LockMode.READ` (performing a version check, bypassing all caches) if you are sure that the object has not been modified.

3.4. Customizing automatic versioning

You may disable Hibernate's automatic version increment for particular properties and collections by setting the `optimistic-lock` mapping attribute to `false`. Hibernate will then no longer increment versions if the property is dirty.

Legacy database schemas are often static and can't be modified. Or, other applications might also access the same database and don't know how to handle version numbers or even timestamps. In both cases, versioning can't rely on a particular column in a table. To force a version check without a version or timestamp property mapping, with a comparison of the state of all fields in a row, turn on `optimistic-lock="all"` in the `<class>` mapping. Note that this concepetually only works if Hibernate can compare the old and new state, i.e. if you use a single long `Session` and not session-per-request-with-detached-objects.

Sometimes concurrent modification can be permitted as long as the changes that have been made don't overlap. If you set `optimistic-lock="dirty"` when mapping the `<class>`, Hibernate will only compare dirty fields during flush.

In both cases, with dedicated version/timestamp columns or with full/dirty field comparison, Hibernate uses a single `UPDATE` statement (with an appropriate `WHERE` clause) per entity to execute the version check and update the information. If you use transitive persistence to cascade reattachment to associated entities, Hibernate might execute uneccessary updates. This is usually not a problem, but *on update* triggers in the database might be executed even when no changes have been made to detached instances. You can customize this behavior by setting `select-before-update="true"` in the `<class>` mapping, forcing Hibernate to `SELECT` the instance to ensure that changes did actually occur, before updating the row.

4. Pessimistic Locking

It is not intended that users spend much time worring about locking strategies. Its usually enough to specify an isolation level for the JDBC connections and then simply let the database do all the work. However, advanced users may sometimes wish to obtain exclusive pessimistic locks, or re-obtain locks at the start of a new transaction.

Hibernate will always use the locking mechanism of the database, never lock objects in memory!

The `LockMode` class defines the different lock levels that may be acquired by Hibernate. A lock is obtained by the following mechanisms:

- `LockMode.WRITE` is acquired automatically when Hibernate updates or inserts a row.
- `LockMode.UPGRADE` may be acquired upon explicit user request using `SELECT ... FOR UPDATE` on databases which support that syntax.
- `LockMode.UPGRADE_NOWAIT` may be acquired upon explicit user request using a `SELECT ... FOR UPDATE NOWAIT` under Oracle.
- `LockMode.READ` is acquired automatically when Hibernate reads data under Repeatable Read or Serializable isolation level. May be re-acquired by explicit user request.
- `LockMode.NONE` represents the absence of a lock. All objects switch to this lock mode at the end of a `Transaction`. Objects associated with the session via a call to `update()` or `saveOrUpdate()` also start out in this lock mode.

The "explicit user request" is expressed in one of the following ways:

- A call to `Session.load()`, specifying a `LockMode`.
- A call to `Session.lock()`.
- A call to `Query.setLockMode()`.

If `Session.load()` is called with `UPGRADE` or `UPGRADE_NOWAIT`, and the requested object was not yet loaded by the session, the object is loaded using `SELECT ... FOR UPDATE`. If `load()` is called for an object that is already loaded with a less restrictive lock than the one requested, Hibernate calls `lock()` for that object.

`Session.lock()` performs a version number check if the specified lock mode is `READ`, `UPGRADE` or `UPGRADE_NOWAIT`. (In the case of `UPGRADE` or `UPGRADE_NOWAIT`, `SELECT ... FOR UPDATE` is used.)

If the database does not support the requested lock mode, Hibernate will use an appropriate alternate mode (instead of throwing an exception). This ensures that applications will be portable.

5. Connection Release Modes

The legacy (2.x) behavior of Hibernate in regards to JDBC connection management was that a `Session` would obtain a connection when it was first needed and then hold unto that connection until the session was closed. Hibernate 3.x introduced the notion of connection release modes to tell a session how to handle its JDBC connections. Note that the following discussion is pertinent only to connections provided through a configured `ConnectionProvider`; user-supplied connections are outside the breadth of this discussion. The different release modes are identified by the enumerated values of `org.hibernate.ConnectionReleaseMode`:

- `ON_CLOSE` - is essentially the legacy behavior described above. The Hibernate session obatins a connection when it first needs to perform some JDBC access and holds unto that connection until the session is closed.
- `AFTER_TRANSACTION` - says to release connections after a `org.hibernate.Transaction` has completed.
- `AFTER_STATEMENT` (also referred to as aggressive release) - says to release connections after each and every statement execution. This aggressive releasing is skipped if that statement leaves open resources associated with the given session; currently the only situation where this occurs is through the use of `org.hibernate.ScrollableResults`.

The configuration parameter `hibernate.connection.release_mode` is used to specify which release mode to use. The possible values:

- `auto` (the default) - this choice delegates to the release mode returned by the `org.hibernate.transaction.TransactionFactory.getDefaultReleaseMode()` method. For JTATransactionFactory, this returns ConnectionReleaseMode.AFTER_STATEMENT; for JDBCTransactionFactory, this returns ConnectionReleaseMode.AFTER_TRANSACTION. It is rarely a good idea to change this default behavior as failures due to the value of this setting tend to indicate bugs and/or invalid assumptions in user code.
- `on_close` - says to use ConnectionReleaseMode.ON_CLOSE. This setting is left for backwards compatibility, but its use is highly discouraged.
- `after_transaction` - says to use ConnectionReleaseMode.AFTER_TRANSACTION. This setting should not be used in JTA environments. Also note that with ConnectionReleaseMode.AFTER_TRANSACTION, if a session is considered to be in auto-commit mode connections will be released as if the release mode were AFTER_STATEMENT.
- `after_statement` - says to use ConnectionReleaseMode.AFTER_STATEMENT. Additionally, the configured `ConnectionProvider` is consulted to see if it supports this setting

(`supportsAggressiveRelease()`). If not, the release mode is reset to ConnectionReleaseMode.AFTER_TRANSACTION. This setting is only safe in environments where we can either re-acquire the same underlying JDBC connection each time we make a call into `ConnectionProvider.getConnection()` or in auto-commit environments where it does not matter whether we get back the same connection.

Interceptors and events

It is often useful for the application to react to certain events that occur inside Hibernate. This allows implementation of certain kinds of generic functionality, and extension of Hibernate functionality.

1. Interceptors

The `Interceptor` interface provides callbacks from the session to the application allowing the application to inspect and/or manipulate properties of a persistent object before it is saved, updated, deleted or loaded. One possible use for this is to track auditing information. For example, the following `Interceptor` automatically sets the `createTimestamp` when an `Auditable` is created and updates the `lastUpdateTimestamp` property when an `Auditable` is updated.

You may either implement `Interceptor` directly or (better) extend `EmptyInterceptor`.

```
package org.hibernate.test;

import java.io.Serializable;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.EmptyInterceptor;
import org.hibernate.Transaction;
import org.hibernate.type.Type;

public class AuditInterceptor extends EmptyInterceptor {

    private int updates;
    private int creates;
    private int loads;

    public void onDelete(Object entity,
                         Serializable id,
                         Object[] state,
                         String[] propertyNames,
                         Type[] types) {
        // do nothing
    }

    public boolean onFlushDirty(Object entity,
                                Serializable id,
                                Object[] currentState,
                                Object[] previousState,
                                String[] propertyNames,
                                Type[] types) {

        if ( entity instanceof Auditable ) {
            updates++;
            for ( int i=0; i < propertyNames.length; i++ ) {
                if ( "lastUpdateTimestamp".equals( propertyNames[i] ) ) {
                    currentState[i] = new Date();
                    return true;
                }
            }
        }
        return false;
    }

    public boolean onLoad(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
                          Type[] types) {
        if ( entity instanceof Auditable ) {
            loads++;
        }
        return false;
    }

    public boolean onSave(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
                          Type[] types) {

        if ( entity instanceof Auditable ) {
            creates++;
            for ( int i=0; i<propertyNames.length; i++ ) {
                if ( "createTimestamp".equals( propertyNames[i] ) ) {
                    state[i] = new Date();
                    return true;
                }
            }
        }
        return false;
    }

    public void afterTransactionCompletion(Transaction tx) {
        if ( tx.wasCommitted() ) {
            System.out.println("Creations: " + creates + ", Updates: " +
updates,
                "Loads: " + loads);
        }
        updates=0;
        creates=0;
        loads=0;
    }

}
```

Interceptors come in two flavors: `Session`-scoped and `SessionFactory`-scoped.

A `Session`-scoped interceptor is specified when a session is opened using one of the overloaded SessionFactory.openSession() methods accepting an `Interceptor`.

```
Session session = sf.openSession( new AuditInterceptor() );
```

A `SessionFactory`-scoped interceptor is registered with the `Configuration` object prior to building the `SessionFactory`. In this case, the supplied interceptor will be applied to all sessions opened from that `SessionFactory`; this is true unless a session is opened explicitly specifying the interceptor to use. `SessionFactory`-scoped interceptors must be thread safe, taking care to not store session-specific state since multiple sessions will use this interceptor (potentially) concurrently.

```
new Configuration().setInterceptor( new AuditInterceptor() );
```

2. Event system

If you have to react to particular events in your persistence layer, you may also use the Hibernate3 *event* architecture. The event system can be used in addition or as a replacement for interceptors.

Essentially all of the methods of the `Session` interface correlate to an event. You have a `LoadEvent`, a `FlushEvent`, etc (consult the XML configuration-file DTD or the `org.hibernate.event` package for the full list of defined event types). When a request is made of one of these methods, the Hibernate `Session` generates an appropriate event and passes it to the configured event listeners for that type. Out-of-the-box, these listeners implement the same processing in which those methods always resulted. However, you are free to implement a customization of one of the listener interfaces (i.e., the `LoadEvent` is processed by the registered implemenation of the `LoadEventListener` interface), in which case their implementation would be responsible for processing any `load()` requests made of the `Session`.

The listeners should be considered effectively singletons; meaning, they are shared between requests, and thus should not save any state as instance variables.

A custom listener should implement the appropriate interface for the event it wants to process and/or extend one of the convenience base classes (or even the default event listeners used by Hibernate out-of-the-box as these are declared non-final for this purpose). Custom listeners can either be registered programmatically through the `Configuration` object, or specified in the Hibernate configuration XML (declarative configuration through the properties file is not supported). Here's an example of a custom load event listener:

```
public class MyLoadListener implements LoadEventListener {
    // this is the single method defined by the LoadEventListener interface
    public void onLoad(LoadEvent event, LoadEventListener.LoadType loadType)
            throws HibernateException {
        if ( !MySecurity.isAuthorized( event.getEntityClassName(),
event.getEntityId() ) ) {
            throw MySecurityException("Unauthorized access");
        }
    }
}
```

You also need a configuration entry telling Hibernate to use the listener in addition to the default listener:

```
<hibernate-configuration>
    <session-factory>
        ...
        <event type="load">
            <listener class="com.eg.MyLoadListener"/>
            <listener
class="org.hibernate.event.def.DefaultLoadEventListener"/>
        </event>
    </session-factory>
</hibernate-configuration>
```

Instead, you may register it programmatically:

```
Configuration cfg = new Configuration();
LoadEventListener[] stack = { new MyLoadListener(), new
DefaultLoadEventListener() };
cfg.EventListeners().setLoadEventListeners(stack);
```

Listeners registered declaratively cannot share instances. If the same class name is used in multiple `<listener/>` elements, each reference will result in a separate instance of that class. If you need the capability to share listener instances between listener types you must use the programmatic registration approach.

Why implement an interface and define the specific type during configuration? Well, a listener implementation could implement multiple event listener interfaces. Having the type additionally defined during registration makes it easier to turn custom listeners on or off during configuration.

3. Hibernate declarative security

Usually, declarative security in Hibernate applications is managed in a session facade layer. Now, Hibernate3 allows certain actions to be permissioned via JACC, and authorized via JAAS. This is optional functionality built on top of the event architecture.

First, you must configure the appropriate event listeners, to enable the use of JAAS authorization.

```
<listener type="pre-delete"
class="org.hibernate.secure.JACCPreDeleteEventListener"/>
<listener type="pre-update"
class="org.hibernate.secure.JACCPreUpdateEventListener"/>
<listener type="pre-insert"
class="org.hibernate.secure.JACCPreInsertEventListener"/>
<listener type="pre-load"
class="org.hibernate.secure.JACCPreLoadEventListener"/>
```

Note that `<listener type="..." class="..."/>` is just a shorthand for `<event type="..."><listener class="..."/></event>` when there is exactly one listener for a particular event type.

Next, still in `hibernate.cfg.xml`, bind the permissions to roles:

```
<grant role="admin" entity-name="User" actions="insert,update,read"/>
<grant role="su" entity-name="User" actions="*"/>
```

The role names are the roles understood by your JACC provider.

Batch processing

A naive approach to inserting 100 000 rows in the database using Hibernate might look like this:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
}
tx.commit();
session.close();
```

This would fall over with an `OutOfMemoryException` somewhere around the 50 000th row. That's because Hibernate caches all the newly inserted `Customer` instances in the session-level cache.

In this chapter we'll show you how to avoid this problem. First, however, if you are doing batch processing, it is absolutely critical that you enable the use of JDBC batching, if you intend to achieve reasonable performance. Set the JDBC batch size to a reasonable number (say, 10-50):

```
hibernate.jdbc.batch_size 20
```

Note that Hibernate disables insert batching at the JDBC level transparently if you use an `identiy` identifier generator.

You also might like to do this kind of work in a process where interaction with the second-level cache is completely disabled:

```
hibernate.cache.use_second_level_cache false
```

However, this is not absolutely necessary, since we can explicitly set the `CacheMode` to disable interaction with the second-level cache.

1. Batch inserts

When making new objects persistent, you must `flush()` and then `clear()` the session regularly, to control the size of the first-level cache.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
    if ( i % 20 == 0 ) { //20, same as the JDBC batch size
```

```
        //flush a batch of inserts and release memory:
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

2. Batch updates

For retrieving and updating data the same ideas apply. In addition, you need to use `scroll()` to take advantage of server-side cursors for queries that return many rows of data.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .setCacheMode(CacheMode.IGNORE)
    .scroll(ScrollMode.FORWARD_ONLY);
int count=0;
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    if ( ++count % 20 == 0 ) {
        //flush a batch of updates and release memory:
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

3. The StatelessSession interface

Alternatively, Hibernate provides a command-oriented API that may be used for streaming data to and from the database in the form of detached objects. A `StatelessSession` has no persistence context associated with it and does not provide many of the higher-level lifecycle semantics. In particular, a stateless session does not implement a first-level cache nor interact with any second-level or query cache. It does not implement transactional write-behind or automatic dirty checking. Operations performed using a stateless session do not ever cascade to associated instances. Collections are ignored by a stateless session. Operations performed via a stateless session bypass Hibernate's event model and interceptors. Stateless sessions are vulnerable to data aliasing effects, due to the lack of a first-level cache. A stateless session is a lower-level abstraction, much closer to the underlying JDBC.

```
StatelessSession session = sessionFactory.openStatelessSession();
Transaction tx = session.beginTransaction();
```

```
ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .scroll(ScrollMode.FORWARD_ONLY);
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    session.update(customer);
}

tx.commit();
session.close();
```

Note that in this code example, the `Customer` instances returned by the query are immediately detached. They are never associated with any persistence context.

The `insert(), update()` and `delete()` operations defined by the `StatelessSession` interface are considered to be direct database row-level operations, which result in immediate execution of a SQL `INSERT, UPDATE` or `DELETE` respectively. Thus, they have very different semantics to the `save(), saveOrUpdate()` and `delete()` operations defined by the `Session` interface.

4. DML-style operations

As already discussed, automatic and transparent object/relational mapping is concerned with the management of object state. This implies that the object state is available in memory, hence manipulating (using the SQL `Data Manipulation Language` (DML) statements: `INSERT`, `UPDATE`, `DELETE`) data directly in the database will not affect in-memory state. However, Hibernate provides methods for bulk SQL-style DML statement execution which are performed through the Hibernate Query Language (*Chapter 15, HQL: The Hibernate Query Language* HQL).

The pseudo-syntax for `UPDATE` and `DELETE` statements is: `( UPDATE | DELETE ) FROM? EntityName (WHERE where_conditions)?`. Some points to note:

- In the from-clause, the FROM keyword is optional
- There can only be a single entity named in the from-clause; it can optionally be aliased. If the entity name is aliased, then any property references must be qualified using that alias; if the entity name is not aliased, then it is illegal for any property references to be qualified.
- No *Section 4, “Forms of join syntax”* joins (either implicit or explicit) can be specified in a bulk HQL query. Sub-queries may be used in the where-clause; the subqueries, themselves, may contain joins.
- The where-clause is also optional.

As an example, to execute an HQL `UPDATE`, use the `Query.executeUpdate()` method (the method is named for those familiar with JDBC's `PreparedStatement.executeUpdate()`):

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
```

```
String hqlUpdate = "update Customer c set c.name = :newName where c.name =
:oldName";
// or String hqlUpdate = "update Customer set name = :newName where name =
:oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

HQL `UPDATE` statements, by default do not effect the *Section 1.7, “version (optional)”* version or the *Section 1.8, “timestamp (optional)”* timestamp property values for the affected entities; this is in keeping with the EJB3 specification. However, you can force Hibernate to properly reset the `version` or `timestamp` property values through the use of a `versioned update`. This is achieved by adding the `VERSIONED` keyword after the `UPDATE` keyword.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
String hqlVersionedUpdate = "update versioned Customer set name = :newName
where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

Note that custom version types (`org.hibernate.usertype.UserVersionType`) are not allowed in conjunction with a `update versioned` statement.

To execute an HQL `DELETE`, use the same `Query.executeUpdate()` method:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlDelete = "delete Customer c where c.name = :oldName";
// or String hqlDelete = "delete Customer where name = :oldName";
int deletedEntities = s.createQuery( hqlDelete )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

The `int` value returned by the `Query.executeUpdate()` method indicate the number of entities effected by the operation. Consider this may or may not correlate to the number of rows effected in the database. An HQL bulk operation might result in multiple actual SQL statements being executed, for joined-subclass, for example. The returned number indicates the number of actual entities affected by the statement. Going back to the example of joined-subclass, a delete against one of the subclasses may actually result in deletes against not just the table to which that subclass is mapped, but also the "root" table and potentially joined-subclass tables further down the inheritence hierarchy.

The pseudo-syntax for `INSERT` statements is: `INSERT INTO EntityName properties_list select_statement`. Some points to note:

- Only the INSERT INTO ... SELECT ... form is supported; not the INSERT INTO ... VALUES ... form.

  The properties_list is analogous to the `column speficiation` in the SQL `INSERT` statement. For entities involved in mapped inheritence, only properties directly defined on that given class-level can be used in the properties_list. Superclass properties are not allowed; and subclass properties do not make sense. In other words, `INSERT` statements are inherently non-polymorphic.
- select_statement can be any valid HQL select query, with the caveat that the return types must match the types expected by the insert. Currently, this is checked during query compilation rather than allowing the check to relegate to the database. Note however that this might cause problems between Hibernate `Type` s which are *equivalent* as opposed to *equal*. This might cause issues with mismatches between a property defined as a `org.hibernate.type.DateType` and a property defined as a `org.hibernate.type.TimestampType`, even though the database might not make a distinction or might be able to handle the conversion.
- For the id property, the insert statement gives you two options. You can either explicitly specify the id property in the properties_list (in which case its value is taken from the corresponding select expression) or omit it from the properties_list (in which case a generated value is used). This later option is only available when using id generators that operate in the database; attempting to use this option with any "in memory" type generators will cause an exception during parsing. Note that for the purposes of this discussion, in-database generators are considered to be `org.hibernate.id.SequenceGenerator` (and its subclasses) and any implementors of `org.hibernate.id.PostInsertIdentifierGenerator`. The most notable exception here is `org.hibernate.id.TableHiLoGenerator`, which cannot be used because it does not expose a selectable way to get its values.
- For properties mapped as either `version` or `timestamp`, the insert statement gives you two options. You can either specify the property in the properties_list (in which case its value is taken from the corresponding select expressions) or omit it from the properties_list (in which case the `seed value` defined by the `org.hibernate.type.VersionType` is used).

An example HQL `INSERT` statement execution:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlInsert = "insert into DelinquentAccount (id, name) select c.id, 
c.name from Customer
    c where ...";
int createdEntities = s.createQuery( hqlInsert )
```

```
        .executeUpdate();
tx.commit();
session.close();
```

Chapter 15.

