Hibernate Reference Documentation

Version: 2.1.6

Table of Contents

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 1, *Quickstart with Tomcat* for a 30 minute tutorial, using Tomcat.

2. Read Chapter 2, *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 source for design patterns and various integration solutions (Tomcat, JBoss, Spring, Struts, EJB, etc.).

8. An offline version of the Hibernate website is distributed with Hibernate in the `doc/wiki/` subdirectory.

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 project of the JBoss Professional Open Source product suite.

Chapter 1. Quickstart with Tomcat

1.1. Getting started with Hibernate

This tutorial explains a setup of Hibernate 2.1 with the Apache Tomcat servlet container for a web-based application. Hibernate works well in a managed environment with all major J2EE application servers, or even in standalone Java applications. The database system used in this tutorial is PostgreSQL 7.3, support for other database is only a matter of changing the Hibernate SQL dialect configuration.

First, we have to copy all required libraries to the Tomcat installation. We use a separate web context (`webapps/quickstart`) for this tutorial, so we've to consider both the global library search path (`TOMCAT/common/lib`) and the classloader at the context level in `webapps/quickstart/WEB-INF/lib` (for JAR files) and `webapps/quickstart/WEB-INF/classes`. We refer to both classloader levels as the global classpath and the context classpath.

Now, copy the libraries to the two classpaths:

1. Copy the JDBC driver for the database to the global classpath. This is required for the DBCP connection pool software which comes bundled with Tomcat. Hibernate uses JDBC connections to execute SQL on the database, so you either have to provide pooled JDBC connections or configure Hibernate to use one of the directly supported pools (C3P0, Proxool). For this tutorial, copy the `pg73jdbc3.jar` library (for PostgreSQL 7.3 and JDK 1.4) to the global classloaders path. If you'd like to use a different database, simply copy its appropriate JDBC driver.

2. Never copy anything else into the global classloader path in Tomcat, or you will get problems with various tools, including Log4j, commons-logging and others. Always use the context classpath for each web application, that is, copy libraries to `WEB-INF/lib` and your own classes and configuration/property files to `WEB-INF/classes`. Both directories are in the context level classpath by default.

3. Hibernate is packaged as a JAR library. The `hibernate2.jar` file should be copied in the context classpath together with other classes of the application. Hibernate requires some 3rd party libraries at runtime, these come bundled with the Hibernate distribution in the `lib/` directory; see Table 1.1, “ Hibernate 3rd party libraries ”. Copy the required 3rd party libraries to the context classpath.

Table 1.1. Hibernate 3rd party libraries

Library	Description
dom4j (required)	Hibernate uses dom4j to parse XML configuration and XML mapping metadata files.
CGLIB (required)	Hibernate uses the code generation library to enhance classes at runtime (in combination with Java reflection).
Commons Collections, Commons Logging (required)	Hibernate uses various utility libraries from the Apache Jakarta Commons project.
ODMG4 (required)	Hibernate provides an optional ODMG compliant persistence manager interface. It is required if you like to map collections, even if you don't intend to use the ODMG API. We don't map collections in this tutorial, but it's a good idea to copy the JAR anyway.

Library	Description
EHCache (required)	Hibernate can use various cache providers for the second-level cache. EHCache is the default cache provider if not changed in the configuration.
Log4j (optional)	Hibernate uses the Commons Logging API, which in turn can use Log4j as the underlying logging mechanism. If the Log4j library is available in the context library directory, Commons Logging will use Log4j and the `log4j.properties` configuration in the context classpath. An example properties file for Log4j is bundled with the Hibernate distribution. So, copy log4j.jar and the configuration file (from `src/`) to your context classpath if you want to see whats going on behind the scenes.
Required or not?	Have a look at the file `lib/README.txt` in the Hibernate distribution. This is an up-to-date list of 3rd party libraries distributed with Hibernate. You will find all required and optional libraries listed there.

We now set up the database connection pooling and sharing in both Tomcat and Hibernate. This means Tomcat will provide pooled JDBC connections (using its builtin DBCP pooling feature), Hibernate requests theses connections through JNDI. Tomcat binds the connection pool to JNDI, we add a resource declaration to Tomcats main configuration file, `TOMCAT/conf/server.xml`:

```
<Context path="/quickstart" docBase="quickstart">
    <Resource name="jdbc/quickstart" scope="Shareable" type="javax.sql.DataSource"/>
    <ResourceParams name="jdbc/quickstart">
        <parameter>
            <name>factory</name>
            <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
        </parameter>

        <!-- DBCP database connection settings -->
        <parameter>
            <name>url</name>
            <value>jdbc:postgresql://localhost/quickstart</value>
        </parameter>
        <parameter>
            <name>driverClassName</name><value>org.postgresql.Driver</value>
        </parameter>
        <parameter>
            <name>username</name>
            <value>quickstart</value>
        </parameter>
        <parameter>
            <name>password</name>
            <value>secret</value>
        </parameter>

        <!-- DBCP connection pooling options -->
        <parameter>
            <name>maxWait</name>
            <value>3000</value>
        </parameter>
        <parameter>
            <name>maxIdle</name>
            <value>100</value>
        </parameter>
        <parameter>
            <name>maxActive</name>
            <value>10</value>
        </parameter>
    </ResourceParams>
</Context>
```

The context we configure in this example is named `quickstart`, its base is the `TOMCAT/webapp/quickstart` directory. To access any servlets, call the path `http://localhost:8080/quickstart` in your browser (of course, adding the name of the servlet as mapped in your `web.xml`). You may also go ahead and create a simple servlet now that has an empty `process()`

Tomcat uses the DBCP connection pool with this configuration and provides pooled JDBC `Connection`s through JNDI at `java:comp/env/jdbc/quickstart`. If you have trouble getting the connection pool running, refer to the Tomcat documentation. If you get JDBC driver exception messages, try to setup JDBC connection pool without Hibernate first. Tomcat & JDBC tutorials are available on the Web.

The next step is to configure Hibernate, using the connections from the JNDI bound pool. We use Hibernates XML based configuration. The basic approach, using properties, is equivalent in features, but doesn't offer any advantages. We use the XML configuration because it is usually more convenient. The XML configuration file is placed in the context classpath (`WEB-INF/classes`), as `hibernate.cfg.xml`:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
    PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">

<hibernate-configuration>

    <session-factory>

        <property name="connection.datasource">java:comp/env/jdbc/quickstart</property>
        <property name="show_sql">false</property>
        <property name="dialect">net.sf.hibernate.dialect.PostgreSQLDialect</property>

        <!-- Mapping files -->
        <mapping resource="Cat.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

We turn logging of SQL commands off and tell Hibernate what database SQL dialect is used and where to get the JDBC connections (by declaring the JNDI address of the Tomcat bound datasource pool). The dialect is a required setting, databases differ in their interpretation of the SQL "standard". Hibernate will take care of the differences and comes bundled with dialects for all major commercial and open source databases.

A `SessionFactory` is Hibernate's concept of a single datastore, multiple databases can be used by creating multiple XML configuration files and creating multiple `Configuration` and `SessionFactory` objects in your application.

The last element of the `hibernate.cfg.xml` declares `Cat.hbm.xml` as the name of a Hibernate XML mapping file for the persistent class `Cat`. This file contains the metadata for the mapping of the POJO class to a datbase table (or multiple tables). We'll come back to that file soon. Let's write the POJO class first and then declare the mapping metadata for it.

1.2. First persistent class

Hibernate works best with the Plain Old Java Objects (POJOs, sometimes called Plain Ordinary Java Objects) programming model for persistent classes. A POJO is much like a JavaBean, with properties of the class accessible via getter and setter methods, shielding the internal representation from the publicly visible interface:

```
package net.sf.hibernate.examples.quickstart;

public class Cat {
```

```
    private String id;
    private String name;
    private char sex;
    private float weight;

    public Cat() {
    }

    public String getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public char getSex() {
        return sex;
    }

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

    public float getWeight() {
        return weight;
    }

    public void setWeight(float weight) {
        this.weight = weight;
    }

}
```

Hibernate is not restricted in its usage of property types, all Java JDK types and primitives (like `String`, `char` and `Date`) can be mapped, including classes from the Java collections framework. You can map them as values, collections of values, or associations to other entities. The `id` is a special property that represents the database identifer (primary key) of that class, it is highly recommended for entities like a `Cat`. Hibernate can use identifiers only internally, but we would lose some of the flexibility in our application architecture.

No special interface has to be implemented for persistent classes nor do we have to subclass from a special root persistent class. Hibernate also doesn't use any build time processing, such as byte-code manipulation, it relies solely on Java reflection and runtime class enhancement (through CGLIB). So, without any dependency in the POJO class on Hibernate, we can map it to a database table.