HQL: The Hibernate Query Language

Hibernate is equipped with an extremely powerful query language that (quite intentionally) looks very much like SQL. But don't be fooled by the syntax; HQL is fully object-oriented, understanding notions like inheritence, polymorphism and association.

1. Case Sensitivity

Queries are case-insensitive, except for names of Java classes and properties. So `SeLeCT` is the same as `sELEct` is the same as `SELECT` but `org.hibernate.eg.FOO` is not `org.hibernate.eg.Foo` and `foo.barSet` is not `foo.BARSET`.

This manual uses lowercase HQL keywords. Some users find queries with uppercase keywords more readable, but we find this convention ugly when embedded in Java code.

2. The from clause

The simplest possible Hibernate query is of the form:

```
from eg.Cat
```

which simply returns all instances of the class `eg.Cat`. We don't usually need to qualify the class name, since `auto-import` is the default. So we almost always just write:

```
from Cat
```

Most of the time, you will need to assign an *alias*, since you will want to refer to the `Cat` in other parts of the query.

```
from Cat as cat
```

This query assigns the alias `cat` to `Cat` instances, so we could use that alias later in the query. The `as` keyword is optional; we could also write:

```
from Cat cat
```

Multiple classes may appear, resulting in a cartesian product or "cross" join.

```
from Formula, Parameter
```

```
from Formula as form, Parameter as param
```

It is considered good practice to name query aliases using an initial lowercase, consistent with Java naming standards for local variables (eg. `domesticCat`).

3. Associations and joins

We may also assign aliases to associated entities, or even to elements of a collection of values, using a `join`.

```
from Cat as cat
    inner join cat.mate as mate
    left outer join cat.kittens as kitten
```

```
from Cat as cat left join cat.mate.kittens as kittens
```

```
from Formula form full join form.parameter param
```

The supported join types are borrowed from ANSI SQL

- `inner join`
- `left outer join`
- `right outer join`
- `full join` (not usually useful)

The `inner join`, `left outer join` and `right outer join` constructs may be abbreviated.

```
from Cat as cat
    join cat.mate as mate
    left join cat.kittens as kitten
```

You may supply extra join conditions using the HQL `with` keyword.

```
from Cat as cat
    left join cat.kittens as kitten
        with kitten.bodyWeight > 10.0
```

In addition, a "fetch" join allows associations or collections of values to be initialized along with their parent objects, using a single select. This is particularly useful in the case of a collection. It effectively overrides the outer join and lazy declarations of the mapping file for associations and collections. See *Section 1, “Fetching strategies”* for more information.

```
from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens
```

A fetch join does not usually need to assign an alias, because the associated objects should not be used in the `where` clause (or any other clause). Also, the associated objects are not returned directly in the query results. Instead, they may be accessed via the parent object. The only reason we might need an alias is if we are recursively join fetching a further collection:

```
from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens child
    left join fetch child.kittens
```

Note that the `fetch` construct may not be used in queries called using `iterate()` (though `scroll()` can be used). Nor should `fetch` be used together with `setMaxResults()` or `setFirstResult()` as these operations are based on the result rows, which usually contain duplicates for eager collection fetching, hence, the number of rows is not what you'd expect. Nor may `fetch` be used together with an ad hoc `with` condition. It is possible to create a cartesian product by join fetching more than one collection in a query, so take care in this case. Join fetching multiple collection roles also sometimes gives unexpected results for bag mappings, so be careful about how you formulate your queries in this case. Finally, note that `full join fetch` and `right join fetch` are not meaningful.

If you are using property-level lazy fetching (with bytecode instrumentation), it is possible to force Hibernate to fetch the lazy properties immediately (in the first query) using `fetch all properties`.

```
from Document fetch all properties order by name
```

```
from Document doc fetch all properties where lower(doc.name) like '%cats%'
```

4. Forms of join syntax

HQL supports two forms of association joining: `implicit` and `explicit`.

The queries shown in the previous section all use the `explicit` form where the join keyword is explicitly used in the from clause. This is the recommended form.

The `implicit` form does not use the join keyword. Instead, the associations are "dereferenced" using dot-notation. `implicit` joins can appear in any of the HQL clauses. `implicit` join result in inner joins in the resulting SQL statement.

```
from Cat as cat where cat.mate.name like '%s%'
```

5. Refering to identifier property

There are, generally speaking, 2 ways to refer to an entity's identifier property:

- The special property (lowercase) `id` may be used to reference the identifier property of an entity *provided that entity does not define a non-identifier property named id*.
- If the entity defines a named identifier property, you may use that property name.

References to composite identifier properties follow the same naming rules. If the entity has a non-identifier property named id, the composite identifier property can only be referenced by its defined named; otherwise, the special `id` property can be used to rerference the identifier property.

Note: this has changed significantly starting in version 3.2.2. In previous versions, `id`*always* referred to the identifier property no matter what its actual name. A ramification of that decision was that non-identifier properties named `id` could never be referenced in Hibernate queries.

6. The select clause

The `select` clause picks which objects and properties to return in the query result set. Consider:

```
select mate
from Cat as cat
    inner join cat.mate as mate
```

The query will select `mate` s of other `Cat` s. Actually, you may express this query more compactly as:

```
select cat.mate from Cat cat
```

Queries may return properties of any value type including properties of component type:

```
select cat.name from DomesticCat cat
where cat.name like 'fri%'
```

```
select cust.name.firstName from Customer as cust
```

Queries may return multiple objects and/or properties as an array of type `Object[]`,

```
select mother, offspr, mate.name
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

or as a `List`,

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
```

```
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

or as an actual typesafe Java object,

```
select new Family(mother, mate, offspr)
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

assuming that the class `Family` has an appropriate constructor.

You may assign aliases to selected expressions using `as`:

```
select max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n
from Cat cat
```

This is most useful when used together with `select new map`:

```
select new map( max(bodyWeight) as max, min(bodyWeight) as min, count(*) as
n )
from Cat cat
```

This query returns a `Map` from aliases to selected values.

7. Aggregate functions

HQL queries may even return the results of aggregate functions on properties:

```
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat)
from Cat cat
```

The supported aggregate functions are

- `avg(...), sum(...), min(...), max(...)`
- `count(*)`
- `count(...), count(distinct ...), count(all...)`

You may use arithmetic operators, concatenation, and recognized SQL functions in the select clause:

```
select cat.weight + sum(kitten.weight)
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.weight
```

```
select firstName||' '||initial||' '||upper(lastName) from Person
```

The `distinct` and `all` keywords may be used and have the same semantics as in SQL.

```
select distinct cat.name from Cat cat

select count(distinct cat.name), count(cat) from Cat cat
```

8. Polymorphic queries

A query like:

```
from Cat as cat
```

returns instances not only of `Cat`, but also of subclasses like `DomesticCat`. Hibernate queries may name *any* Java class or interface in the `from` clause. The query will return instances of all persistent classes that extend that class or implement the interface. The following query would return all persistent objects:

```
from java.lang.Object o
```

The interface `Named` might be implemented by various persistent classes:

```
from Named n, Named m where n.name = m.name
```

Note that these last two queries will require more than one SQL `SELECT`. This means that the `order by` clause does not correctly order the whole result set. (It also means you can't call these queries using `Query.scroll()`.)

9. The where clause

The `where` clause allows you to narrow the list of instances returned. If no alias exists, you may refer to properties by name:

```
from Cat where name='Fritz'
```

If there is an alias, use a qualified property name:

```
from Cat as cat where cat.name='Fritz'
```

returns instances of `Cat` named 'Fritz'.

```
select foo
from Foo foo, Bar bar
where foo.startDate = bar.date
```

will return all instances of `Foo` for which there exists an instance of `bar` with a `date` property equal to the `startDate` property of the `Foo`. Compound path expressions make the `where` clause extremely powerful. Consider:

```
from Cat cat where cat.mate.name is not null
```

This query translates to an SQL query with a table (inner) join. If you were to write something like

```
from Foo foo
where foo.bar.baz.customer.address.city is not null
```

you would end up with a query that would require four table joins in SQL.

The = operator may be used to compare not only properties, but also instances:

```
from Cat cat, Cat rival where cat.mate = rival.mate
```

```
select cat, mate
from Cat cat, Cat mate
where cat.mate = mate
```

The special property (lowercase) `id` may be used to reference the unique identifier of an object. See *Section 5, “Refering to identifier property”* for more information.

```
from Cat as cat where cat.id = 123

from Cat as cat where cat.mate.id = 69
```

The second query is efficient. No table join is required!

Properties of composite identifiers may also be used. Suppose `Person` has a composite identifier consisting of `country` and `medicareNumber`. Again, see *Section 5, “Refering to identifier property”* for more information regarding referencing identifier properties.

```
from bank.Person person
where person.id.country = 'AU'
    and person.id.medicareNumber = 123456
```

```
from bank.Account account
where account.owner.id.country = 'AU'
    and account.owner.id.medicareNumber = 123456
```

Once again, the second query requires no table join.

Likewise, the special property `class` accesses the discriminator value of an instance in the case of polymorphic persistence. A Java class name embedded in the where clause will be translated to its discriminator value.

```
from Cat cat where cat.class = DomesticCat
```

You may also use components or composite user types, or properties of said component types. See *Section 17, “Components”* for more details.

An "any" type has the special properties `id` and `class`, allowing us to express a join in the following way (where `AuditLog.item` is a property mapped with `<any>`).

```
from AuditLog log, Payment payment
where log.item.class = 'Payment' and log.item.id = payment.id
```

Notice that `log.item.class` and `payment.class` would refer to the values of completely different database columns in the above query.

10. Expressions

Expressions allowed in the `where` clause include most of the kind of things you could write in SQL:

- mathematical operators `+, -, *, /`
- binary comparison operators `=, >=, <=, <>, !=, like`
- logical operations `and, or, not`
- Parentheses `( )`, indicating grouping
- `in`, `not in`, `between`, `is null`, `is not null`, `is empty`, `is not empty`, `member of` and `not member of`
- "Simple" case, `case ... when ... then ... else ... end`, and "searched" case, `case when ... then ... else ... end`
- string concatenation `...||...` or `concat(...,...)`
- `current_date()`, `current_time()`, `current_timestamp()`
- `second(...)`, `minute(...)`, `hour(...)`, `day(...)`, `month(...)`, `year(...)`,
- Any function or operator defined by EJB-QL 3.0: `substring(), trim(), lower(), upper(), length(), locate(), abs(), sqrt(), bit_length(), mod()`
- `coalesce()` and `nullif()`
- `str()` for converting numeric or temporal values to a readable string
- `cast(... as ...)`, where the second argument is the name of a Hibernate type, and

  `extract(... from ...)` if ANSI `cast()` and `extract()` is supported by the underlying database
- the HQL `index()` function, that applies to aliases of a joined indexed collection
- HQL functions that take collection-valued path expressions: `size(), minelement(), maxelement(), minindex(), maxindex()`, along with the special `elements()` and `indices` functions which may be quantified using `some, all, exists, any, in`.
- Any database-supported SQL scalar function like `sign()`, `trunc()`, `rtrim()`, `sin()`
- JDBC-style positional parameters `?`
- named parameters `:name`, `:start_date`, `:x1`
- SQL literals `'foo'`, `69`, `6.66E+2`, `'1970-01-01 10:00:01.0'`
- Java `public static final` constants `eg.Color.TABBY`

`in` and `between` may be used as follows:

```
from DomesticCat cat where cat.name between 'A' and 'B'
```

```
from DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )
```

and the negated forms may be written

```
from DomesticCat cat where cat.name not between 'A' and 'B'
```

```
from DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' )
```

Likewise, `is null` and `is not null` may be used to test for null values.

Booleans may be easily used in expressions by declaring HQL query substitutions in Hibernate configuration:

```
<property name="hibernate.query.substitutions">true 1, false 0</property>
```

This will replace the keywords `true` and `false` with the literals `1` and `0` in the translated SQL from this HQL:

```
from Cat cat where cat.alive = true
```

You may test the size of a collection with the special property `size`, or the special `size()` function.

```
from Cat cat where cat.kittens.size > 0
```

```
from Cat cat where size(cat.kittens) > 0
```

For indexed collections, you may refer to the minimum and maximum indices using `minindex` and `maxindex` functions. Similarly, you may refer to the minimum and maximum elements of a collection of basic type using the `minelement` and `maxelement` functions.

```
from Calendar cal where maxelement(cal.holidays) > current_date
```

```
from Order order where maxindex(order.items) > 100
```

```
from Order order where minelement(order.items) > 10000
```

The SQL functions `any, some, all, exists, in` are supported when passed the element or index set of a collection (`elements` and `indices` functions) or the result of a subquery (see below).

```
select mother from Cat as mother, Cat as kit
where kit in elements(foo.kittens)
```

```
select p from NameList list, Person p
where p.name = some elements(list.names)
```

```
from Cat cat where exists elements(cat.kittens)
```

```
from Player p where 3 > all elements(p.scores)
```

```
from Show show where 'fizard' in indices(show.acts)
```

Note that these constructs - `size`, `elements`, `indices`, `minindex`, `maxindex`, `minelement`, `maxelement` - may only be used in the where clause in Hibernate3.

Elements of indexed collections (arrays, lists, maps) may be referred to by index (in a where clause only):

```
from Order order where order.items[0].id = 1234
```

```
select person from Person person, Calendar calendar
where calendar.holidays['national day'] = person.birthDay
    and person.nationality.calendar = calendar
```

```
select item from Item item, Order order
where order.items[ order.deliveredItemIndices[0] ] = item and order.id = 11
```

```
select item from Item item, Order order
where order.items[ maxindex(order.items) ] = item and order.id = 11
```

The expression inside `[]` may even be an arithmetic expression.

```
select item from Item item, Order order
where order.items[ size(order.items) - 1 ] = item
```

HQL also provides the built-in `index()` function, for elements of a one-to-many association or collection of values.

```
select item, index(item) from Order order
    join order.items item
where index(item) < 5
```

Scalar SQL functions supported by the underlying database may be used

```
from DomesticCat cat where upper(cat.name) like 'FRI%'
```

If you are not yet convinced by all this, think how much longer and less readable the following query would be in SQL:

```
select cust
from Product prod,
    Store store
    inner join store.customers cust
where prod.name = 'widget'
    and store.location.name in ( 'Melbourne', 'Sydney' )
    and prod = all elements(cust.currentOrder.lineItems)
```

Hint: something like

```
SELECT cust.name, cust.address, cust.phone, cust.id, cust.current_order
FROM customers cust,
    stores store,
    locations loc,
    store_customers sc,
    product prod
WHERE prod.name = 'widget'
    AND store.loc_id = loc.id
    AND loc.name IN ( 'Melbourne', 'Sydney' )
    AND sc.store_id = store.id
    AND sc.cust_id = cust.id
    AND prod.id = ALL(
        SELECT item.prod_id
        FROM line_items item, orders o
        WHERE item.order_id = o.id
            AND cust.current_order = o.id
    )
```

11. The order by clause

The list returned by a query may be ordered by any property of a returned class or components:

```
from DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

The optional `asc` or `desc` indicate ascending or descending order respectively.

12. The group by clause

A query that returns aggregate values may be grouped by any property of a returned class or components:

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
```

```
select foo.id, avg(name), max(name)
from Foo foo join foo.names name
group by foo.id
```

A `having` clause is also allowed.

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

SQL functions and aggregate functions are allowed in the `having` and `order by` clauses, if supported by the underlying database (eg. not in MySQL).

```
select cat
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.name, cat.other, cat.properties
having avg(kitten.weight) > 100
order by count(kitten) asc, sum(kitten.weight) desc
```

Note that neither the `group by` clause nor the `order by` clause may contain arithmetic expressions. Also note that Hibernate currently does not expand a grouped entity, so you can't write `group by cat` if all properties of `cat` are non-aggregated. You have to list all non-aggregated properties explicitly.

13. Subqueries

For databases that support subselects, Hibernate supports subqueries within queries. A subquery must be surrounded by parentheses (often by an SQL aggregate function call). Even correlated subqueries (subqueries that refer to an alias in the outer query) are allowed.

```
from Cat as fatcat
where fatcat.weight > (
    select avg(cat.weight) from DomesticCat cat
)
```

```
from DomesticCat as cat
where cat.name = some (
    select name.nickName from Name as name
)
```

```
from Cat as cat
where not exists (
    from Cat as mate where mate.mate = cat
)
```

```
from DomesticCat as cat
where cat.name not in (
    select name.nickName from Name as name
)
```

```
select cat.id, (select max(kit.weight) from cat.kitten kit)
from Cat as cat
```

Note that HQL subqueries may occur only in the select or where clauses.

Note that subqueries can also utilize `row value constructor` syntax. See *Section 18, “Row value constructor syntax”* for more details.

14. HQL examples

Hibernate queries can be quite powerful and complex. In fact, the power of the query language is one of Hibernate's main selling points. Here are some example queries very similar to queries that I used on a recent project. Note that most queries you will write are much simpler than these!

The following query returns the order id, number of items and total value of the order for all unpaid orders for a particular customer and given minimum total value, ordering the results by total value. In determining the prices, it uses the current catalog. The resulting SQL query, against the `ORDER`, `ORDER_LINE`, `PRODUCT`, `CATALOG` and `PRICE` tables has four inner joins and an (uncorrelated) subselect.