1.3. Mapping the cat

The `Cat.hbm.xml` mapping file contains the metadata required for the object/relational mapping. The metadata includes declaration of persistent classes and the mapping of properties (to columns and foreign key relationships to other entities) to database tables.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
```

```
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>

    <class name="net.sf.hibernate.examples.quickstart.Cat" table="CAT">

        <!-- A 32 hex character is our surrogate key. It's automatically
            generated by Hibernate with the UUID pattern. -->
        <id name="id" type="string" unsaved-value="null" >
            <column name="CAT_ID" sql-type="char(32)" not-null="true"/>
            <generator class="uuid.hex"/>
        </id>

        <!-- A cat has to have a name, but it shouldn' be too long. -->
        <property name="name">
            <column name="NAME" length="16" not-null="true"/>
        </property>

        <property name="sex"/>

        <property name="weight"/>

    </class>

</hibernate-mapping>
```

Every persistent class should have an identifer attribute (actually, only classes representing entities, not dependent value objects, which are mapped as components of an entity). This property is used to distinguish persistent objects: Two cats are equal if `catA.getId().equals(catB.getId())` is true, this concept is called *database identity*. Hibernate comes bundled with various identifer generators for different scenarios (including native generators for database sequences, hi/lo identifier tables, and application assigned identifiers). We use the UUID generator (only recommended for testing, as integer surrogate keys generated by the database should be prefered) and also specify the column `CAT_ID` of the table `CAT` for the Hibernate generated identifier value (as a primary key of the table).

All other properties of `Cat` are mapped to the same table. In the case of the `name` property, we mapped it with an explicit database column declaration. This is especially useful when the database schema is automatically generated (as SQL DDL statements) from the mapping declaration with Hibernate's *SchemaExport* tool. All other properties are mapped using Hibernate's default settings, which is what you need most of the time. The table `CAT` in the database looks like this:

```
 Column |         Type          | Modifiers
--------+-----------------------+-----------
 cat_id | character(32)         | not null
 name   | character varying(16) | not null
 sex    | character(1)          |
 weight | real                  |
Indexes: cat_pkey primary key btree (cat_id)
```

You should now create this table in your database manually, and later read Chapter 15, *Toolset Guide* if you want to automate this step with the SchemaExport tool. This tool can create a full SQL DDL, including table definition, custom column type constraints, unique constraints and indexes.

1.4. Playing with cats

We're now ready to start Hibernate's `Session`. It is the *persistence manager*interface, we use it to store and retrieve `Cats` to and from the database. But first, we've to get a `Session` (Hibernate's unit-of-work) from the `SessionFactory`:

```
SessionFactory sessionFactory =
```

```
            new Configuration().configure().buildSessionFactory();
```

A `SessionFactory` is responsible for one database and may only use one XML configuration file (`hibernate.cfg.xml`). You can set other properties (and even change the mapping metadata) by accessing the `Configuration` *before* you build the `SessionFactory` (it is immutable). Where do we create the `SessionFactory` and how can we access it in our application?

A `SessionFactory` is usually only build once, e.g. at startup with a *load-on-startup* servlet. This also means you should not keep it in an instance variable in your servlets, but in some other location. Furthermore, we need some kind of *Singleton*, so we can access the `SessionFactory` easily in application code. The approach shown next solves both problems: configuration and easy access to a `SessionFactory`.

We implement a `HibernateUtil` helper class:

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.*;

public class HibernateUtil {

    private static Log log = LogFactory.getLog(HibernateUtil.class);

    private static final SessionFactory sessionFactory;

    static {
        try {
            // Create the SessionFactory
            sessionFactory = new Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            log.error("Initial SessionFactory creation failed.", ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static final ThreadLocal session = new ThreadLocal();

    public static Session currentSession() throws HibernateException {
        Session s = (Session) session.get();
        // Open a new Session, if this Thread has none yet
        if (s == null) {
            s = sessionFactory.openSession();
            session.set(s);
        }
        return s;
    }

    public static void closeSession() throws HibernateException {
        Session s = (Session) session.get();
        session.set(null);
        if (s != null)
            s.close();
    }
}
```

This class does not only take care of the `SessionFactory` with its static attribute, but also has a `ThreadLocal` to hold the `Session` for the current executing thread. Make sure you understand the Java concept of a thread-local variable before you try to use this helper.

A `SessionFactory` is threadsafe, many threads can access it concurrently and request `Sessions`. A `Session` is a non-threadsafe object that represents a single unit-of-work with the database. `Sessions` are opened by a `SessionFactory` and are closed when all work is completed:

```
Session session = HibernateUtil.currentSession();

Transaction tx= session.beginTransaction();
```

```
Cat princess = new Cat();
princess.setName("Princess");
princess.setSex('F');
princess.setWeight(7.4f);

session.save(princess);
tx.commit();

HibernateUtil.closeSession();
```

In a `Session`, every database operation occurs inside a transaction that isolates the database operations (even read-only operations). We use Hibernates `Transaction` API to abstract from the underlying transaction strategy (in our case, JDBC transactions). This allows our code to be deployed with container-managed transactions (using JTA) without any changes. Please note that the example above does not handle any exceptions.

Also note that you may call `HibernateUtil.currentSession();` as many times as you like, you will always get the current `Session` of this thread. You have to make sure the `Session` is closed after your unit-of-work completes, either in your servlet code or in a servlet filter before the HTTP response is send. The nice side effect of the latter is easy lazy initialization: the `Session` is still open when the view is rendered, so Hibernate can load unitialized objects while you navigate the graph.

Hibernate has various methods that can be used to retrieve objects from the database. The most flexible way is using the Hibernate Query Language (HQL), which is an easy to learn and powerful object-oriented extension to SQL:

```
Transaction tx = session.beginTransaction();

Query query = session.createQuery("select c from Cat as c where c.sex = :sex");
query.setCharacter("sex", 'F');
for (Iterator it = query.iterate(); it.hasNext();) {
    Cat cat = (Cat) it.next();
    out.println("Female Cat: " + cat.getName() );
}

tx.commit();
```

Hibernate also offers an object-oriented *query by criteria* API that can be used to formulate type-safe queries. Hibernate of course uses `PreparedStatements` and parameter binding for all SQL communication with the database. You may also use Hibernate's direct SQL query feature or get a plain JDBC connection from a `Session` in rare cases.

1.5. Finally

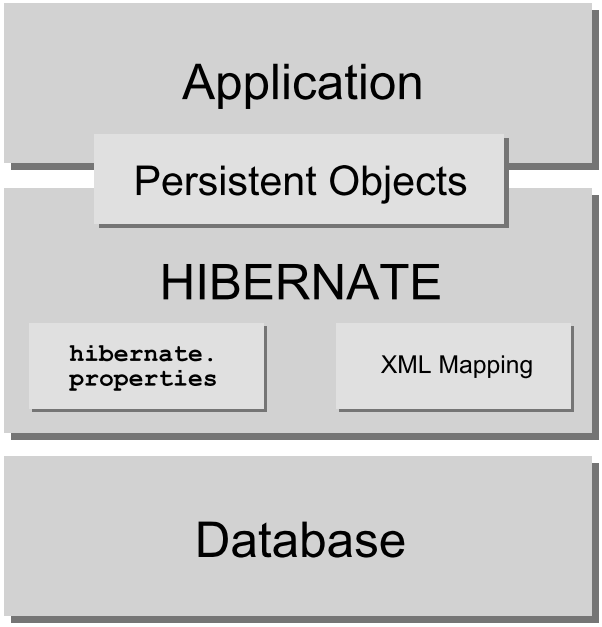
We only scratched the surface of Hibernate in this small tutorial. Please note that we don't include any servlet specific code in our examples. You have to create a servlet yourself and insert the Hibernate code as you see fit.

Keep in mind that Hibernate, as a data access layer, is tightly integrated into your application. Usually, all other layers depent on the persistence mechanism. Make sure you understand the implications of this design.