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog.effectiveDate < sysdate
    and catalog.effectiveDate >= all (
        select cat.effectiveDate
        from Catalog as cat
        where cat.effectiveDate < sysdate
    )
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

What a monster! Actually, in real life, I'm not very keen on subqueries, so my query was really more like this:

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog = :currentCatalog
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

The next query counts the number of payments in each status, excluding all payments in the `AWAITING_APPROVAL` status where the most recent status change was made by the current user. It translates to an SQL query with two inner joins and a correlated subselect against the `PAYMENT`, `PAYMENT_STATUS` and `PAYMENT_STATUS_CHANGE` tables.

```
select count(payment), status.name
from Payment as payment
```

```
    join payment.currentStatus as status
    join payment.statusChanges as statusChange
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or (
        statusChange.timeStamp = (
            select max(change.timeStamp)
            from PaymentStatusChange change
            where change.payment = payment
        )
        and statusChange.user <> :currentUser
    )
group by status.name, status.sortOrder
order by status.sortOrder
```

If I would have mapped the `statusChanges` collection as a list, instead of a set, the query would have been much simpler to write.

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or payment.statusChanges[ maxIndex(payment.statusChanges) ].user <>
:currentUser
group by status.name, status.sortOrder
order by status.sortOrder
```

The next query uses the MS SQL Server `isNull()` function to return all the accounts and unpaid payments for the organization to which the current user belongs. It translates to an SQL query with three inner joins, an outer join and a subselect against the `ACCOUNT`, `PAYMENT`, `PAYMENT_STATUS`, `ACCOUNT_TYPE`, `ORGANIZATION` and `ORG_USER` tables.

```
select account, payment
from Account as account
    left outer join account.payments as payment
where :currentUser in elements(account.holder.users)
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name,
PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

For some databases, we would need to do away with the (correlated) subselect.

```
select account, payment
from Account as account
    join account.holder.users as user
    left outer join account.payments as payment
where :currentUser = user
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name,
PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

15. Bulk update and delete

HQL now supports `update`, `delete` and `insert ... select ...` statements. See *Section 4, “DML-style operations”* for details.

16. Tips & Tricks

You can count the number of query results without actually returning them:

```
( (Integer) session.iterate("select count(*) from ....").next() ).intValue()
```

To order a result by the size of a collection, use the following query:

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
order by count(msg)
```

If your database supports subselects, you can place a condition upon selection size in the where clause of your query:

```
from User usr where size(usr.messages) >= 1
```

If your database doesn't support subselects, use the following query:

```
select usr.id, usr.name
from User usr.name
    join usr.messages msg
group by usr.id, usr.name
having count(msg) >= 1
```

As this solution can't return a `User` with zero messages because of the inner join, the following form is also useful:

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
having count(msg) = 0
```

Properties of a JavaBean can be bound to named query parameters:

```
Query q = s.createQuery("from foo Foo as foo where foo.name=:name and
foo.size=:size");
q.setProperties(fooBean); // fooBean has getName() and getSize()
```

```
List foos = q.list();
```

Collections are pageable by using the `Query` interface with a filter:

```
Query q = s.createFilter( collection, "" ); // the trivial filter
q.setMaxResults(PAGE_SIZE);
q.setFirstResult(PAGE_SIZE * pageNumber);
List page = q.list();
```

Collection elements may be ordered or grouped using a query filter:

```
Collection orderedCollection = s.filter( collection, "order by this.amount"
);
Collection counts = s.filter( collection, "select this.type, count(this)
group by this.type" );
```

You can find the size of a collection without initializing it:

```
( (Integer) session.iterate("select count(*) from ....").next()
).intValue();
```

17. Components

Components might be used in just about every way that simple value types can be used in HQL queries. They can appear in the `select` clause:

```
select p.name from from Person p
```

```
select p.name.first from from Person p
```

where the Person's name property is a component. Components can also be used in the `where` clause:

```
from from Person p where p.name = :name
```

```
from from Person p where p.name.first = :firstName
```

Components can also be used in the `order by` clause:

```
from from Person p order by p.name
from from Person p order by p.name.first
```

Another common use of components is in *Section 18, “Row value constructor syntax”* row value constructors.

18. Row value constructor syntax

HQL supports the use of ANSI SQL `row value constructor` syntax (sometimes called `tuple` syntax), even though the underlying database may not support that notion. Here we are generally referring to multi-valued comparisons, typically associated with components. Consider an entity Person which defines a name component:

```
from Person p where p.name.first='John' and
p.name.last='Jingleheimer-Schmidt'
```

That's valid syntax, although a little verbose. It be nice to make this a bit more concise and use `row value constructor` syntax:

```
from Person p where p.name=('John', 'Jingleheimer-Schmidt')
```

It can also be useful to specify this in the `select` clause:

```
select p.name from from Person p
```

Another time using `row value constructor` syntax can be beneficial is when using subqueries needing to compare against multiple values:

```
from Cat as cat
where not ( cat.name, cat.color ) in (
    select cat.name, cat.color from DomesticCat cat
)
```

One thing to consider when deciding if you want to use this syntax is that the query will be dependent upon the ordering of the component sub-properties in the metadata.

Criteria Queries

Hibernate features an intuitive, extensible criteria query API.

1. Creating a Criteria instance

The interface `org.hibernate.Criteria` represents a query against a particular persistent class. The `Session` is a factory for `Criteria` instances.

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

2. Narrowing the result set

An individual query criterion is an instance of the interface `org.hibernate.criterion.Criterion`. The class `org.hibernate.criterion.Restrictions` defines factory methods for obtaining certain built-in `Criterion` types.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.between("weight", minWeight, maxWeight) )
    .list();
```

Restrictions may be grouped logically.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.or(
        Restrictions.eq( "age", new Integer(0) ),
        Restrictions.isNull("age")
    ) )
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.in( "name", new String[] { "Fritz", "Izi", "Pk" } ) )
    .add( Restrictions.disjunction()
        .add( Restrictions.isNull("age") )
        .add( Restrictions.eq("age", new Integer(0) ) )
        .add( Restrictions.eq("age", new Integer(1) ) )
        .add( Restrictions.eq("age", new Integer(2) ) )
    ) )
    .list();
```

There are quite a range of built-in criterion types (`Restrictions` subclasses), but one that is especially useful lets you specify SQL directly.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.sqlRestriction("lower({alias}.name) like lower(?)",
"Fritz%",
        Hibernate.STRING) )
    .list();
```

The `{alias}` placeholder with be replaced by the row alias of the queried entity.

An alternative approach to obtaining a criterion is to get it from a `Property` instance. You can create a `Property` by calling `Property.forName()`.

```

Property age = Property.forName("age");
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.disjunction()
        .add( age.isNull() )
        .add( age.eq( new Integer(0) ) )
        .add( age.eq( new Integer(1) ) )
        .add( age.eq( new Integer(2) ) )
    ) )
    .add( Property.forName("name").in( new String[] { "Fritz", "Izi", "Pk" }
) )
    .list();
```

3. Ordering the results

You may order the results using `org.hibernate.criterion.Order`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%")
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Property.forName("name").like("F%") )
    .addOrder( Property.forName("name").asc() )
    .addOrder( Property.forName("age").desc() )
    .setMaxResults(50)
    .list();
```

4. Associations

You may easily specify constraints upon related entities by navigating associations using `createCriteria()`.

```
List cats = sess.createCriteria(Cat.class)
```

```
    .add( Restrictions.like("name", "F%") )
    .createCriteria("kittens")
        .add( Restrictions.like("name", "F%") )
    .list();
```

note that the second `createCriteria()` returns a new instance of `Criteria`, which refers to the elements of the `kittens` collection.

The following, alternate form is useful in certain circumstances.

```
List cats = sess.createCriteria(Cat.class)
    .createAlias("kittens", "kt")
    .createAlias("mate", "mt")
    .add( Restrictions.eqProperty("kt.name", "mt.name") )
    .list();
```

(`createAlias()` does not create a new instance of `Criteria`.)

Note that the kittens collections held by the `Cat` instances returned by the previous two queries are *not* pre-filtered by the criteria! If you wish to retrieve just the kittens that match the criteria, you must use a `ResultTransformer`.

```
List cats = sess.createCriteria(Cat.class)
    .createCriteria("kittens", "kt")
        .add( Restrictions.eq("name", "F%") )
    .setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP)
    .list();
Iterator iter = cats.iterator();
while ( iter.hasNext() ) {
    Map map = (Map) iter.next();
    Cat cat = (Cat) map.get(Criteria.ROOT_ALIAS);
    Cat kitten = (Cat) map.get("kt");
}
```

5. Dynamic association fetching

You may specify association fetching semantics at runtime using `setFetchMode()`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .setFetchMode("mate", FetchMode.EAGER)
    .setFetchMode("kittens", FetchMode.EAGER)
    .list();
```

This query will fetch both `mate` and `kittens` by outer join. See *Section 1, “Fetching strategies”* for more information.

6. Example queries

The class `org.hibernate.criterion.Example` allows you to construct a query criterion from a given instance.

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .list();
```

Version properties, identifiers and associations are ignored. By default, null valued properties are excluded.

You can adjust how the `Example` is applied.

```
Example example = Example.create(cat)
    .excludeZeroes()           //exclude zero valued properties
    .excludeProperty("color")  //exclude the property named "color"
    .ignoreCase()              //perform case insensitive string comparisons
    .enableLike();             //use like for string comparisons
List results = session.createCriteria(Cat.class)
    .add(example)
    .list();
```

You can even use examples to place criteria upon associated objects.

```
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .createCriteria("mate")
        .add( Example.create( cat.getMate() ) )
    .list();
```

7. Projections, aggregation and grouping

The class `org.hibernate.criterion.Projections` is a factory for `Projection` instances. We apply a projection to a query by calling `setProjection()`.

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.rowCount() )
    .add( Restrictions.eq("color", Color.BLACK) )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount() )
        .add( Projections.avg("weight") )
        .add( Projections.max("weight") )
        .add( Projections.groupProperty("color") )
    )
    .list();
```

There is no explicit "group by" necessary in a criteria query. Certain projection types are defined to be *grouping projections*, which also appear in the SQL `group by` clause.

An alias may optionally be assigned to a projection, so that the projected value may be referred to in restrictions or orderings. Here are two different ways to do this:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.alias( Projections.groupProperty("color"),
"colr" ) )
    .addOrder( Order.asc("colr") )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.groupProperty("color").as("colr") )
    .addOrder( Order.asc("colr") )
    .list();
```

The `alias()` and `as()` methods simply wrap a projection instance in another, aliased, instance of `Projection`. As a shortcut, you can assign an alias when you add the projection to a projection list:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount(), "catCountByColor" )
        .add( Projections.avg("weight"), "avgWeight" )
        .add( Projections.max("weight"), "maxWeight" )
        .add( Projections.groupProperty("color"), "color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

```
List results = session.createCriteria(Domestic.class, "cat")
    .createAlias("kittens", "kit")
    .setProjection( Projections.projectionList()
        .add( Projections.property("cat.name"), "catName" )
        .add( Projections.property("kit.name"), "kitName" )
    )
    .addOrder( Order.asc("catName") )
    .addOrder( Order.asc("kitName") )
    .list();
```

You can also use `Property.forName()` to express projections:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Property.forName("name") )
    .add( Property.forName("color").eq(Color.BLACK) )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount().as("catCountByColor") )
        .add( Property.forName("weight").avg().as("avgWeight") )
        .add( Property.forName("weight").max().as("maxWeight") )
        .add( Property.forName("color").group().as("color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

8. Detached queries and subqueries

The `DetachedCriteria` class lets you create a query outside the scope of a session, and then later execute it using some arbitrary `Session`.

```
DetachedCriteria query = DetachedCriteria.forClass(Cat.class)
    .add( Property.forName("sex").eq('F') );

Session session = ....;
Transaction txn = session.beginTransaction();
List results =
query.getExecutableCriteria(session).setMaxResults(100).list();
txn.commit();
session.close();
```

A `DetachedCriteria` may also be used to express a subquery. Criterion instances involving subqueries may be obtained via `Subqueries` or `Property`.

```
DetachedCriteria avgWeight = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight").avg() );
session.createCriteria(Cat.class)
    .add( Property.forName("weight).gt(avgWeight) )
    .list();
```

```
DetachedCriteria weights = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight") );
session.createCriteria(Cat.class)
    .add( Subqueries.geAll("weight", weights) )
    .list();
```

Even correlated subqueries are possible:

```
DetachedCriteria avgWeightForSex = DetachedCriteria.forClass(Cat.class,
"cat2")
    .setProjection( Property.forName("weight").avg() )
    .add( Property.forName("cat2.sex").eqProperty("cat.sex") );
session.createCriteria(Cat.class, "cat")
    .add( Property.forName("weight).gt(avgWeightForSex) )
    .list();
```

9. Queries by natural identifier

For most queries, including criteria queries, the query cache is not very efficient, because query cache invalidation occurs too frequently. However, there is one special kind of query where we can optimize the cache invalidation algorithm: lookups by a constant natural key. In some applications, this kind of query occurs frequently. The criteria API provides special provision for this use case.

First, you should map the natural key of your entity using `<natural-id>`, and enable use of the second-level cache.

```
<class name="User">
    <cache usage="read-write"/>
    <id name="id">
        <generator class="increment"/>
    </id>
    <natural-id>
        <property name="name"/>
        <property name="org"/>
    </natural-id>
    <property name="password"/>
</class>
```

Note that this functionality is not intended for use with entities with *mutable* natural keys.

Next, enable the Hibernate query cache.

Now, `Restrictions.naturalId()` allows us to make use of the more efficient cache algorithm.

```
session.createCriteria(User.class)
    .add( Restrictions.naturalId()
        .set("name", "gavin")
        .set("org", "hb")
    ).setCacheable(true)
    .uniqueResult();
```

Native SQL

You may also express queries in the native SQL dialect of your database. This is useful if you want to utilize database specific features such as query hints or the `CONNECT` keyword in Oracle. It also provides a clean migration path from a direct SQL/JDBC based application to Hibernate.

Hibernate3 allows you to specify handwritten SQL (including stored procedures) for all create, update, delete, and load operations.

1. Using a SQLQuery

Execution of native SQL queries is controlled via the `SQLQuery` interface, which is obtained by calling `Session.createSQLQuery()`. The following describes how to use this API for querying.

1.1. Scalar queries

The most basic SQL query is to get a list of scalars (values).

```
sess.createSQLQuery("SELECT * FROM CATS").list();
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").list();
```

These will both return a List of Object arrays (Object[]) with scalar values for each column in the CATS table. Hibernate will use ResultSetMetadata to deduce the actual order and types of the returned scalar values.

To avoid the overhead of using `ResultSetMetadata` or simply to be more explicit in what is returned one can use `addScalar()`.

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME", Hibernate.STRING)
 .addScalar("BIRTHDATE", Hibernate.DATE)
```

This query specified:

- the SQL query string
- the columns and types to return

This will still return Object arrays, but now it will not use `ResultSetMetdata` but will instead explicitly get the ID, NAME and BIRTHDATE column as respectively a Long, String and a Short from the underlying resultset. This also means that only these three columns will be returned, even though the query is using `*` and could return more than the three listed columns.

It is possible to leave out the type information for all or some of the scalars.

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME")
 .addScalar("BIRTHDATE")
```

This is essentially the same query as before, but now `ResultSetMetaData` is used to decide the type of NAME and BIRTHDATE where as the type of ID is explicitly specified.

How the java.sql.Types returned from ResultSetMetaData is mapped to Hibernate types is controlled by the Dialect. If a specific type is not mapped or does not result in the expected type it is possible to customize it via calls to `registerHibernateType` in the Dialect.

1.2. Entity queries

The above queries were all about returning scalar values, basically returning the "raw" values from the resultset. The following shows how to get entity objects from a native sql query via `addEntity()`.

```
sess.createSQLQuery("SELECT * FROM CATS").addEntity(Cat.class);
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM
CATS").addEntity(Cat.class);
```

This query specified:

- the SQL query string
- the entity returned by the query

Assuming that Cat is mapped as a class with the columns ID, NAME and BIRTHDATE the above queries will both return a List where each element is a Cat entity.

If the entity is mapped with a `many-to-one` to another entity it is required to also return this when performing the native query, otherwise a database specific "column not found" error will occur. The additional columns will automatically be returned when using the * notation, but we prefer to be explicit as in the following example for a `many-to-one` to a `Dog`:

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, DOG_ID FROM
CATS").addEntity(Cat.class);
```

This will allow cat.getDog() to function properly.

1.3. Handling associations and collections

It is possible to eagerly join in the `Dog` to avoid the possible extra roundtrip for initializing the proxy. This is done via the `addJoin()` method, which allows you to join in an association or collection.

```
sess.createSQLQuery("SELECT c.ID, NAME, BIRTHDATE, DOG_ID, D_ID, D_NAME FROM
CATS c, DOGS d
    WHERE c.DOG_ID = d.D_ID")
 .addEntity("cat", Cat.class)
 .addJoin("cat.dog");
```

In this example the returned `Cat`'s will have their `dog` property fully initialized without any extra roundtrip to the database. Notice that we added a alias name ("cat") to be able to specify the target property path of the join. It is possible to do the same eager joining for collections, e.g. if the `Cat` had a one-to-many to `Dog` instead.

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, D_ID, D_NAME, CAT_ID FROM
CATS c, DOGS d
    WHERE c.ID = d.CAT_ID")
 .addEntity("cat", Cat.class)
 .addJoin("cat.dogs");
```

At this stage we are reaching the limits of what is possible with native queries without starting to enhance the sql queries to make them usable in Hibernate; the problems starts to arise when returning multiple entities of the same type or when the default alias/column names are not enough.

1.4. Returning multiple entities

Until now the result set column names are assumed to be the same as the column names specified in the mapping document. This can be problematic for SQL queries which join multiple tables, since the same column names may appear in more than one table.

Column alias injection is needed in the following query (which most likely will fail):

```
sess.createSQLQuery("SELECT c.*, m.*  FROM CATS c, CATS m WHERE c.MOTHER_ID
= c.ID")
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

The intention for this query is to return two Cat instances per row, a cat and its mother. This will fail since there is a conflict of names since they are mapped to the same column names and on some databases the returned column aliases will most likely be on the form "c.ID", "c.NAME", etc. which are not equal to the columns specificed in the mappings ("ID" and "NAME").

The following form is not vulnerable to column name duplication:

```
sess.createSQLQuery("SELECT {cat.*}, {mother.*}  FROM CATS c, CATS m WHERE
c.MOTHER_ID = c.ID")
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

This query specified:

- the SQL query string, with placeholders for Hibernate to inject column aliases
- the entities returned by the query

The {cat.*} and {mother.*} notation used above is a shorthand for "all properties". Alternatively, you may list the columns explicity, but even in this case we let Hibernate inject the SQL column aliases for each property. The placeholder for a column alias is just the property name qualified by the table alias. In the following example, we retrieve Cats and their mothers from a different table (cat_log) to the one declared in the mapping metadata. Notice that we may even use the property aliases in the where clause if we like.