Chapter 2. Architecture

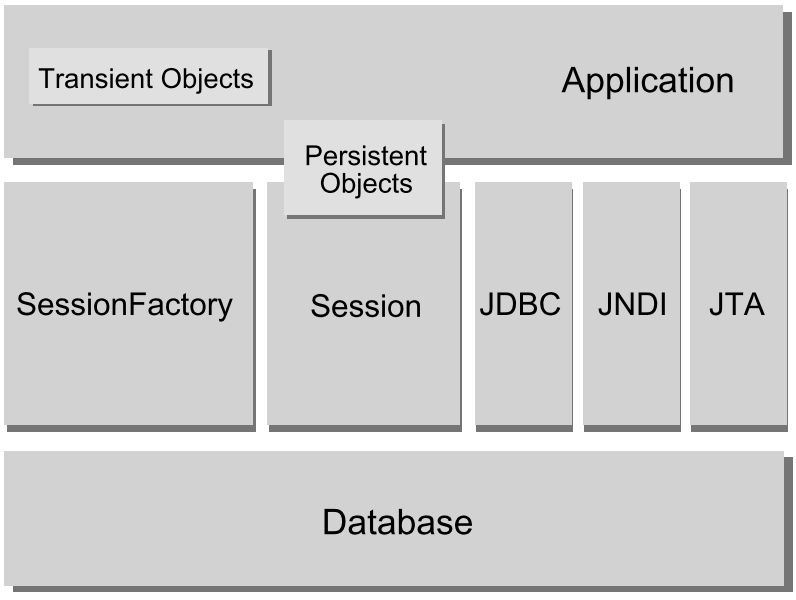
2.1. Overview

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



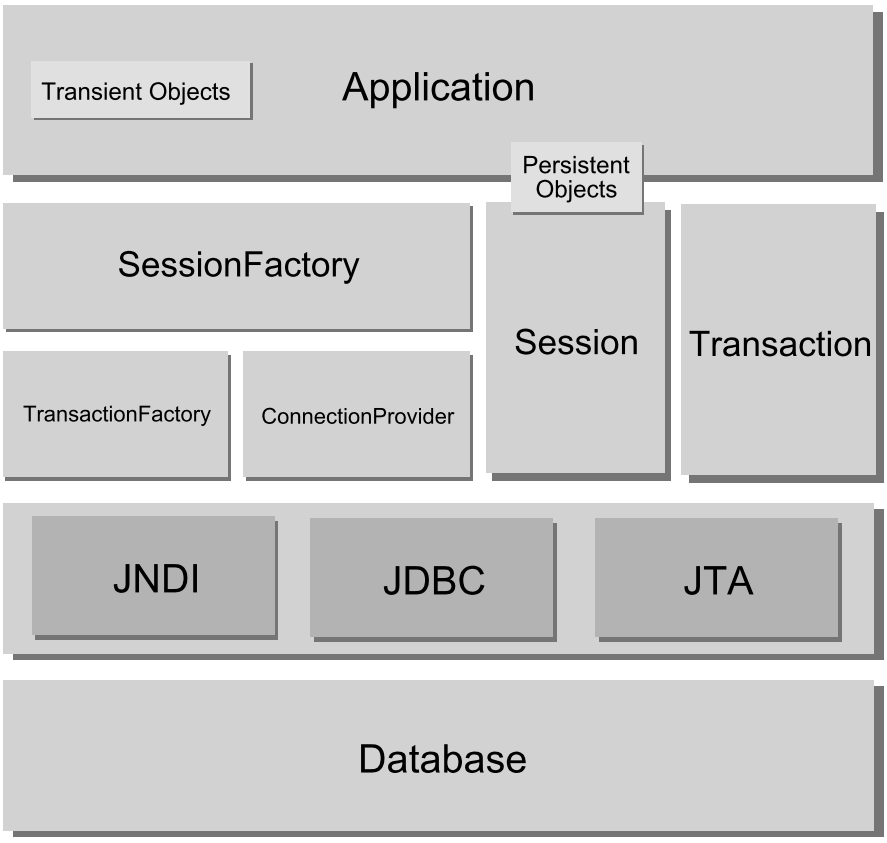
This diagram shows Hibernate using the database and configuration data to provide persistence services (and persistent objects) to the application.

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:



The "full cream" architecture abstracts the application away from the underlying JDBC/JTA APIs and lets Hi-

bernate take care of the details.



Heres some definitions of the objects in the diagrams:

SessionFactory (`net.sf.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 (`net.sf.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 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 (`net.sf.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 `Transactions` in some cases.

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

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

2.2. JMX Integration

JMX is the J2EE standard for management of Java components. Hibernate may be managed via a JMX standard MBean but because most application servers do not yet support JMX, Hibernate also affords some non-standard configuration mechanisms.

Please see the Hibernate website for more information on how to configure Hibernate to run as a JMX component inside JBoss.

2.3. JCA Support

Hibernate may also be configured as a JCA connector. Please see the website for more details.

Chapter 3. SessionFactory 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 that shows the various options. You usually only have to put that file in your classpath and customize it.

3.1. Programmatic Configuration

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

You may obtain a `Configuration` instance by instantiating it directly. Heres an example of setting up a datastore from mappings defined in two XML configuration files (in the classpath):

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

An alternative (sometimes better) way is to let Hibernate load a mapping file using `getResourceAsStream()`:

```
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/autcion/Item.hbm.xml` and `/org/hibernate/autcion/Bid.hbm.xml` in the classpath. This approach eliminates any hardcoded filenames.

A `Configuration` also specifies various optional properties:

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

A `Configuration` is intended as a configuration-time object, to be discarded once a `SessionFactory` is built.

3.2. 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();
```

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

3.3. User provided JDBC connection

A SessionFactory may open a Session on a user-provided JDBC connection. This design choice frees the application to obtain JDBC connections wherever it pleases:

```
java.sql.Connection conn = datasource.getConnection();
Session session = sessions.openSession(conn);

// do some data access work
```

The application must be careful not to open two concurrent Sessions on the same JDBC connection!

3.4. Hibernate provided JDBC connection

Alternatively, you can have the SessionFactory open connections for you. The SessionFactory must be provided with JDBC connection properties in one of the following ways:

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).

If you take this approach, opening a Session is as simple as:

```
Session session = sessions.openSession(); // open a new Session
// do some data access work, a JDBC connection will be used on demand
```

All Hibernate property names and semantics are defined on the class net.sf.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:

Table 3.1. Hibernate JDBC 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*

Hibernate's own connection pooling algorithm is 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. Use a third party pool for best performance and stability, i.e., replace the hibernate.connection.pool_size property with connection pool specific settings.

C3P0 is an open source JDBC connection pool distributed along with Hibernate in the lib directory. Hibernate will use the built-in C3P0ConnectionProvider for connection pooling if you set the hibernate.c3p0.* properties. There is also built-in support for Apache DBCP and for Proxool. You must set the properties hibernate.dbcp.* (DBCP connection pool properties) to enable the DBCPConnectionProvider. Prepared statement

caching is enabled (highly recommend) if `hibernate.dbcp.ps.*` (DBCP statement cache properties) are set. Please refer the the Apache commons-pool documentation for the interpretation of these properties. You should set the `hibernate.proxool.*` properties if you wish to use Proxool.

This is an example using 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 = net.sf.hibernate.dialect.PostgreSQLDialect
```

For use inside an application server, Hibernate may obtain connections from a `javax.sql.Datasource` registered in JNDI. Set the following properties:

Table 3.2. Hibernate Datasource 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)

This is an example using an application server provided JNDI datasource:

```
hibernate.connection.datasource = java:/comp/env/jdbc/MyDB
hibernate.transaction.factory_class = \
    net.sf.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    net.sf.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = \
    net.sf.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.connnection.charSet`.

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

3.5. 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.

System-level properties can only be set via `java -Dproperty=value` or be defined in `hibernate.properties` and not with an instance of `Properties` passed to the `Configuration`.

Table 3.3. Hibernate Configuration Properties

Property name	Purpose
`hibernate.dialect`	The classname of a Hibernate `Dialect` - enables certain platform dependent features. *eg.* `full.classname.of.Dialect`
`hibernate.default_schema`	Qualify unqualified tablenames with the given schema/tablespace in generated SQL. *eg.* `SCHEMA_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`
`hibernate.use_outer_join`	Enables outer join fetching. Deprecated, use `max_fetch_depth`. *eg.* `true` | `false`
`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.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. *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.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).

Property name	Purpose
	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.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`
`hibernate.jndi.<propertyName>`	Pass the property `propertyName` to the JNDI `InitialContextFactory`.
`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. *eg.* `1, 2, 4, 8`
`hibernate.connection.<propertyName>`	Pass the JDBC property `propertyName` to `DriverManager.getConnection()`.
`hibernate.connection.provider_class`	The classname of a custom `ConnectionProvider`. *eg.* `classname.of.ConnectionProvider`
`hibernate.cache.provider_class`	The classname of a custom `CacheProvider`. *eg.* `classname.of.CacheProvider`
`hibernate.cache.use_minimal_puts`	Optimize second-level cache operation to minimize writes, at the cost of more frequent reads (useful for clustered caches). *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.query_cache_factory`	The classname of a custom `QueryCache` interface, defaults to the built-in `StandardQueryCache`. *eg.* `classname.of.QueryCache`

Property name	Purpose
hibernate.cache.region_prefix	A prefix to use for second-level cache region names. *eg.* prefix
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. *eg.* jndi/composite/name
hibernate.transaction.manager_lookup_class	The classname of a TransactionManagerLookup - required when JVM-level caching is enabled in a JTA environment. *eg.* classname.of.TransactionManagerLookup
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.show_sql	Write all SQL statements to console. *eg.* true | false
hibernate.hbm2ddl.auto	Automatically export schema DDL to the database when the SessionFactory is created. With create-drop, the database schema will be dropped when the SessionFactory is closed explicitely. *eg.* update | create | create-drop

3.5.1. SQL Dialects

You should always set the hibernate.dialect property to the correct net.sf.hibernate.dialect.Dialect subclass for your database. This is not strictly essential unless you wish to use native or sequence primary key generation or pessimistic locking (with, eg. Session.lock() or Query.setLockMode()). However, 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.