```
String sql = "SELECT ID as {c.id}, NAME as {c.name}, " +
         "BIRTHDATE as {c.birthDate}, MOTHER_ID as {c.mother}, {mother.*} "
+
         "FROM CAT_LOG c, CAT_LOG m WHERE {c.mother} = c.ID";

List loggedCats = sess.createSQLQuery(sql)
        .addEntity("cat", Cat.class)
        .addEntity("mother", Cat.class).list()
```

1.4.1. Alias and property references

For most cases the above alias injection is needed, but for queries relating to more complex mappings like composite properties, inheritance discriminators, collections etc. there are some specific aliases to use to allow Hibernate to inject the proper aliases.

The following table shows the different possibilities of using the alias injection. Note: the alias names in the result are examples, each alias will have a unique and probably different name when used.

Description	Syntax	Example
A simple property	`{[aliasname].[propertyname]}`	`A_NAME as {item.name}`
A composite property	`{[aliasname].[componentname].[propertyname]}`	`CURRENCY as {item.amount.currency}, VALUE as {item.amount.value}`
Discriminator of an entity	`{[aliasname].class}`	`DISC as {item.class}`
All properties of an entity	`{[aliasname].*}`	`{item.*}`
A collection key	`{[aliasname].key}`	`ORGID as {coll.key}`
The id of an collection	`{[aliasname].id}`	`EMPID as {coll.id}`
The element of an collection	`{[aliasname].element}`	`XID as {coll.element}`

Description	Syntax	Example
roperty of the element in the collection	`{[aliasname].element.[propertyname]}`	`NAME as {coll.element.name}`
All properties of the element in the collection	`{[aliasname].element.*}`	`{coll.element.*}`
All properties of the the collection	`{[aliasname].*}`	`{coll.*}`

Table 17.1. Alias injection names

1.5. Returning non-managed entities

It is possible to apply a ResultTransformer to native sql queries. Allowing it to e.g. return non-managed entities.

```
sess.createSQLQuery("SELECT NAME, BIRTHDATE FROM CATS")
        .setResultTransformer(Transformers.aliasToBean(CatDTO.class))
```

This query specified:

- the SQL query string
- a result transformer

The above query will return a list of `CatDTO` which has been instantiated and injected the values of NAME and BIRTHNAME into its corresponding properties or fields.

1.6. Handling inheritance

Native sql queries which query for entities that is mapped as part of an inheritance must include all properties for the baseclass and all it subclasses.

1.7. Parameters

Native sql queries support positional as well as named parameters:

```
Query query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME like
?").addEntity(Cat.class);
List pusList = query.setString(0, "Pus%").list();

query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME like
:name").addEntity(Cat.class);
List pusList = query.setString("name", "Pus%").list();
```

2. Named SQL queries

Named SQL queries may be defined in the mapping document and called in exactly the same way as a named HQL query. In this case, we do *not* need to call `addEntity()`.

```
<sql-query name="persons">
    <return alias="person" class="eg.Person"/>
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex}
    FROM PERSON person
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

```
List people = sess.getNamedQuery("persons")
    .setString("namePattern", namePattern)
    .setMaxResults(50)
    .list();
```

The `<return-join>` and `<load-collection>` elements are used to join associations and define queries which initialize collections, respectively.

```
<sql-query name="personsWith">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress"/>
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           adddress.STREET AS {address.street},
           adddress.CITY AS {address.city},
           adddress.STATE AS {address.state},
           adddress.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS adddress
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

A named SQL query may return a scalar value. You must declare the column alias and Hibernate type using the `<return-scalar>` element:

```
<sql-query name="mySqlQuery">
    <return-scalar column="name" type="string"/>
    <return-scalar column="age" type="long"/>
    SELECT p.NAME AS name,
           p.AGE AS age,
    FROM PERSON p WHERE p.NAME LIKE 'Hiber%'
```

```
</sql-query>
```

You can externalize the resultset mapping informations in a `<resultset>` element to either reuse them accross several named queries or through the `setResultSetMapping()` API.

```
<resultset name="personAddress">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress"/>
</resultset>

<sql-query name="personsWith" resultset-ref="personAddress">
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           adddress.STREET AS {address.street},
           adddress.CITY AS {address.city},
           adddress.STATE AS {address.state},
           adddress.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS adddress
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

You can alternatively use the resultset mapping information in your hbm files directly in java code.

```
List cats = sess.createSQLQuery(
        "select {cat.*}, {kitten.*} from cats cat, cats kitten where
kitten.mother = cat.id"
    )
    .setResultSetMapping("catAndKitten")
    .list();
```

2.1. Using return-property to explicitly specify column/alias names

With `<return-property>` you can explicitly tell Hibernate what column aliases to use, instead of using the `{}`-syntax to let Hibernate inject its own aliases.

```
<sql-query name="mySqlQuery">
    <return alias="person" class="eg.Person">
        <return-property name="name" column="myName"/>
        <return-property name="age" column="myAge"/>
        <return-property name="sex" column="mySex"/>
    </return>
    SELECT person.NAME AS myName,
           person.AGE AS myAge,
           person.SEX AS mySex,
    FROM PERSON person WHERE person.NAME LIKE :name
```

```
</sql-query>
```

`<return-property>` also works with multiple columns. This solves a limitation with the `{}`-syntax which can not allow fine grained control of multi-column properties.

```
<sql-query name="organizationCurrentEmployments">
    <return alias="emp" class="Employment">
        <return-property name="salary">
            <return-column name="VALUE"/>
            <return-column name="CURRENCY"/>
        </return-property>
        <return-property name="endDate" column="myEndDate"/>
    </return>
        SELECT EMPLOYEE AS {emp.employee}, EMPLOYER AS {emp.employer},
        STARTDATE AS {emp.startDate}, ENDDATE AS {emp.endDate},
        REGIONCODE as {emp.regionCode}, EID AS {emp.id}, VALUE, CURRENCY
        FROM EMPLOYMENT
        WHERE EMPLOYER = :id AND ENDDATE IS NULL
        ORDER BY STARTDATE ASC
</sql-query>
```

Notice that in this example we used `<return-property>` in combination with the `{}`-syntax for injection. Allowing users to choose how they want to refer column and properties.

If your mapping has a discriminator you must use `<return-discriminator>` to specify the discriminator column.

2.2. Using stored procedures for querying

Hibernate 3 introduces support for queries via stored procedures and functions. Most of the following documentation is equivalent for both. The stored procedure/function must return a resultset as the first out-parameter to be able to work with Hibernate. An example of such a stored function in Oracle 9 and higher is as follows:

```
CREATE OR REPLACE FUNCTION selectAllEmployments
    RETURN SYS_REFCURSOR
AS
    st_cursor SYS_REFCURSOR;
BEGIN
    OPEN st_cursor FOR
 SELECT EMPLOYEE, EMPLOYER,
 STARTDATE, ENDDATE,
 REGIONCODE, EID, VALUE, CURRENCY
 FROM EMPLOYMENT;
      RETURN  st_cursor;
 END;
```

To use this query in Hibernate you need to map it via a named query.

```
<sql-query name="selectAllEmployees_SP" callable="true">
    <return alias="emp" class="Employment">
        <return-property name="employee" column="EMPLOYEE"/>
        <return-property name="employer" column="EMPLOYER"/>
        <return-property name="startDate" column="STARTDATE"/>
        <return-property name="endDate" column="ENDDATE"/>
        <return-property name="regionCode" column="REGIONCODE"/>
        <return-property name="id" column="EID"/>
        <return-property name="salary">
            <return-column name="VALUE"/>
            <return-column name="CURRENCY"/>
        </return-property>
    </return>
    { ? = call selectAllEmployments() }
</sql-query>
```

Notice stored procedures currently only return scalars and entities. `<return-join>` and `<load-collection>` are not supported.

2.2.1. Rules/limitations for using stored procedures

To use stored procedures with Hibernate the procedures/functions have to follow some rules. If they do not follow those rules they are not usable with Hibernate. If you still want to use these procedures you have to execute them via `session.connection()`. The rules are different for each database, since database vendors have different stored procedure semantics/syntax.

Stored procedure queries can't be paged with `setFirstResult()/setMaxResults()`.

Recommended call form is standard SQL92: `{ ? = call functionName(<parameters>) }` or `{ ? = call procedureName(<parameters>}`. Native call syntax is not supported.

For Oracle the following rules apply:

- A function must return a result set. The first parameter of a procedure must be an `OUT` that returns a result set. This is done by using a `SYS_REFCURSOR` type in Oracle 9 or 10. In Oracle you need to define a `REF CURSOR` type, see Oracle literature.

For Sybase or MS SQL server the following rules apply:

- The procedure must return a result set. Note that since these servers can/will return multiple result sets and update counts, Hibernate will iterate the results and take the first result that is a result set as its return value. Everything else will be discarded.
- If you can enable `SET NOCOUNT ON` in your procedure it will probably be more efficient, but this is not a requirement.

3. Custom SQL for create, update and delete

Hibernate3 can use custom SQL statements for create, update, and delete operations. The class and collection persisters in Hibernate already contain a set of configuration time generated strings (insertsql, deletesql, updatesql etc.). The mapping tags `<sql-insert>`, `<sql-delete>`, and `<sql-update>` override these strings:

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <sql-insert>INSERT INTO PERSON (NAME, ID) VALUES ( UPPER(?), ?
)</sql-insert>
    <sql-update>UPDATE PERSON SET NAME=UPPER(?) WHERE ID=?</sql-update>
    <sql-delete>DELETE FROM PERSON WHERE ID=?</sql-delete>
</class>
```

The SQL is directly executed in your database, so you are free to use any dialect you like. This will of course reduce the portability of your mapping if you use database specific SQL.

Stored procedures are supported if the `callable` attribute is set:

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <sql-insert callable="true">{call createPerson (?, ?)}</sql-insert>
    <sql-delete callable="true">{? = call deletePerson (?)}</sql-delete>
    <sql-update callable="true">{? = call updatePerson (?, ?)}</sql-update>
</class>
```

The order of the positional parameters are currently vital, as they must be in the same sequence as Hibernate expects them.

You can see the expected order by enabling debug logging for the `org.hibernate.persister.entity` level. With this level enabled Hibernate will print out the static SQL that is used to create, update, delete etc. entities. (To see the expected sequence, remember to not include your custom SQL in the mapping files as that will override the Hibernate generated static sql.)

The stored procedures are in most cases (read: better do it than not) required to return the number of rows inserted/updated/deleted, as Hibernate has some runtime checks for the success of the statement. Hibernate always registers the first statement parameter as a numeric output parameter for the CUD operations:

```
CREATE OR REPLACE FUNCTION updatePerson (uid IN NUMBER, uname IN VARCHAR2)
    RETURN NUMBER IS
BEGIN

    update PERSON
    set
```

```
        NAME = uname,
    where
        ID = uid;

    return SQL%ROWCOUNT;

END updatePerson;
```

4. Custom SQL for loading

You may also declare your own SQL (or HQL) queries for entity loading:

```
<sql-query name="person">
    <return alias="pers" class="Person" lock-mode="upgrade"/>
    SELECT NAME AS {pers.name}, ID AS {pers.id}
    FROM PERSON
    WHERE ID=?
    FOR UPDATE
</sql-query>
```

This is just a named query declaration, as discussed earlier. You may reference this named query in a class mapping:

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <loader query-ref="person"/>
</class>
```

This even works with stored procedures.

You may even define a query for collection loading:

```
<set name="employments" inverse="true">
    <key/>
    <one-to-many class="Employment"/>
    <loader query-ref="employments"/>
</set>
```

```
<sql-query name="employments">
    <load-collection alias="emp" role="Person.employments"/>
    SELECT {emp.*}
    FROM EMPLOYMENT emp
    WHERE EMPLOYER = :id
    ORDER BY STARTDATE ASC, EMPLOYEE ASC
</sql-query>
```

You could even define an entity loader that loads a collection by join fetching:

```
<sql-query name="person">
    <return alias="pers" class="Person"/>
    <return-join alias="emp" property="pers.employments"/>
    SELECT NAME AS {pers.*}, {emp.*}
    FROM PERSON pers
    LEFT OUTER JOIN EMPLOYMENT emp
        ON pers.ID = emp.PERSON_ID
    WHERE ID=?
</sql-query>
```

Filtering data

Hibernate3 provides an innovative new approach to handling data with "visibility" rules. A *Hibernate filter* is a global, named, parameterized filter that may be enabled or disabled for a particular Hibernate session.

1. Hibernate filters

Hibernate3 adds the ability to pre-define filter criteria and attach those filters at both a class and a collection level. A filter criteria is the ability to define a restriction clause very similiar to the existing "where" attribute available on the class and various collection elements. Except these filter conditions can be parameterized. The application can then make the decision at runtime whether given filters should be enabled and what their parameter values should be. Filters can be used like database views, but parameterized inside the application.

In order to use filters, they must first be defined and then attached to the appropriate mapping elements. To define a filter, use the `<filter-def/>` element within a `<hibernate-mapping/>` element:

```
<filter-def name="myFilter">
    <filter-param name="myFilterParam" type="string"/>
</filter-def>
```

Then, this filter can be attached to a class:

```
<class name="myClass" ...>
    ...
    <filter name="myFilter" condition=":myFilterParam =
MY_FILTERED_COLUMN"/>
</class>
```

or, to a collection:

```
<set ...>
    <filter name="myFilter" condition=":myFilterParam =
MY_FILTERED_COLUMN"/>
</set>
```

or, even to both (or multiples of each) at the same time.

The methods on `Session` are: `enableFilter(String filterName)`, `getEnabledFilter(String filterName)`, and `disableFilter(String filterName)`. By default, filters are *not* enabled for a given session; they must be explcitly enabled through use of the `Session.enabledFilter()` method, which returns an instance of the `Filter` interface. Using the simple filter defined above, this would look like:

```
session.enableFilter("myFilter").setParameter("myFilterParam",
"some-value");
```

Note that methods on the org.hibernate.Filter interface do allow the method-chaining common to much of Hibernate.

A full example, using temporal data with an effective record date pattern:

```
<filter-def name="effectiveDate">
    <filter-param name="asOfDate" type="date"/>
</filter-def>

<class name="Employee" ...>
...
    <many-to-one name="department" column="dept_id" class="Department"/>
    <property name="effectiveStartDate" type="date" column="eff_start_dt"/>
    <property name="effectiveEndDate" type="date" column="eff_end_dt"/>
...
    <!--
        Note that this assumes non-terminal records have an eff_end_dt set
to
        a max db date for simplicity-sake
    -->
    <filter name="effectiveDate"
            condition=":asOfDate BETWEEN eff_start_dt and eff_end_dt"/>
</class>

<class name="Department" ...>
...
    <set name="employees" lazy="true">
        <key column="dept_id"/>
        <one-to-many class="Employee"/>
        <filter name="effectiveDate"
                condition=":asOfDate BETWEEN eff_start_dt and eff_end_dt"/>
    </set>
</class>
```

Then, in order to ensure that you always get back currently effective records, simply enable the filter on the session prior to retrieving employee data:

```
Session session = ...;
session.enabledFilter("effectiveDate").setParameter("asOfDate", new Date());
List results = session.createQuery("from Employee as e where e.salary >
:targetSalary")
         .setLong("targetSalary", new Long(1000000))
         .list();
```

In the HQL above, even though we only explicitly mentioned a salary constraint on the results, because of the enabled filter the query will return only currently active employees who have a salary greater than a million dollars.

Note: if you plan on using filters with outer joining (either through HQL or load fetching) be careful of the direction of the condition expression. Its safest to set this up for left outer joining; in general, place the parameter first followed by the column name(s) after the operator.

After being defined a filter might be attached to multiple entities and/or collections each with its own condition. That can be tedious when the conditions are the same each time. Thus `<filter-def/>` allows defining a default condition, either as an attribute or CDATA:

```
<filter-def name="myFilter" condition="abc > xyz">...</filter-def>
<filter-def name="myOtherFilter">abc=xyz</filter-def>
```

This default condition will then be used whenever the filter is attached to something without specifying a condition. Note that this means you can give a specific condition as part of the attachment of the filter which overrides the default condition in that particular case.

XML Mapping

Note that this is an experimental feature in Hibernate 3.0 and is under extremely active development.

1. Working with XML data

Hibernate lets you work with persistent XML data in much the same way you work with persistent POJOs. A parsed XML tree can be thought of as just another way to represent the relational data at the object level, instead of POJOs.

Hibernate supports dom4j as API for manipulating XML trees. You can write queries that retrieve dom4j trees from the database and have any modification you make to the tree automatically synchronized to the database. You can even take an XML document, parse it using dom4j, and write it to the database with any of Hibernate's basic operations: `persist(), saveOrUpdate(), merge(), delete(), replicate()` (merging is not yet supported).

This feature has many applications including data import/export, externalization of entity data via JMS or SOAP and XSLT-based reporting.

A single mapping may be used to simultaneously map properties of a class and nodes of an XML document to the database, or, if there is no class to map, it may be used to map just the XML.

1.1. Specifying XML and class mapping together

Here is an example of mapping a POJO and XML simultaneously:

```
<class name="Account"
        table="ACCOUNTS"
        node="account">

    <id name="accountId"
            column="ACCOUNT_ID"
            node="@id"/>

    <many-to-one name="customer"
            column="CUSTOMER_ID"
            node="customer/@id"
            embed-xml="false"/>

    <property name="balance"
            column="BALANCE"
            node="balance"/>

    ...

</class>
```

1.2. Specifying only an XML mapping

Here is an example where there is no POJO class:

```
<class entity-name="Account"
        table="ACCOUNTS"
        node="account">

    <id name="id"
            column="ACCOUNT_ID"
            node="@id"
            type="string"/>

    <many-to-one name="customerId"
            column="CUSTOMER_ID"
            node="customer/@id"
            embed-xml="false"
            entity-name="Customer"/>

    <property name="balance"
            column="BALANCE"
            node="balance"
            type="big_decimal"/>

    ...

</class>
```

This mapping allows you to access the data as a dom4j tree, or as a graph of property name/value pairs (java `Map` s). The property names are purely logical constructs that may be referred to in HQL queries.

2. XML mapping metadata

Many Hibernate mapping elements accept the `node` attribute. This let's you specify the name of an XML attribute or element that holds the property or entity data. The format of the `node` attribute must be one of the following:

- `"element-name"` - map to the named XML element
- `"@attribute-name"` - map to the named XML attribute
- `"."` - map to the parent element
- `"element-name/@attribute-name"` - map to the named attribute of the named element

For collections and single valued associations, there is an additional `embed-xml` attribute. If `embed-xml="true"`, the default, the XML tree for the associated entity (or collection of value type) will be embedded directly in the XML tree for the entity that owns the association. Otherwise, if `embed-xml="false"`, then only the referenced identifier value will appear in the XML for single point associations and collections will simply not appear at all.

You should be careful not to leave `embed-xml="true"` for too many associations, since XML does not deal well with circularity!

```
<class name="Customer"
        table="CUSTOMER"
        node="customer">

    <id name="id"
            column="CUST_ID"
            node="@id"/>

    <map name="accounts"
            node="."
            embed-xml="true">
        <key column="CUSTOMER_ID"
                not-null="true"/>
        <map-key column="SHORT_DESC"
                node="@short-desc"
                type="string"/>
        <one-to-many entity-name="Account"
                embed-xml="false"
                node="account"/>
    </map>

    <component name="name"
            node="name">
        <property name="firstName"
                node="first-name"/>
        <property name="initial"
                node="initial"/>
        <property name="lastName"
                node="last-name"/>
    </component>

    ...

</class>
```

in this case, we have decided to embed the collection of account ids, but not the actual account data. The following HQL query:

```
from Customer c left join fetch c.accounts where c.lastName like :lastName
```

Would return datasets such as this:

```
<customer id="123456789">
    <account short-desc="Savings">987632567</account>
    <account short-desc="Credit Card">985612323</account>
    <name>
        <first-name>Gavin</first-name>
        <initial>A</initial>
        <last-name>King</last-name>
    </name>
    ...
</customer>
```

If you set `embed-xml="true"` on the `<one-to-many>` mapping, the data might look more like this:

```
<customer id="123456789">
    <account id="987632567" short-desc="Savings">
        <customer id="123456789"/>
        <balance>100.29</balance>
    </account>
    <account id="985612323" short-desc="Credit Card">
        <customer id="123456789"/>
        <balance>-2370.34</balance>
    </account>
    <name>
        <first-name>Gavin</first-name>
        <initial>A</initial>
        <last-name>King</last-name>
    </name>
    ...
</customer>
```

3. Manipulating XML data

Let's rearead and update XML documents in the application. We do this by obtaining a dom4j session:

```
Document doc = ....;

Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();

List results = dom4jSession
    .createQuery("from Customer c left join fetch c.accounts where
c.lastName like :lastName")
    .list();
for ( int i=0; i<results.size(); i++ ) {
    //add the customer data to the XML document
    Element customer = (Element) results.get(i);
    doc.add(customer);
}

tx.commit();
session.close();
```

```
Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();

Element cust = (Element) dom4jSession.get("Customer", customerId);
```

```
for ( int i=0; i<results.size(); i++ ) {
    Element customer = (Element) results.get(i);
    //change the customer name in the XML and database
    Element name = customer.element("name");
    name.element("first-name").setText(firstName);
    name.element("initial").setText(initial);
    name.element("last-name").setText(lastName);
}

tx.commit();
session.close();
```

It is extremely useful to combine this feature with Hibernate's `replicate()` operation to implement XML-based data import/export.

Improving performance

1. Fetching strategies

A *fetching strategy* is the strategy Hibernate will use for retrieving associated objects if the application needs to navigate the association. Fetch strategies may be declared in the O/R mapping metadata, or over-ridden by a particular HQL or `Criteria` query.

Hibernate3 defines the following fetching strategies:

- *Join fetching* - Hibernate retrieves the associated instance or collection in the same `SELECT`, using an `OUTER JOIN`.
- *Select fetching* - a second `SELECT` is used to retrieve the associated entity or collection. Unless you explicitly disable lazy fetching by specifying `lazy="false"`, this second select will only be executed when you actually access the association.
- *Subselect fetching* - a second `SELECT` is used to retrieve the associated collections for all entities retrieved in a previous query or fetch. Unless you explicitly disable lazy fetching by specifying `lazy="false"`, this second select will only be executed when you actually access the association.
- *Batch fetching* - an optimization strategy for select fetching - Hibernate retrieves a batch of entity instances or collections in a single `SELECT`, by specifying a list of primary keys or foreign keys.

Hibernate also distinguishes between:

- *Immediate fetching* - an association, collection or attribute is fetched immediately, when the owner is loaded.
- *Lazy collection fetching* - a collection is fetched when the application invokes an operation upon that collection. (This is the default for collections.)
- *"Extra-lazy" collection fetching* - individual elements of the collection are accessed from the database as needed. Hibernate tries not to fetch the whole collection into memory unless absolutely needed (suitable for very large collections)
- *Proxy fetching* - a single-valued association is fetched when a method other than the identifier getter is invoked upon the associated object.
- *"No-proxy" fetching* - a single-valued association is fetched when the instance variable is accessed. Compared to proxy fetching, this approach is less lazy (the association is fetched even when only the identifier is accessed) but more transparent, since no proxy is visible to the application. This approach requires buildtime bytecode instrumentation and is rarely necessary.

- *Lazy attribute fetching* - an attribute or single valued association is fetched when the instance variable is accessed. This approach requires buildtime bytecode instrumentation and is rarely necessary.

We have two orthogonal notions here: *when* is the association fetched, and *how* is it fetched (what SQL is used). Don't confuse them! We use `fetch` to tune performance. We may use `lazy` to define a contract for what data is always available in any detached instance of a particular class.

1.1. Working with lazy associations

By default, Hibernate3 uses lazy select fetching for collections and lazy proxy fetching for single-valued associations. These defaults make sense for almost all associations in almost all applications.

Note: if you set `hibernate.default_batch_fetch_size`, Hibernate will use the batch fetch optimization for lazy fetching (this optimization may also be enabled at a more granular level).

However, lazy fetching poses one problem that you must be aware of. Access to a lazy association outside of the context of an open Hibernate session will result in an exception. For example:

```
s = sessions.openSession();
Transaction tx = s.beginTransaction();

User u = (User) s.createQuery("from User u where u.name=:userName")
    .setString("userName", userName).uniqueResult();
Map permissions = u.getPermissions();

tx.commit();
s.close();

Integer accessLevel = (Integer) permissions.get("accounts");  // Error!
```

Since the permissions collection was not initialized when the `Session` was closed, the collection will not be able to load its state. *Hibernate does not support lazy initialization for detached objects*. The fix is to move the code that reads from the collection to just before the transaction is committed.

Alternatively, we could use a non-lazy collection or association, by specifying `lazy="false"` for the association mapping. However, it is intended that lazy initialization be used for almost all collections and associations. If you define too many non-lazy associations in your object model, Hibernate will end up needing to fetch the entire database into memory in every transaction!

On the other hand, we often want to choose join fetching (which is non-lazy by nature) instead of select fetching in a particular transaction. We'll now see how to customize the fetching strategy. In Hibernate3, the mechanisms for choosing a fetch strategy are identical for single-valued associations and collections.

1.2. Tuning fetch strategies

Select fetching (the default) is extremely vulnerable to N+1 selects problems, so we might want to enable join fetching in the mapping document:

```
<set name="permissions"
            fetch="join">
    <key column="userId"/>
    <one-to-many class="Permission"/>
</set
```

```
<many-to-one name="mother" class="Cat" fetch="join"/>
```

The `fetch` strategy defined in the mapping document affects:

- retrieval via `get()` or `load()`
- retrieval that happens implicitly when an association is navigated
- `Criteria` queries
- HQL queries if `subselect` fetching is used

No matter what fetching strategy you use, the defined non-lazy graph is guaranteed to be loaded into memory. Note that this might result in several immediate selects being used to execute a particular HQL query.

Usually, we don't use the mapping document to customize fetching. Instead, we keep the default behavior, and override it for a particular transaction, using `left join fetch` in HQL. This tells Hibernate to fetch the association eagerly in the first select, using an outer join. In the `Criteria` query API, you would use `setFetchMode(FetchMode.JOIN)`.

If you ever feel like you wish you could change the fetching strategy used by `get()` or `load()`, simply use a `Criteria` query, for example:

```
User user = (User) session.createCriteria(User.class)
                .setFetchMode("permissions", FetchMode.JOIN)
                .add( Restrictions.idEq(userId) )
                .uniqueResult();
```

(This is Hibernate's equivalent of what some ORM solutions call a "fetch plan".)

A completely different way to avoid problems with N+1 selects is to use the second-level cache.

1.3. Single-ended association proxies

Lazy fetching for collections is implemented using Hibernate's own implementation of persistent collections. However, a different mechanism is needed for lazy behavior in single-ended associations. The target entity of the association must be proxied. Hibernate implements lazy initializing proxies for persistent objects using runtime bytecode enhancement (via the excellent CGLIB library).

By default, Hibernate3 generates proxies (at startup) for all persistent classes and uses them to enable lazy fetching of `many-to-one` and `one-to-one` associations.

The mapping file may declare an interface to use as the proxy interface for that class, with the `proxy` attribute. By default, Hibernate uses a subclass of the class. *Note that the proxied class must implement a default constructor with at least package visibility. We recommend this constructor for all persistent classes!*

There are some gotchas to be aware of when extending this approach to polymorphic classes, eg.

```
<class name="Cat" proxy="Cat">
    ......
    <subclass name="DomesticCat">
        .....
    </subclass>
</class>
```

Firstly, instances of `Cat` will never be castable to `DomesticCat`, even if the underlying instance is an instance of `DomesticCat`:

```
Cat cat = (Cat) session.load(Cat.class, id);  // instantiate a proxy (does
not hit the db)
if ( cat.isDomesticCat() ) {                  // hit the db to initialize
the proxy
    DomesticCat dc = (DomesticCat) cat;       // Error!
    ....
}
```

Secondly, it is possible to break proxy ==.

```
Cat cat = (Cat) session.load(Cat.class, id);            // instantiate a Cat
proxy
DomesticCat dc =
        (DomesticCat) session.load(DomesticCat.class, id);  // acquire new
DomesticCat proxy!
System.out.println(cat==dc);                            // false
```

However, the situation is not quite as bad as it looks. Even though we now have two references to different proxy objects, the underlying instance will still be the same object:

```
cat.setWeight(11.0);  // hit the db to initialize the proxy
System.out.println( dc.getWeight() );  // 11.0
```

Third, you may not use a CGLIB proxy for a `final` class or a class with any `final` methods.

Finally, if your persistent object acquires any resources upon instantiation (eg. in initializers or default constructor), then those resources will also be acquired by the proxy. The proxy class is an actual subclass of the persistent class.

These problems are all due to fundamental limitations in Java's single inheritance model. If you wish to avoid these problems your persistent classes must each implement an interface that declares its business methods. You should specify these interfaces in the mapping file. eg.

```
<class name="CatImpl" proxy="Cat">
    ......
    <subclass name="DomesticCatImpl" proxy="DomesticCat">
        .....
    </subclass>
</class>
```

where `CatImpl` implements the interface `Cat` and `DomesticCatImpl` implements the interface `DomesticCat`. Then proxies for instances of `Cat` and `DomesticCat` may be returned by `load()` or `iterate()`. (Note that `list()` does not usually return proxies.)

```
Cat cat = (Cat) session.load(CatImpl.class, catid);
Iterator iter = session.iterate("from CatImpl as cat where
cat.name='fritz'");
Cat fritz = (Cat) iter.next();
```

Relationships are also lazily initialized. This means you must declare any properties to be of type `Cat`, not `CatImpl`.

Certain operations do *not* require proxy initialization

- `equals()`, if the persistent class does not override `equals()`
- `hashCode()`, if the persistent class does not override `hashCode()`
- The identifier getter method

Hibernate will detect persistent classes that override `equals()` or `hashCode()`.

By choosing `lazy="no-proxy"` instead of the default `lazy="proxy"`, we can avoid the problems associated with typecasting. However, we will require buildtime bytecode instrumentation, and all operations will result in immediate proxy initialization.

1.4. Initializing collections and proxies

A `LazyInitializationException` will be thrown by Hibernate if an uninitialized collection or proxy is accessed outside of the scope of the `Session`, ie. when the entity owning the collection or having the reference to the proxy is in the detached state.

Sometimes we need to ensure that a proxy or collection is initialized before closing the `Session`.

Of course, we can alway force initialization by calling `cat.getSex()` or `cat.getKittens().size()`, for example. But that is confusing to readers of the code and is not convenient for generic code.

The static methods `Hibernate.initialize()` and `Hibernate.isInitialized()` provide the application with a convenient way of working with lazily initialized collections or proxies. `Hibernate.initialize(cat)` will force the initialization of a proxy, `cat`, as long as its `Session` is still open. `Hibernate.initialize( cat.getKittens() )` has a similar effect for the collection of kittens.

Another option is to keep the `Session` open until all needed collections and proxies have been loaded. In some application architectures, particularly where the code that accesses data using Hibernate, and the code that uses it are in different application layers or different physical processes, it can be a problem to ensure that the `Session` is open when a collection is initialized. There are two basic ways to deal with this issue:

- In a web-based application, a servlet filter can be used to close the `Session` only at the very end of a user request, once the rendering of the view is complete (the *Open Session in View* pattern). Of course, this places heavy demands on the correctness of the exception handling of your application infrastructure. It is vitally important that the `Session` is closed and the transaction ended before returning to the user, even when an exception occurs during rendering of the view. See the Hibernate Wiki for examples of this "Open Session in View" pattern.
- In an application with a separate business tier, the business logic must "prepare" all collections that will be needed by the web tier before returning. This means that the business tier should load all the data and return all the data already initialized to the presentation/web tier that is required for a particular use case. Usually, the application calls `Hibernate.initialize()` for each collection that will be needed in the web tier (this call must occur before the session is closed) or retrieves the collection eagerly using a Hibernate query with a `FETCH` clause or a `FetchMode.JOIN` in `Criteria`. This is usually easier if you adopt the *Command* pattern instead of a *Session Facade*.
- You may also attach a previously loaded object to a new `Session` with `merge()` or `lock()` before accessing uninitialized collections (or other proxies). No, Hibernate does not, and certainly *should* not do this automatically, since it would introduce ad hoc transaction semantics!

Sometimes you don't want to initialize a large collection, but still need some information about it (like its size) or a subset of the data.

You can use a collection filter to get the size of a collection without initializing it:

```
( (Integer) s.createFilter( collection, "select count(*)" ).list().get(0)
).intValue()
```

The `createFilter()` method is also used to efficiently retrieve subsets of a collection without needing to initialize the whole collection:

```
s.createFilter( lazyCollection,
"").setFirstResult(0).setMaxResults(10).list();
```

1.5. Using batch fetching

Hibernate can make efficient use of batch fetching, that is, Hibernate can load several uninitialized proxies if one proxy is accessed (or collections. Batch fetching is an optimization of the lazy select fetching strategy. There are two ways you can tune batch fetching: on the class and the collection level.

Batch fetching for classes/entities is easier to understand. Imagine you have the following situation at runtime: You have 25 `Cat` instances loaded in a `Session`, each `Cat` has a reference to its `owner`, a `Person`. The `Person` class is mapped with a proxy, `lazy="true"`. If you now iterate through all cats and call `getOwner()` on each, Hibernate will by default execute 25 `SELECT` statements, to retrieve the proxied owners. You can tune this behavior by specifying a `batch-size` in the mapping of `Person`:

```
<class name="Person" batch-size="10">...</class>
```

Hibernate will now execute only three queries, the pattern is 10, 10, 5.

You may also enable batch fetching of collections. For example, if each `Person` has a lazy collection of `Cat` s, and 10 persons are currently loaded in the `Sesssion`, iterating through all persons will generate 10 `SELECT` s, one for every call to `getCats()`. If you enable batch fetching for the `cats` collection in the mapping of `Person`, Hibernate can pre-fetch collections:

```
<class name="Person">
    <set name="cats" batch-size="3">
        ...
    </set>
</class>
```

With a `batch-size` of 8, Hibernate will load 3, 3, 3, 1 collections in four `SELECT` s. Again, the value of the attribute depends on the expected number of uninitialized collections in a particular `Session`.

Batch fetching of collections is particularly useful if you have a nested tree of items, ie. the typical bill-of-materials pattern. (Although a *nested set* or a *materialized path* might be a better option for read-mostly trees.)

1.6. Using subselect fetching

If one lazy collection or single-valued proxy has to be fetched, Hibernate loads all of them, re-running the original query in a subselect. This works in the same way as batch-fetching, without the piecemeal loading.

1.7. Using lazy property fetching

Hibernate3 supports the lazy fetching of individual properties. This optimization technique is also known as *fetch groups*. Please note that this is mostly a marketing feature, as in practice, optimizing row reads is much more important than optimization of column reads. However, only loading some properties of a class might be useful in extreme cases, when legacy tables have hundreds of columns and the data model can not be improved.

To enable lazy property loading, set the `lazy` attribute on your particular property mappings:

```
<class name="Document">
       <id name="id">
        <generator class="native"/>
    </id>
    <property name="name" not-null="true" length="50"/>
    <property name="summary" not-null="true" length="200" lazy="true"/>
    <property name="text" not-null="true" length="2000" lazy="true"/>
</class>
```

Lazy property loading requires buildtime bytecode instrumentation! If your persistent classes are not enhanced, Hibernate will silently ignore lazy property settings and fall back to immediate fetching.

For bytecode instrumentation, use the following Ant task:

```
<target name="instrument" depends="compile">
    <taskdef name="instrument"
classname="org.hibernate.tool.instrument.InstrumentTask">
        <classpath path="${jar.path}"/>
        <classpath path="${classes.dir}"/>
        <classpath refid="lib.class.path"/>
    </taskdef>

    <instrument verbose="true">
        <fileset dir="${testclasses.dir}/org/hibernate/auction/model">
            <include name="*.class"/>
        </fileset>
    </instrument>
</target>
```

A different (better?) way to avoid unnecessary column reads, at least for read-only transactions is to use the projection features of HQL or Criteria queries. This avoids the need for buildtime bytecode processing and is certainly a prefered solution.

You may force the usual eager fetching of properties using `fetch all properties` in HQL.

2. The Second Level Cache

A Hibernate `Session` is a transaction-level cache of persistent data. It is possible to configure a cluster or JVM-level (`SessionFactory`-level) cache on a class-by-class and collection-by-collection basis. You may even plug in a clustered cache. Be careful. Caches are never aware of changes made to the persistent store by another application (though they may be configured to regularly expire cached data).

You have the option to tell Hibernate which caching implementation to use by specifying the name of a class that implements `org.hibernate.cache.CacheProvider` using the property `hibernate.cache.provider_class`. Hibernate comes bundled with a number of built-in integrations with open-source cache providers (listed below); additionally, you could implement your own and plug it in as outlined above. Note that versions prior to 3.2 defaulted to use EhCache as the default cache provider; that is no longer the case as of 3.2.

Cache	Provider class	Type	Cluster Safe	Query Cache Supported
Hashtable (not intended for production use)	`org.hibernate.cache.HashtableCacheProvider`	memory		yes
EHCache	`org.hibernate.cache.EhCacheProvider`	memory, disk		yes
OSCache	`org.hibernate.cache.OSCacheProvider`	memory, disk	yes	
SwarmCache	`org.hibernate.cache.SwarmCacheProvider`	clustered (ip multicast)	yes (clustered invalidation)	
JBoss TreeCache	`org.hibernate.cache.TreeCacheProvider`	clustered (ip multicast), transactional	yes (replication)	yes (clock sync req.)

Table 20.1. Cache Providers

2.1. Cache mappings

The `<cache>` element of a class or collection mapping has the following form:

```
<cache
    usage="transactional|read-write|nonstrict-read-write|read-only"
    region="RegionName"
    include="all|non-lazy"
/>
```

`usage` (required) specifies the caching strategy: `transactional`, `read-write`, `nonstrict-read-write` or `read-only`

`region` (optional, defaults to the class or collection role name) specifies the name of the second level cache region

`include` (optional, defaults to `all`) `non-lazy` specifies that properties of the entity mapped with `lazy="true"` may not be cached when attribute-level lazy fetching is enabled

Alternatively (preferrably?), you may specify `<class-cache>` and `<collection-cache>` elements in `hibernate.cfg.xml`.

The `usage` attribute specifies a *cache concurrency strategy*.

2.2. Strategy: read only

If your application needs to read but never modify instances of a persistent class, a `read-only` cache may be used. This is the simplest and best performing strategy. It's even perfectly safe for use in a cluster.