Table 3.4. Hibernate SQL Dialects (hibernate.dialect)

RDBMS	Dialect
DB2	net.sf.hibernate.dialect.DB2Dialect

RDBMS	Dialect
DB2 AS/400	`net.sf.hibernate.dialect.DB2400Dialect`
DB2 OS390	`net.sf.hibernate.dialect.DB2390Dialect`
PostgreSQL	`net.sf.hibernate.dialect.PostgreSQLDialect`
MySQL	`net.sf.hibernate.dialect.MySQLDialect`
Oracle (any version)	`net.sf.hibernate.dialect.OracleDialect`
Oracle 9/10g	`net.sf.hibernate.dialect.Oracle9Dialect`
Sybase	`net.sf.hibernate.dialect.SybaseDialect`
Sybase Anywhere	`net.sf.hibernate.dialect.SybaseAnywhereDialect`
Microsoft SQL Server	`net.sf.hibernate.dialect.SQLServerDialect`
SAP DB	`net.sf.hibernate.dialect.SAPDBDialect`
Informix	`net.sf.hibernate.dialect.InformixDialect`
HypersonicSQL	`net.sf.hibernate.dialect.HSQLDialect`
Ingres	`net.sf.hibernate.dialect.IngresDialect`
Progress	`net.sf.hibernate.dialect.ProgressDialect`
Mckoi SQL	`net.sf.hibernate.dialect.MckoiDialect`
Interbase	`net.sf.hibernate.dialect.InterbaseDialect`
Pointbase	`net.sf.hibernate.dialect.PointbaseDialect`
FrontBase	`net.sf.hibernate.dialect.FrontbaseDialect`
Firebird	`net.sf.hibernate.dialect.FirebirdDialect`

3.5.2. Outer Join Fetching

If your database supports ANSI or Oracle style outer joins, *outer join fetching* might 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 graph of objects connected by many-to-one, one-to-many or one-to-one associations to be retrieved in a single SQL `SELECT`.

By default, the fetched graph when loading an objects ends at leaf objects, collections, objects with proxies, or where circularities occur.

For a *particular association*, fetching may be enabled or disabled (and the default behaviour overridden) by setting the `outer-join` attribute in the XML mapping.

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 all one-to-one and many-to-one associations, which are, also by default, set to `auto` outer join. However, one-to-many associations and collections are never fetched with an outer-join, unless explicitely declared for each particular association. This behavior can also be overriden at runtime with Hibernate queries.

3.5.3. 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 JVM-level setting only.*

3.5.4. Custom `CacheProvider`

You may integrate a JVM-level (or clustered) second-level cache system by implementing the interface `net.sf.hibernate.cache.CacheProvider`. You may select the custom implementation by setting `hibernate.cache.provider_class`.

3.5.5. Transaction strategy configuration

If you wish to use the Hibernate `Transaction` API, you must specify a factory class for `Transaction` instances by setting the property `hibernate.transaction.factory_class`. The `Transaction` API hides the underlying transaction mechanism and allows Hibernate code to run in managed and non-managed environments.

There are two standard (built-in) choices:

`net.sf.hibernate.transaction.JDBCTransactionFactory`
: delegates to database (JDBC) transactions (default)

`net.sf.hibernate.transaction.JTATransactionFactory`
: delegates to JTA (if an existing transaction is underway, the `Session` performs its work in that context, otherwise a new transaction is started)

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

If you wish to use JVM-level caching of mutable data in a JTA environment, you must specify a strategy for obtaining the JTA `TransactionManager`, as this is not standardized for J2EE containers:

Table 3.5. JTA TransactionManagers

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

3.5.6. JNDI-bound `SessionFactory`

A JNDI bound Hibernate `SessionFactory` can simplify the lookup of the factory and the creation of new `Sessions`.

If you wish to have the `SessionFactory` bound to a JNDI namespace, specify a name (eg. `java:comp/env/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, eg. 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.

If you do choose to use JNDI, an EJB or other utility class may obtain the `SessionFactory` using a JNDI lookup.

3.5.7. 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.

3.6. 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. Also don't forget to enable SQL logging as described above (`hibernate.show_sql`), it is your first step when looking for performance problems.