```
<class name="eg.Immutable" mutable="false">
    <cache usage="read-only"/>
    ....
</class>
```

2.3. Strategy: read/write

If the application needs to update data, a `read-write` cache might be appropriate. This cache strategy should never be used if serializable transaction isolation level is required. If the cache is used in a JTA environment, you must specify the property `hibernate.transaction.manager_lookup_class`, naming a strategy for obtaining the JTA `TransactionManager`. In other environments, you should ensure that the transaction is completed when `Session.close()` or `Session.disconnect()` is called. If you wish to use this strategy in a cluster, you should ensure that the underlying cache implementation supports locking. The built-in cache providers do *not*.

```
<class name="eg.Cat" .... >
    <cache usage="read-write"/>
    ....
    <set name="kittens" ... >
        <cache usage="read-write"/>
        ....
    </set>
</class>
```

2.4. Strategy: nonstrict read/write

If the application only occasionally needs to update data (ie. if it is extremely unlikely that two transactions would try to update the same item simultaneously) and strict transaction isolation is not required, a `nonstrict-read-write` cache might be appropriate. If the cache is used in a JTA environment, you must specify `hibernate.transaction.manager_lookup_class`. In other environments, you should ensure that the transaction is completed when `Session.close()` or `Session.disconnect()` is called.

2.5. Strategy: transactional

The `transactional` cache strategy provides support for fully transactional cache providers such as JBoss TreeCache. Such a cache may only be used in a JTA environment and you must specify `hibernate.transaction.manager_lookup_class`.

None of the cache providers support all of the cache concurrency strategies. The following table shows which providers are compatible with which concurrency strategies.

Cache	read-only	nonstrict-read-wri	read-write	transactional
Hashtable (not intended for production use)	yes	yes	yes	
EHCache	yes	yes	yes	
OSCache	yes	yes	yes	
SwarmCache	yes	yes		
JBoss TreeCache	yes	yes		

Table 20.2. Cache Concurrency Strategy Support

3. Managing the caches

Whenever you pass an object to `save()`, `update()` or `saveOrUpdate()` and whenever you retrieve an object using `load()`, `get()`, `list()`, `iterate()` or `scroll()`, that object is added to the internal cache of the `Session`.

When `flush()` is subsequently called, the state of that object will be synchronized with the database. If you do not want this synchronization to occur or if you are processing a huge number of objects and need to manage memory efficiently, the `evict()` method may be used to remove the object and its collections from the first-level cache.

```
ScrollableResult cats = sess.createQuery("from Cat as cat").scroll(); //a
huge result set
while ( cats.next() ) {
    Cat cat = (Cat) cats.get(0);
    doSomethingWithACat(cat);
    sess.evict(cat);
}
```

The `Session` also provides a `contains()` method to determine if an instance belongs to the session cache.

To completely evict all objects from the session cache, call `Session.clear()`

For the second-level cache, there are methods defined on `SessionFactory` for evicting the cached state of an instance, entire class, collection instance or entire collection role.

```
sessionFactory.evict(Cat.class, catId); //evict a particular Cat
sessionFactory.evict(Cat.class);  //evict all Cats
sessionFactory.evictCollection("Cat.kittens", catId); //evict a particular
collection of kittens
sessionFactory.evictCollection("Cat.kittens"); //evict all kitten
collections
```

The `CacheMode` controls how a particular session interacts with the second-level cache.

- `CacheMode.NORMAL` - read items from and write items to the second-level cache
- `CacheMode.GET` - read items from the second-level cache, but don't write to the second-level cache except when updating data
- `CacheMode.PUT` - write items to the second-level cache, but don't read from the second-level cache
- `CacheMode.REFRESH` - write items to the second-level cache, but don't read from the second-level cache, bypass the effect of `hibernate.cache.use_minimal_puts`, forcing a refresh of the second-level cache for all items read from the database

To browse the contents of a second-level or query cache region, use the `Statistics` API:

```
Map cacheEntries = sessionFactory.getStatistics()
        .getSecondLevelCacheStatistics(regionName)
        .getEntries();
```

You'll need to enable statistics, and, optionally, force Hibernate to keep the cache entries in a more human-understandable format:

```
hibernate.generate_statistics true
hibernate.cache.use_structured_entries true
```

4. The Query Cache

Query result sets may also be cached. This is only useful for queries that are run frequently with the same parameters. To use the query cache you must first enable it:

```
hibernate.cache.use_query_cache true
```

This setting causes the creation of two new cache regions - one holding cached query result sets (`org.hibernate.cache.StandardQueryCache`), the other holding timestamps of the most recent updates to queryable tables (`org.hibernate.cache.UpdateTimestampsCache`). Note that the query cache does not cache the state of the actual entities in the result set; it caches only identifier values and results of value type. So the query cache should always be used in conjunction with the second-level cache.

Most queries do not benefit from caching, so by default queries are not cached. To enable caching, call `Query.setCacheable(true)`. This call allows the query to look for existing cache results or add its results to the cache when it is executed.

If you require fine-grained control over query cache expiration policies, you may specify a named cache region for a particular query by calling `Query.setCacheRegion()`.

```
List blogs = sess.createQuery("from Blog blog where blog.blogger =
:blogger")
    .setEntity("blogger", blogger)
    .setMaxResults(15)
    .setCacheable(true)
    .setCacheRegion("frontpages")
    .list();
```

If the query should force a refresh of its query cache region, you should call `Query.setCacheMode(CacheMode.REFRESH)`. This is particularly useful in cases where underlying data may have been updated via a separate process (i.e., not modified through Hibernate) and allows the application to selectively refresh particular query result sets. This is a more efficient alternative to eviction of a query cache region via `SessionFactory.evictQueries()`.

5. Understanding Collection performance

We've already spent quite some time talking about collections. In this section we will highlight a couple more issues about how collections behave at runtime.

5.1. Taxonomy

Hibernate defines three basic kinds of collections:

- collections of values
- one to many associations
- many to many associations

This classification distinguishes the various table and foreign key relationships but does not tell us quite everything we need to know about the relational model. To fully understand the relational structure and performance characteristics, we must also consider the structure of the primary key that is used by Hibernate to update or delete collection rows. This suggests the following classification:

- indexed collections
- sets
- bags

All indexed collections (maps, lists, arrays) have a primary key consisting of the `<key>` and `<index>` columns. In this case collection updates are usually extremely efficient - the primary key may be efficiently indexed and a particular row may be efficiently located when Hibernate tries to update or delete it.

Sets have a primary key consisting of `<key>` and element columns. This may be less efficient for some types of collection element, particularly composite elements or large text or binary fields; the database may not be able to index a complex primary key as efficently. On the other hand, for one to many or many to many associations, particularly in the case of synthetic identifiers, it is likely to be just as efficient. (Side-note: if you want `SchemaExport` to actually create the primary key of a `<set>` for you, you must declare all columns as `not-null="true"`.)

`<idbag>` mappings define a surrogate key, so they are always very efficient to update. In fact, they are the best case.

Bags are the worst case. Since a bag permits duplicate element values and has no index column, no primary key may be defined. Hibernate has no way of distinguishing between duplicate rows. Hibernate resolves this problem by completely removing (in a single `DELETE`) and recreating the collection whenever it changes. This might be very inefficient.

Note that for a one-to-many association, the "primary key" may not be the physical primary key of the database table - but even in this case, the above classification is still useful. (It still reflects how Hibernate "locates" individual rows of the collection.)

5.2. Lists, maps, idbags and sets are the most efficient collections to update

From the discussion above, it should be clear that indexed collections and (usually) sets allow the most efficient operation in terms of adding, removing and updating elements.

There is, arguably, one more advantage that indexed collections have over sets for many to many associations or collections of values. Because of the structure of a `Set`, Hibernate doesn't ever `UPDATE` a row when an element is "changed". Changes to a `Set` always work via `INSERT` and `DELETE` (of individual rows). Once again, this consideration does not apply to one to many associations.

After observing that arrays cannot be lazy, we would conclude that lists, maps and idbags are the most performant (non-inverse) collection types, with sets not far behind. Sets are expected to be the most common kind of collection in Hibernate applications. This is because the "set" semantics are most natural in the relational model.

However, in well-designed Hibernate domain models, we usually see that most collections are in fact one-to-many associations with `inverse="true"`. For these associations, the update is handled by the many-to-one end of the association, and so considerations of collection update performance simply do not apply.

5.3. Bags and lists are the most efficient inverse collections

Just before you ditch bags forever, there is a particular case in which bags (and also lists) are much more performant than sets. For a collection with `inverse="true"` (the standard bidirectional one-to-many relationship idiom, for example) we can add elements to a bag or list without needing to initialize (fetch) the bag elements! This is because `Collection.add()` or `Collection.addAll()` must always return true for a bag or `List` (unlike a `Set`). This can make the following common code much faster.

```
Parent p = (Parent) sess.load(Parent.class, id);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c);  //no need to fetch the collection!
sess.flush();
```

5.4. One shot delete

Occasionally, deleting collection elements one by one can be extremely inefficient. Hibernate isn't completely stupid, so it knows not to do that in the case of an newly-empty collection (if you called `list.clear()`, for example). In this case, Hibernate will issue a single `DELETE` and we are done!

Suppose we add a single element to a collection of size twenty and then remove two elements. Hibernate will issue one `INSERT` statement and two `DELETE` statements (unless the collection is a bag). This is certainly desirable.

However, suppose that we remove eighteen elements, leaving two and then add thee new elements. There are two possible ways to proceed

- delete eighteen rows one by one and then insert three rows
- remove the whole collection (in one SQL `DELETE`) and insert all five current elements (one by one)

Hibernate isn't smart enough to know that the second option is probably quicker in this case. (And it would probably be undesirable for Hibernate to be that smart; such behaviour might confuse database triggers, etc.)

Fortunately, you can force this behaviour (ie. the second strategy) at any time by discarding (ie. dereferencing) the original collection and returning a newly instantiated collection with all the current elements. This can be very useful and powerful from time to time.

Of course, one-shot-delete does not apply to collections mapped `inverse="true"`.

6. Monitoring performance

Optimization is not much use without monitoring and access to performance numbers. Hibernate provides a full range of figures about its internal operations. Statistics in Hibernate are available per `SessionFactory`.

6.1. Monitoring a SessionFactory

You can access `SessionFactory` metrics in two ways. Your first option is to call `sessionFactory.getStatistics()` and read or display the `Statistics` yourself.

Hibernate can also use JMX to publish metrics if you enable the `StatisticsService` MBean. You may enable a single MBean for all your `SessionFactory` or one per factory. See the following code for minimalistic configuration examples:

```
// MBean service registration for a specific SessionFactory
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "myFinancialApp");
ObjectName on = new ObjectName("hibernate", tb); // MBean object name

StatisticsService stats = new StatisticsService(); // MBean implementation
stats.setSessionFactory(sessionFactory); // Bind the stats to a
SessionFactory
server.registerMBean(stats, on); // Register the Mbean on the server
```

```
// MBean service registration for all SessionFactory's
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "all");
ObjectName on = new ObjectName("hibernate", tb); // MBean object name

StatisticsService stats = new StatisticsService(); // MBean implementation
server.registerMBean(stats, on); // Register the MBean on the server
```

TODO: This doesn't make sense: In the first case, we retrieve and use the MBean directly. In the second one, we must give the JNDI name in which the session factory is held before using it. Use `hibernateStatsBean.setSessionFactoryJNDIName("my/JNDI/Name")`

You can (de)activate the monitoring for a `SessionFactory`

- at configuration time, set `hibernate.generate_statistics` to `false`

- at runtime: `sf.getStatistics().setStatisticsEnabled(true)` or `hibernateStatsBean.setStatisticsEnabled(true)`

Statistics can be reset programatically using the `clear()` method. A summary can be sent to a logger (info level) using the `logSummary()` method.

6.2. Metrics

Hibernate provides a number of metrics, from very basic to the specialized information only relevant in certain scenarios. All available counters are described in the `Statistics` interface API, in three categories:

- Metrics related to the general `Session` usage, such as number of open sessions, retrieved JDBC connections, etc.
- Metrics related to he entities, collections, queries, and caches as a whole (aka global metrics),
- Detailed metrics related to a particular entity, collection, query or cache region.

For exampl,e you can check the cache hit, miss, and put ratio of entities, collections and queries, and the average time a query needs. Beware that the number of milliseconds is subject to approximation in Java. Hibernate is tied to the JVM precision, on some platforms this might even only be accurate to 10 seconds.

Simple getters are used to access the global metrics (i.e. not tied to a particular entity, collection, cache region, etc.). You can access the metrics of a particular entity, collection or cache region through its name, and through its HQL or SQL representation for queries. Please refer to the `Statistics`, `EntityStatistics`, `CollectionStatistics`, `SecondLevelCacheStatistics`, and `QueryStatistics` API Javadoc for more information. The following code shows a simple example:

```
Statistics stats = HibernateUtil.sessionFactory.getStatistics();

double queryCacheHitCount  = stats.getQueryCacheHitCount();
double queryCacheMissCount = stats.getQueryCacheMissCount();
double queryCacheHitRatio =
  queryCacheHitCount / (queryCacheHitCount + queryCacheMissCount);

log.info("Query Hit ratio:" + queryCacheHitRatio);

EntityStatistics entityStats =
  stats.getEntityStatistics( Cat.class.getName() );
long changes =
        entityStats.getInsertCount()
        + entityStats.getUpdateCount()
        + entityStats.getDeleteCount();
log.info(Cat.class.getName() + " changed " + changes + "times"  );
```

To work on all entities, collections, queries and region caches, you can retrieve the list of names of entities, collections, queries and region caches with the following methods: `getQueries()`, `getEntityNames()`, `getCollectionRoleNames()`, and `getSecondLevelCacheRegionNames()`.

Toolset Guide

Roundtrip engineering with Hibernate is possible using a set of Eclipse plugins, commandline tools, as well as Ant tasks.

The *Hibernate Tools* currently include plugins for the Eclipse IDE as well as Ant tasks for reverse engineering of existing databases:

- *Mapping Editor:* An editor for Hibernate XML mapping files, supporting auto-completion and syntax highlighting. It also supports semantic auto-completion for class names and property/field names, making it much more versatile than a normal XML editor.
- *Console:* The console is a new view in Eclipse. In addition to a tree overview of your console configurations, you also get an interactive view of your persistent classes and their relationships. The console allows you to execute HQL queries against your database and browse the result directly in Eclipse.
- *Development Wizards:* Several wizards are provided with the Hibernate Eclipse tools; you can use a wizard to quickly generate Hibernate configuration (cfg.xml) files, or you may even completely reverse engineer an existing database schema into POJO source files and Hibernate mapping files. The reverse engineering wizard supports customizable templates.
- *Ant Tasks:*

Please refer to the *Hibernate Tools* package and it's documentation for more information.

However, the Hibernate main package comes bundled with an integrated tool (it can even be used from "inside" Hibernate on-the-fly): *SchemaExport* aka `hbm2ddl`.

1. Automatic schema generation

DDL may be generated from your mapping files by a Hibernate utility. The generated schema includes referential integrity constraints (primary and foreign keys) for entity and collection tables. Tables and sequences are also created for mapped identifier generators.

You *must* specify a SQL `Dialect` via the `hibernate.dialect` property when using this tool, as DDL is highly vendor specific.

First, customize your mapping files to improve the generated schema.

1.1. Customizing the schema

Many Hibernate mapping elements define optional attributes named `length`, `precision` and `scale`. You may set the length, precision and scale of a column with this attribute.

```
<property name="zip" length="5"/>
```

```
<property name="balance" precision="12" scale="2"/>
```

Some tags also accept a `not-null` attribute (for generating a `NOT NULL` constraint on table columns) and a `unique` attribute (for generating `UNIQUE` constraint on table columns).

```
<many-to-one name="bar" column="barId" not-null="true"/>
```

```
<element column="serialNumber" type="long" not-null="true" unique="true"/>
```

A `unique-key` attribute may be used to group columns in a single unique key constraint. Currently, the specified value of the `unique-key` attribute is *not* used to name the constraint in the generated DDL, only to group the columns in the mapping file.

```
<many-to-one name="org" column="orgId" unique-key="OrgEmployeeId"/>
<property name="employeeId" unique-key="OrgEmployee"/>
```

An `index` attribute specifies the name of an index that will be created using the mapped column or columns. Multiple columns may be grouped into the same index, simply by specifying the same index name.

```
<property name="lastName" index="CustName"/>
<property name="firstName" index="CustName"/>
```

A `foreign-key` attribute may be used to override the name of any generated foreign key constraint.

```
<many-to-one name="bar" column="barId" foreign-key="FKFooBar"/>
```

Many mapping elements also accept a child `<column>` element. This is particularly useful for mapping multi-column types:

```
<property name="name" type="my.customtypes.Name"/>
    <column name="last" not-null="true" index="bar_idx" length="30"/>
    <column name="first" not-null="true" index="bar_idx" length="20"/>
    <column name="initial"/>
</property>
```

The `default` attribute lets you specify a default value for a column (you should assign the same value to the mapped property before saving a new instance of the mapped class).

```
<property name="credits" type="integer" insert="false">
    <column name="credits" default="10"/>
</property>
```

```
<version name="version" type="integer" insert="false">
    <column name="version" default="0"/>
</property>
```

The `sql-type` attribute allows the user to override the default mapping of a Hibernate type to SQL datatype.

```
<property name="balance" type="float">
    <column name="balance" sql-type="decimal(13,3)"/>
</property>
```

The `check` attribute allows you to specify a check constraint.

```
<property name="foo" type="integer">
    <column name="foo" check="foo > 10"/>
</property>
```

```
<class name="Foo" table="foos" check="bar < 100.0">
    ...
    <property name="bar" type="float"/>
</class>
```

Attribute	Values	Interpretation
`length`	number	column length
`precision`	number	column decimal precision
`scale`	number	column decimal scale
`not-null`	`true|false`	specfies that the column should be non-nullable
`unique`	`true|false`	specifies that the column should have a unique constraint
`index`	`index_name`	specifies the name of a (multi-column) index
`unique-key`	`unique_key_name`	specifies the name of a multi-column unique constraint
`foreign-key`	`foreign_key_name`	specifies the name of the foreign key constraint generated for an association, for a `<one-to-one>`, `<many-to-one>`, `<key>`, or `<many-to-many>` mapping element. Note that `inverse="true"` sides will not be considered by `SchemaExport`.
`sql-type`	`SQL column type`	overrides the default column type (attribute of `<column>` element only)
`default`	SQL expression	specify a default value for the column

Attribute	Values	Interpretation
`check`	SQL expression	create an SQL check constraint on either column or table

Table 21.1. Summary

The `<comment>` element allows you to specify comments for the generated schema.

```
<class name="Customer" table="CurCust">
    <comment>Current customers only</comment>
    ...
</class>
```

```
<property name="balance">
    <column name="bal">
        <comment>Balance in USD</comment>
    </column>
</property>
```

This results in a `comment on table` or `comment on column` statement in the generated DDL (where supported).

1.2. Running the tool

The `SchemaExport` tool writes a DDL script to standard out and/or executes the DDL statements.

`java -cp` *hibernate_classpaths*`org.hibernate.tool.hbm2ddl.SchemaExport`*options mapping_files*

Option	Description
`--quiet`	don't output the script to stdout
`--drop`	only drop the tables
`--create`	only create the tables
`--text`	don't export to the database
`--output=my_schema.ddl`	output the ddl script to a file
`--naming=eg.MyNamingStrategy`	select a `NamingStrategy`
`--config=hibernate.cfg.xml`	read Hibernate configuration from an XML file
`--properties=hibernate.properties`	read database properties from a file
`--format`	format the generated SQL nicely in the script
`--delimiter=;`	set an end of line delimiter for the script

Table 21.2. `SchemaExport` Command Line Options

You may even embed `SchemaExport` in your application:

```
Configuration cfg = ....;
new SchemaExport(cfg).create(false, true);
```

1.3. Properties

Database properties may be specified

- as system properties with `-D`<*property*>
- in `hibernate.properties`
- in a named properties file with `--properties`

The needed properties are:

Property Name	Description
`hibernate.connection.driver_class`	jdbc driver class
`hibernate.connection.url`	jdbc url
`hibernate.connection.username`	database user
`hibernate.connection.password`	user password
`hibernate.dialect`	dialect

Table 21.3. SchemaExport Connection Properties

1.4. Using Ant

You can call `SchemaExport` from your Ant build script:

```
<target name="schemaexport">
    <taskdef name="schemaexport"
        classname="org.hibernate.tool.hbm2ddl.SchemaExportTask"
        classpathref="class.path"/>

    <schemaexport
        properties="hibernate.properties"
        quiet="no"
        text="no"
        drop="no"
        delimiter=";"
        output="schema-export.sql">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaexport>
</target>
```

1.5. Incremental schema updates

The `SchemaUpdate` tool will update an existing schema with "incremental" changes. Note that `SchemaUpdate` depends heavily upon the JDBC metadata API, so it will not work with all JDBC drivers.

`java -cp` *hibernate_classpaths*`org.hibernate.tool.hbm2ddl.SchemaUpdate`*options mapping_files*

Option	Description
`--quiet`	don't output the script to stdout
`--text`	don't export the script to the database
`--naming=eg.MyNamingStrategy`	select a `NamingStrategy`
`--properties=hibernate.properties`	read database properties from a file
`--config=hibernate.cfg.xml`	specify a `.cfg.xml` file

Table 21.4. `SchemaUpdate` Command Line Options

You may embed `SchemaUpdate` in your application:

```
Configuration cfg = ....;
new SchemaUpdate(cfg).execute(false);
```

1.6. Using Ant for incremental schema updates

You can call `SchemaUpdate` from the Ant script:

```
<target name="schemaupdate">
    <taskdef name="schemaupdate"
        classname="org.hibernate.tool.hbm2ddl.SchemaUpdateTask"
        classpathref="class.path"/>

    <schemaupdate
        properties="hibernate.properties"
        quiet="no">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaupdate>
</target>
```

1.7. Schema validation

The `SchemaValidator` tool will validate that the existing database schema "matches" your mapping documents. Note that `SchemaValidator` depends heavily upon the JDBC metadata API, so it will not work with all JDBC drivers. This tool is extremely useful for testing.

`java -cp` *hibernate_classpaths*`org.hibernate.tool.hbm2ddl.SchemaValidator`*options mapping_files*

Option	Description
`--naming=eg.MyNamingStrategy`	select a `NamingStrategy`
`--properties=hibernate.properties`	read database properties from a file
`--config=hibernate.cfg.xml`	specify a `.cfg.xml` file

Table 21.5. `SchemaValidator` Command Line Options

You may embed `SchemaValidator` in your application:

```
Configuration cfg = ....;
new SchemaValidator(cfg).validate();
```

1.8. Using Ant for schema validation

You can call `SchemaValidator` from the Ant script:

```
<target name="schemavalidate">
    <taskdef name="schemavalidator"
        classname="org.hibernate.tool.hbm2ddl.SchemaValidatorTask"
        classpathref="class.path"/>

    <schemavalidator
        properties="hibernate.properties">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaupdate>
</target>
```

Example: Parent/Child

One of the very first things that new users try to do with Hibernate is to model a parent / child type relationship. There are two different approaches to this. For various reasons the most convenient approach, especially for new users, is to model both `Parent` and `Child` as entity classes with a `<one-to-many>` association from `Parent` to `Child`. (The alternative approach is to declare the `Child` as a `<composite-element>`.) Now, it turns out that default semantics of a one to many association (in Hibernate) are much less close to the usual semantics of a parent / child relationship than those of a composite element mapping. We will explain how to use a *bidirectional one to many association with cascades* to model a parent / child relationship efficiently and elegantly. It's not at all difficult!

1. A note about collections

Hibernate collections are considered to be a logical part of their owning entity; never of the contained entities. This is a crucial distinction! It has the following consequences:

- When we remove / add an object from / to a collection, the version number of the collection owner is incremented.
- If an object that was removed from a collection is an instance of a value type (eg, a composite element), that object will cease to be persistent and its state will be completely removed from the database. Likewise, adding a value type instance to the collection will cause its state to be immediately persistent.
- On the other hand, if an entity is removed from a collection (a one-to-many or many-to-many association), it will not be deleted, by default. This behaviour is completely consistent - a change to the internal state of another entity should not cause the associated entity to vanish! Likewise, adding an entity to a collection does not cause that entity to become persistent, by default.

Instead, the default behaviour is that adding an entity to a collection merely creates a link between the two entities, while removing it removes the link. This is very appropriate for all sorts of cases. Where it is not appropriate at all is the case of a parent / child relationship, where the life of the child is bound to the lifecycle of the parent.

2. Bidirectional one-to-many

Suppose we start with a simple `<one-to-many>` association from `Parent` to `Child`.

```
<set name="children">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

If we were to execute the following code

```
Parent p = .....;
Child c = new Child();
p.getChildren().add(c);
session.save(c);
session.flush();
```

Hibernate would issue two SQL statements:

- an `INSERT` to create the record for `c`
- an `UPDATE` to create the link from `p` to `c`

This is not only inefficient, but also violates any `NOT NULL` constraint on the `parent_id` column. We can fix the nullability constraint violation by specifying `not-null="true"` in the collection mapping:

```
<set name="children">
    <key column="parent_id" not-null="true"/>
    <one-to-many class="Child"/>
</set>
```

However, this is not the recommended solution.

The underlying cause of this behaviour is that the link (the foreign key `parent_id`) from `p` to `c` is not considered part of the state of the `Child` object and is therefore not created in the `INSERT`. So the solution is to make the link part of the `Child` mapping.

```
<many-to-one name="parent" column="parent_id" not-null="true"/>
```

(We also need to add the `parent` property to the `Child` class.)

Now that the `Child` entity is managing the state of the link, we tell the collection not to update the link. We use the `inverse` attribute.

```
<set name="children" inverse="true">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

The following code would be used to add a new `Child`

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
c.setParent(p);
```

```
p.getChildren().add(c);
session.save(c);
session.flush();
```

And now, only one SQL `INSERT` would be issued!

To tighten things up a bit, we could create an `addChild()` method of `Parent`.

```
public void addChild(Child c) {
    c.setParent(this);
    children.add(c);
}
```

Now, the code to add a `Child` looks like

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.save(c);
session.flush();
```

3. Cascading lifecycle

The explicit call to `save()` is still annoying. We will address this by using cascades.

```
<set name="children" inverse="true" cascade="all">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

This simplifies the code above to

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.flush();
```

Similarly, we don't need to iterate over the children when saving or deleting a `Parent`. The following removes `p` and all its children from the database.

```
Parent p = (Parent) session.load(Parent.class, pid);
session.delete(p);
session.flush();
```

However, this code

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
c.setParent(null);
session.flush();
```

will not remove `c` from the database; it will ony remove the link to `p` (and cause a `NOT NULL` constraint violation, in this case). You need to explicitly `delete()` the `Child`.

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
session.delete(c);
session.flush();
```

Now, in our case, a `Child` can't really exist without its parent. So if we remove a `Child` from the collection, we really do want it to be deleted. For this, we must use `cascade="all-delete-orphan"`.

```
<set name="children" inverse="true" cascade="all-delete-orphan">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

Note: even though the collection mapping specifies `inverse="true"`, cascades are still processed by iterating the collection elements. So if you require that an object be saved, deleted or updated by cascade, you must add it to the collection. It is not enough to simply call `setParent()`.

4. Cascades and unsaved-value

Suppose we loaded up a `Parent` in one `Session`, made some changes in a UI action and wish to persist these changes in a new session by calling `update()`. The `Parent` will contain a collection of childen and, since cascading update is enabled, Hibernate needs to know which children are newly instantiated and which represent existing rows in the database. Lets assume that both `Parent` and `Child` have genenerated identifier properties of type `Long`. Hibernate will use the identifier and version/timestamp property value to determine which of the children are new. (See *Section 7, “Automatic state detection”*.) *In Hibernate3, it is no longer necessary to specify an `unsaved-value` explicitly.*

The following code will update `parent` and `child` and insert `newChild`.

```
//parent and child were both loaded in a previous session
parent.addChild(child);
Child newChild = new Child();
parent.addChild(newChild);
session.update(parent);
session.flush();
```

Well, that's all very well for the case of a generated identifier, but what about assigned identifiers and composite identifiers? This is more difficult, since Hibernate can't use the identifier property to distinguish between a newly instantiated object (with an identifier assigned by the user) and an object loaded in a previous session. In this case, Hibernate will either use the timestamp or version property, or will actually query the second-level cache or, worst case, the database, to see if the row exists.

5. Conclusion

There is quite a bit to digest here and it might look confusing first time around. However, in practice, it all works out very nicely. Most Hibernate applications use the parent / child pattern in many places.

We mentioned an alternative in the first paragraph. None of the above issues exist in the case of `<composite-element>` mappings, which have exactly the semantics of a parent / child relationship. Unfortunately, there are two big limitations to composite element classes: composite elements may not own collections, and they should not be the child of any entity other than the unique parent.

Example: Weblog Application

1. Persistent Classes

The persistent classes represent a weblog, and an item posted in a weblog. They are to be modelled as a standard parent/child relationship, but we will use an ordered bag, instead of a set.

```
package eg;

import java.util.List;

public class Blog {
    private Long _id;
    private String _name;
    private List _items;

    public Long getId() {
        return _id;
    }
    public List getItems() {
        return _items;
    }
    public String getName() {
        return _name;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setItems(List list) {
        _items = list;
    }
    public void setName(String string) {
        _name = string;
    }
}
```

```
package eg;

import java.text.DateFormat;
import java.util.Calendar;

public class BlogItem {
    private Long _id;
    private Calendar _datetime;
    private String _text;
    private String _title;
    private Blog _blog;

    public Blog getBlog() {
        return _blog;
    }
    public Calendar getDatetime() {
        return _datetime;
    }
    public Long getId() {
        return _id;
    }
    public String getText() {
        return _text;
    }
    public String getTitle() {
        return _title;
    }
    public void setBlog(Blog blog) {
        _blog = blog;
    }
    public void setDatetime(Calendar calendar) {
        _datetime = calendar;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setText(String string) {
        _text = string;
    }
    public void setTitle(String string) {
        _title = string;
    }
}
```

2. Hibernate Mappings

The XML mappings should now be quite straightforward.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

    <class
        name="Blog"
        table="BLOGS">

        <id
            name="id"
            column="BLOG_ID">

            <generator class="native"/>

        </id>

        <property
            name="name"
            column="NAME"
            not-null="true"
            unique="true"/>

        <bag
            name="items"
            inverse="true"
            order-by="DATE_TIME"
            cascade="all">

            <key column="BLOG_ID"/>
            <one-to-many class="BlogItem"/>

        </bag>

    </class>

</hibernate-mapping>
```

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

    <class
        name="BlogItem"
        table="BLOG_ITEMS"
        dynamic-update="true">

        <id
            name="id"
            column="BLOG_ITEM_ID">

            <generator class="native"/>

        </id>

        <property
            name="title"
            column="TITLE"
            not-null="true"/>

        <property
            name="text"
            column="TEXT"
            not-null="true"/>

        <property
            name="datetime"
            column="DATE_TIME"
            not-null="true"/>

        <many-to-one
            name="blog"
            column="BLOG_ID"
            not-null="true"/>

    </class>

</hibernate-mapping>
```

3. Hibernate Code

The following class demonstrates some of the kinds of things we can do with these classes, using Hibernate.

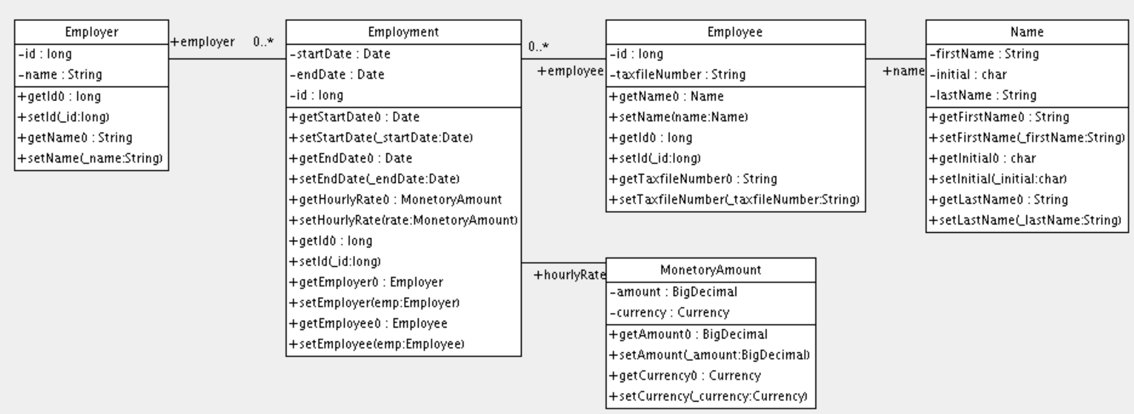
```
package eg;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.Iterator;
import java.util.List;

import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.tool.hbm2ddl.SchemaExport;

public class BlogMain {

    private SessionFactory _sessions;

    public void configure() throws HibernateException {
        _sessions = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class)
            .buildSessionFactory();
    }

    public void exportTables() throws HibernateException {
        Configuration cfg = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class);
        new SchemaExport(cfg).create(true, true);
    }

    public Blog createBlog(String name) throws HibernateException {

        Blog blog = new Blog();
        blog.setName(name);
        blog.setItems( new ArrayList() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.persist(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }

    public BlogItem createBlogItem(Blog blog, String title, String text)
                        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setBlog(blog);
        item.setDatetime( Calendar.getInstance() );
        blog.getItems().add(item);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public BlogItem createBlogItem(Long blogid, String title, String text)
                        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setDatetime( Calendar.getInstance() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            Blog blog = (Blog) session.load(Blog.class, blogid);
            item.setBlog(blog);
            blog.getItems().add(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public void updateBlogItem(BlogItem item, String text)
                    throws HibernateException {

        item.setText(text);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }

    public void updateBlogItem(Long itemid, String text)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            BlogItem item = (BlogItem) session.load(BlogItem.class, itemid);
            item.setText(text);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }

    public List listAllBlogNamesAndItemCounts(int max)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "select blog.id, blog.name, count(blogItem) " +
                "from Blog as blog " +
                "left outer join blog.items as blogItem " +
                "group by blog.name, blog.id " +
                "order by max(blogItem.datetime)"
            );
            q.setMaxResults(max);
            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }

    public Blog getBlogAndAllItems(Long blogid)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        Blog blog = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "left outer join fetch blog.items " +
                "where blog.id = :blogid"
            );
            q.setParameter("blogid", blogid);
            blog  = (Blog) q.uniqueResult();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }

    public List listBlogsAndRecentItems() throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "inner join blog.items as blogItem " +
                "where blogItem.datetime > :minDate"
            );

            Calendar cal = Calendar.getInstance();
            cal.roll(Calendar.MONTH, false);
            q.setCalendar("minDate", cal);

            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }
}
```

Example: Various Mappings

This chapters shows off some more complex association mappings.

1. Employer/Employee

The following model of the relationship between `Employer` and `Employee` uses an actual entity class (`Employment`) to represent the association. This is done because there might be more than one period of employment for the same two parties. Components are used to model monetary values and employee names.



Heres a possible mapping document:

```
<hibernate-mapping>

    <class name="Employer" table="employers">
        <id name="id">
            <generator class="sequence">
                <param name="sequence">employer_id_seq</param>
            </generator>
        </id>
        <property name="name"/>
    </class>

    <class name="Employment" table="employment_periods">

        <id name="id">
            <generator class="sequence">
                <param name="sequence">employment_id_seq</param>
            </generator>
        </id>
        <property name="startDate" column="start_date"/>
        <property name="endDate" column="end_date"/>

        <component name="hourlyRate" class="MonetaryAmount">
            <property name="amount">
                <column name="hourly_rate" sql-type="NUMERIC(12, 2)"/>
            </property>
            <property name="currency" length="12"/>
        </component>
        <many-to-one name="employer" column="employer_id" not-null="true"/>
        <many-to-one name="employee" column="employee_id" not-null="true"/>

    </class>

    <class name="Employee" table="employees">
        <id name="id">
            <generator class="sequence">
                <param name="sequence">employee_id_seq</param>
            </generator>
        </id>
        <property name="taxfileNumber"/>
        <component name="name" class="Name">
            <property name="firstName"/>
            <property name="initial"/>
            <property name="lastName"/>
        </component>
    </class>

</hibernate-mapping>
```

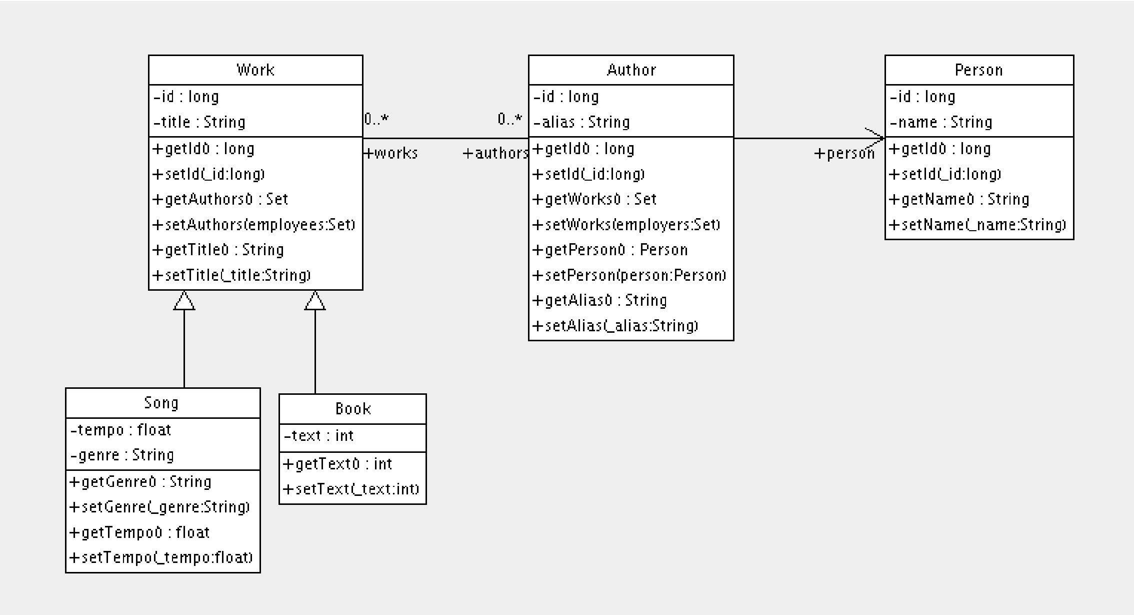
And heres the table schema generated by `SchemaExport`.

```
create table employers (
    id BIGINT not null,
    name VARCHAR(255),
    primary key (id)
)

create table employment_periods (
    id BIGINT not null,
    hourly_rate NUMERIC(12, 2),
    currency VARCHAR(12),
    employee_id BIGINT not null,
    employer_id BIGINT not null,
    end_date TIMESTAMP,
    start_date TIMESTAMP,
    primary key (id)
)

create table employees (
    id BIGINT not null,
    firstName VARCHAR(255),
    initial CHAR(1),
    lastName VARCHAR(255),
    taxfileNumber VARCHAR(255),
    primary key (id)
)

alter table employment_periods
    add constraint employment_periodsFK0 foreign key (employer_id)
references employers
alter table employment_periods
    add constraint employment_periodsFK1 foreign key (employee_id)
```

```
references employees
create sequence employee_id_seq
create sequence employment_id_seq
create sequence employer_id_seq
```

2. Author/Work

Consider the following model of the relationships between `Work`, `Author` and `Person`. We represent the relationship between `Work` and `Author` as a many-to-many association. We choose to represent the relationship between `Author` and `Person` as one-to-one association. Another possibility would be to have `Author` extend `Person`.



The following mapping document correctly represents these relationships:

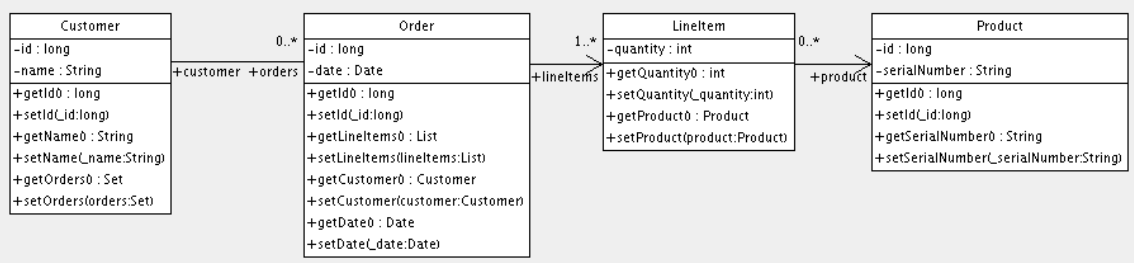
```
<hibernate-mapping>

    <class name="Work" table="works" discriminator-value="W">

        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <discriminator column="type" type="character"/>

        <property name="title"/>
        <set name="authors" table="author_work">
            <key column name="work_id"/>
            <many-to-many class="Author" column name="author_id"/>
        </set>

        <subclass name="Book" discriminator-value="B">
            <property name="text"/>
        </subclass>
        <subclass name="Song" discriminator-value="S">
            <property name="tempo"/>
            <property name="genre"/>
        </subclass>

    </class>

    <class name="Author" table="authors">

        <id name="id" column="id">
            <!-- The Author must have the same identifier as the Person -->
            <generator class="assigned"/>
        </id>

        <property name="alias"/>
        <one-to-one name="person" constrained="true"/>

        <set name="works" table="author_work" inverse="true">
            <key column="author_id"/>
            <many-to-many class="Work" column="work_id"/>
        </set>

    </class>

    <class name="Person" table="persons">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

There are four tables in this mapping. `works`, `authors` and `persons` hold work, author and person data respectively. `author_work` is an association table linking authors to works. Heres the table schema, as generated by `SchemaExport`.

```
create table works (
    id BIGINT not null generated by default as identity,
    tempo FLOAT,
    genre VARCHAR(255),
    text INTEGER,
    title VARCHAR(255),
    type CHAR(1) not null,
    primary key (id)
)

create table author_work (
    author_id BIGINT not null,
    work_id BIGINT not null,
    primary key (work_id, author_id)
)

create table authors (
    id BIGINT not null generated by default as identity,
    alias VARCHAR(255),
    primary key (id)
)

create table persons (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

alter table authors
    add constraint authorsFK0 foreign key (id) references persons
alter table author_work
    add constraint author_workFK0 foreign key (author_id) references authors
alter table author_work
    add constraint author_workFK1 foreign key (work_id) references works
```

3. Customer/Order/Product

Now consider a model of the relationships between `Customer`, `Order` and `LineItem` and `Product`. There is a one-to-many association between `Customer` and `Order`, but how should we represent `Order` / `LineItem` / `Product`? I've chosen to map `LineItem` as an association class representing the many-to-many association between `Order` and `Product`. In Hibernate, this is called a composite element.



The mapping document:

```
<hibernate-mapping>

    <class name="Customer" table="customers">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
        <set name="orders" inverse="true">
            <key column="customer_id"/>
            <one-to-many class="Order"/>
        </set>
    </class>

    <class name="Order" table="orders">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="date"/>
        <many-to-one name="customer" column="customer_id"/>
        <list name="lineItems" table="line_items">
            <key column="order_id"/>
            <list-index column="line_number"/>
            <composite-element class="LineItem">
                <property name="quantity"/>
                <many-to-one name="product" column="product_id"/>
            </composite-element>
        </list>
    </class>

    <class name="Product" table="products">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="serialNumber"/>
    </class>

</hibernate-mapping>
```

`customers`, `orders`, `line_items` and `products` hold customer, order, order line item and product data respectively. `line_items` also acts as an association table linking orders with products.

```
create table customers (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

create table orders (
    id BIGINT not null generated by default as identity,
    customer_id BIGINT,
    date TIMESTAMP,
    primary key (id)
)

create table line_items (
    line_number INTEGER not null,
    order_id BIGINT not null,
    product_id BIGINT,
    quantity INTEGER,
    primary key (order_id, line_number)
)

create table products (
    id BIGINT not null generated by default as identity,
    serialNumber VARCHAR(255),
    primary key (id)
)

alter table orders
```

```
    add constraint ordersFK0 foreign key (customer_id) references customers
alter table line_items
    add constraint line_itemsFK0 foreign key (product_id) references
products
alter table line_items
    add constraint line_itemsFK1 foreign key (order_id) references orders
```

4. Miscellaneous example mappings

These examples are all taken from the Hibernate test suite. You will find many other useful example mappings there. Look in the `test` folder of the Hibernate distribution.

TODO: put words around this stuff

4.1. "Typed" one-to-one association

```
<class name="Person">
    <id name="name"/>
    <one-to-one name="address"
            cascade="all">
        <formula>name</formula>
        <formula>'HOME'</formula>
    </one-to-one>
    <one-to-one name="mailingAddress"
            cascade="all">
        <formula>name</formula>
        <formula>'MAILING'</formula>
    </one-to-one>
</class>

<class name="Address" batch-size="2"
        check="addressType in ('MAILING', 'HOME', 'BUSINESS')">
    <composite-id>
        <key-many-to-one name="person"
                column="personName"/>
        <key-property name="type"
                column="addressType"/>
    </composite-id>
    <property name="street" type="text"/>
    <property name="state"/>
    <property name="zip"/>
</class>
```

4.2. Composite key example

```
<class name="Customer">

    <id name="customerId"
        length="10">
        <generator class="assigned"/>
    </id>
    <property name="name" not-null="true" length="100"/>
    <property name="address" not-null="true" length="200"/>

    <list name="orders"
            inverse="true"
            cascade="save-update">
        <key column="customerId"/>
        <index column="orderNumber"/>
        <one-to-many class="Order"/>
    </list>

</class>

<class name="Order" table="CustomerOrder" lazy="true">
    <synchronize table="LineItem"/>
    <synchronize table="Product"/>

    <composite-id name="id"
            class="Order$Id">
        <key-property name="customerId" length="10"/>
        <key-property name="orderNumber"/>
    </composite-id>

    <property name="orderDate"
            type="calendar_date"
            not-null="true"/>

    <property name="total">
        <formula>
            ( select sum(li.quantity*p.price)
            from LineItem li, Product p
            where li.productId = p.productId
                and li.customerId = customerId
                and li.orderNumber = orderNumber )
        </formula>
    </property>

    <many-to-one name="customer"
            column="customerId"
            insert="false"
            update="false"
            not-null="true"/>

    <bag name="lineItems"
            fetch="join"
            inverse="true"
            cascade="save-update">
        <key>
            <column name="customerId"/>
            <column name="orderNumber"/>
        </key>
        <one-to-many class="LineItem"/>
    </bag>

</class>
```

```
<class name="LineItem">

    <composite-id name="id"
            class="LineItem$Id">
        <key-property name="customerId" length="10"/>
        <key-property name="orderNumber"/>
        <key-property name="productId" length="10"/>
    </composite-id>

    <property name="quantity"/>

    <many-to-one name="order"
            insert="false"
            update="false"
            not-null="true">
        <column name="customerId"/>
        <column name="orderNumber"/>
    </many-to-one>

    <many-to-one name="product"
            insert="false"
            update="false"
            not-null="true"
            column="productId"/>

</class>

<class name="Product">
    <synchronize table="LineItem"/>

    <id name="productId"
        length="10">
        <generator class="assigned"/>
    </id>

    <property name="description"
        not-null="true"
        length="200"/>
    <property name="price" length="3"/>
    <property name="numberAvailable"/>

    <property name="numberOrdered">
        <formula>
            ( select sum(li.quantity)
            from LineItem li
            where li.productId = productId )
        </formula>
    </property>

</class>
```

4.3. Many-to-many with shared composite key attribute

```
<class name="User" table="`User`">
```

```
    <composite-id>
        <key-property name="name"/>
        <key-property name="org"/>
    </composite-id>
    <set name="groups" table="UserGroup">
        <key>
            <column name="userName"/>
            <column name="org"/>
        </key>
        <many-to-many class="Group">
            <column name="groupName"/>
            <formula>org</formula>
        </many-to-many>
    </set>
</class>

<class name="Group" table="`Group`">
    <composite-id>
        <key-property name="name"/>
        <key-property name="org"/>
    </composite-id>
    <property name="description"/>
    <set name="users" table="UserGroup" inverse="true">
        <key>
            <column name="groupName"/>
            <column name="org"/>
        </key>
        <many-to-many class="User">
            <column name="userName"/>
            <formula>org</formula>
        </many-to-many>
    </set>
</class>
```

4.4. Content based discrimination

```
<class name="Person"
    discriminator-value="P">

    <id name="id"
        column="person_id"
        unsaved-value="0">
        <generator class="native"/>
    </id>


    <discriminator
        type="character">
        <formula>
            case
                when title is not null then 'E'
                when salesperson is not null then 'C'
                else 'P'
            end
        </formula>
    </discriminator>

    <property name="name"
        not-null="true"
        length="80"/>

    <property name="sex"
        not-null="true"
        update="false"/>

    <component name="address">
        <property name="address"/>
        <property name="zip"/>
        <property name="country"/>
    </component>

    <subclass name="Employee"
        discriminator-value="E">
            <property name="title"
                length="20"/>
            <property name="salary"/>
            <many-to-one name="manager"/>
    </subclass>

    <subclass name="Customer"
        discriminator-value="C">
            <property name="comments"/>
            <many-to-one name="salesperson"/>
    </subclass>

</class>
```

4.5. Associations on alternate keys

```
<class name="Person">

    <id name="id">
        <generator class="hilo"/>
    </id>

    <property name="name" length="100"/>

    <one-to-one name="address"
        property-ref="person"
        cascade="all"
        fetch="join"/>

    <set name="accounts"
        inverse="true">
        <key column="userId"
            property-ref="userId"/>
        <one-to-many class="Account"/>
    </set>
    <property name="userId" length="8"/>

</class>

<class name="Address">

    <id name="id">
        <generator class="hilo"/>
    </id>

    <property name="address" length="300"/>
    <property name="zip" length="5"/>
    <property name="country" length="25"/>
    <many-to-one name="person" unique="true" not-null="true"/>

</class>

<class name="Account">
    <id name="accountId" length="32">
        <generator class="uuid"/>
    </id>

    <many-to-one name="user"
        column="userId"
        property-ref="userId"/>

    <property name="type" not-null="true"/>

</class>
```

Best Practices

Write fine-grained classes and map them using `<component>`.
: Use an `Address` class to encapsulate `street`, `suburb`, `state`, `postcode`. This encourages code reuse and simplifies refactoring.

Declare identifier properties on persistent classes.
: Hibernate makes identifier properties optional. There are all sorts of reasons why you should use them. We recommend that identifiers be 'synthetic' (generated, with no business meaning).

Identify natural keys.
: Identify natural keys for all entities, and map them using `<natural-id>`. Implement `equals()` and `hashCode()` to compare the properties that make up the natural key.

Place each class mapping in its own file.
: Don't use a single monolithic mapping document. Map `com.eg.Foo` in the file `com/eg/Foo.hbm.xml`. This makes particularly good sense in a team environment.

Load mappings as resources.
: Deploy the mappings along with the classes they map.

Consider externalising query strings.
: This is a good practice if your queries call non-ANSI-standard SQL functions. Externalising the query strings to mapping files will make the application more portable.

Use bind variables.
: As in JDBC, always replace non-constant values by "?". Never use string manipulation to bind a non-constant value in a query! Even better, consider using named parameters in queries.

Don't manage your own JDBC connections.
: Hibernate lets the application manage JDBC connections. This approach should be considered a last-resort. If you can't use the built-in connections providers, consider providing your own implementation of `org.hibernate.connection.ConnectionProvider`.

Consider using a custom type.
: Suppose you have a Java type, say from some library, that needs to be persisted but doesn't provide the accessors needed to map it as a component. You should consider implementing `org.hibernate.UserType`. This approach frees the application code from implementing transformations to / from a Hibernate type.

Use hand-coded JDBC in bottlenecks.
: In performance-critical areas of the system, some kinds of operations might benefit from direct JDBC. But please, wait until you *know* something is a bottleneck. And don't assume that direct JDBC is necessarily faster. If you need to use direct JDBC, it might be worth opening a Hibernate `Session` and using that JDBC connection. That way you can still use

the same transaction strategy and underlying connection provider.

Understand `Session` flushing.

From time to time the Session synchronizes its persistent state with the database. Performance will be affected if this process occurs too often. You may sometimes minimize unnecessary flushing by disabling automatic flushing or even by changing the order of queries and other operations within a particular transaction.

In a three tiered architecture, consider using detached objects.

When using a servlet / session bean architecture, you could pass persistent objects loaded in the session bean to and from the servlet / JSP layer. Use a new session to service each request. Use `Session.merge()` or `Session.saveOrUpdate()` to synchronize objects with the database.

In a two tiered architecture, consider using long persistence contexts.

Database Transactions have to be as short as possible for best scalability. However, it is often neccessary to implement long running *application transactions*, a single unit-of-work from the point of view of a user. An application transaction might span several client request/response cycles. It is common to use detached objects to implement application transactions. An alternative, extremely appropriate in two tiered architecture, is to maintain a single open persistence contact (session) for the whole lifecycle of the application transaction and simply disconnect from the JDBC connection at the end of each request and reconnect at the beginning of the subsequent request. Never share a single session across more than one application transaction, or you will be working with stale data.

Don't treat exceptions as recoverable.

This is more of a necessary practice than a "best" practice. When an exception occurs, roll back the `Transaction` and close the `Session`. If you don't, Hibernate can't guarantee that in-memory state accurately represents persistent state. As a special case of this, do not use `Session.load()` to determine if an instance with the given identifier exists on the database; use `Session.get()` or a query instead.

Prefer lazy fetching for associations.

Use eager fetching sparingly. Use proxies and lazy collections for most associations to classes that are not likely to be completely held in the second-level cache. For associations to cached classes, where there is an a extremely high probability of a cache hit, explicitly disable eager fetching using `lazy="false"`. When an join fetching is appropriate to a particular use case, use a query with a `left join fetch`.

Use the *open session in view* pattern, or a disciplined *assembly phase* to avoid problems with unfetched data.

Hibernate frees the developer from writing tedious *Data Transfer Objects* (DTO). In a traditional EJB architecture, DTOs serve dual purposes: first, they work around the problem that entity beans are not serializable; second, they implicitly define an assembly phase where all data to be used by the view is fetched and marshalled into the DTOs before returning control to the presentation tier. Hibernate eliminates the first purpose. However, you will still need an assembly phase (think of your business methods as having a strict contract with the presentation tier about what data is available in the detached objects) unless you are prepared to hold the persistence context (the session) open across the view rendering process. This is not a limitation of Hibernate! It is a fundamental requirement of safe transactional data access.

Consider abstracting your business logic from Hibernate.

Hide (Hibernate) data-access code behind an interface. Combine the *DAO* and *Thread Local Session* patterns. You can even have some classes persisted by handcoded JDBC, associated to Hibernate via a `UserType`. (This advice is intended for "sufficiently large" applications; it is not appropriate for an application with five tables!)

Don't use exotic association mappings.

Good usecases for a real many-to-many associations are rare. Most of the time you need additional information stored in the "link table". In this case, it is much better to use two one-to-many associations to an intermediate link class. In fact, we think that most associations are one-to-many and many-to-one, you should be careful when using any other association style and ask yourself if it is really neccessary.

Prefer bidirectional associations.

Unidirectional associations are more difficult to query. In a large application, almost all associations must be navigable in both directions in queries.

Index