3.7. Implementing a `NamingStrategy`

The interface `net.sf.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 ex-

ample). 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();
```

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

3.8. XML Configuration File

An alternative approach 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, 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 2.0//EN"

 "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">

<hibernate-configuration>

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

        <!-- properties -->
        <property name="connection.datasource">my/first/datasource</property>
        <property name="dialect">net.sf.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">false</property>
        <property name="use_outer_join">true</property>
        <property name="transaction.factory_class">
            net.sf.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"/>

    </session-factory>

</hibernate-configuration>
```

Configuring 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();
```

Chapter 4. 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). Persistent classes have, as the name implies, transient and also persistent instance stored in the database.

Hibernate works best if these classes follow some simple rules, also known as the Plain Old Java Object (POJO) programming model.

4.1. 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 String name;
    private Date birthdate;
    private Cat mate;
    private Set kittens
    private Color color;
    private char sex;
    private float weight;

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

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

    void setMate(Cat mate) {
        this.mate = mate;
    }
    public Cat getMate() {
        return mate;
    }

    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 setKittens(Set kittens) {
        this.kittens = kittens;
    }
    public Set getKittens() {
        return kittens;
    }
    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
        kittens.add(kitten);
    }
    void setSex(char sex) {
        this.sex=sex;
    }
    public char getSex() {
        return sex;
    }
}
```

There are four main rules to follow here:

4.1.1. Declare accessors and mutators for persistent fields

`Cat` declares accessor methods for all its persistent fields. Many other ORM tools directly persist instance variables. We believe it is far better to decouple this implementation detail from the persistence mechanism. Hibernate persists JavaBeans style properties, and recognizes method names of the form `getFoo`, `isFoo` and `setFoo`.

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

4.1.2. Implement a default constructor

`Cat` has an implicit default (no-argument) constructor. All persistent classes must have a default constructor (which may be non-public) so Hibernate can instantiate them using `Constructor.newInstance()`.

4.1.3. Provide an identifier property (optional)

`Cat` has a property called `id`. This property holds 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 below.)

The identifier property is optional. You can leave it off and let Hibernate keep track of object identifiers internally. However, for many applications it is still a good (and very popular) design decision.

What's more, some functionality is available only to classes which declare an identifier property:

- Cascaded updates (see "Lifecycle Objects")
- `Session.saveOrUpdate()`

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

4.1.4. 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 - which will limit your options for performance tuning somewhat.

4.2. Implementing inheritance

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

```
package eg;

public class DomesticCat extends Cat {
        private String name;

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

4.3. Implementing `equals()` and `hashCode()`

You have to override the `equals()` and `hashCode()` methods if you intend to mix objects of persistent classes (e.g. in a `Set`).

This only applies if these objects are loaded in two different `Sessions`, as Hibernate only guarantees JVM identity (`a == b` , the default implementation of `equals()`) inside a single `Session`!

Even if both objecs `a` and `b` are the same database row (they have the same primary key value as their identifier), we can't guarantee that they are the same Java instance outside of a particular `Session` context.

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. Hibernate will only assign identifier values to objects that are persistent, a newly created instance will not have any identifier value! 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 (!getName().equals(cat.getName())) return false;
        if (!getBirthday().equals(cat.getBirthday())) return false;

```

```
        return true;
    }

    public int hashCode() {
        int result;
        result = getName().hashCode();
        result = 29 * result + getBirthday().hashCode();
        return result;
    }

}
```

Keep in mind that our candidate key (in this case a composite of name and birthday) has to be only valid for a particular comparison operation (maybe even only in a single use case). We don't need the stability criteria we usually apply to a real primary key!

4.4. Lifecycle Callbacks

Optionally, a persistent class might implement the interface `Lifecycle` which provides some callbacks that allow the persistent object to perform necessary initialization/cleanup after save or load and before deletion or update.

The Hibernate `Interceptor` offers a less intrusive alternative, however.

```
public interface Lifecycle {
        public boolean onSave(Session s) throws CallbackException;   (1)
        public boolean onUpdate(Session s) throws CallbackException; (2)
        public boolean onDelete(Session s) throws CallbackException; (3)
        public void onLoad(Session s, Serializable id);              (4)
}
```

(1) `onSave` - called just before the object is saved or inserted

(2) `onUpdate` - called just before an object is updated (when the object is passed to `Session.update()`)

(3) `onDelete` - called just before an object is deleted

(4) `onLoad` - called just after an object is loaded

`onSave()`, `onDelete()` and `onUpdate()` may be used to cascade saves and deletions of dependent objects. This is an alternative to declaring cascaded operations in the mapping file. `onLoad()` may be used to initialize transient properties of the object from its persistent state. It may not be used to load dependent objects since the `Session` interface may not be invoked from inside this method. A further intended usage of `onLoad()`, `onSave()` and `onUpdate()` is to store a reference to the current `Session` for later use.

Note that `onUpdate()` is not called every time the object's persistent state is updated. It is called only when a transient object is passed to `Session.update()`.

If `onSave()`, `onUpdate()` or `onDelete()` return `true`, the operation is silently vetoed. If a `CallbackException` is thrown, the operation is vetoed and the exception is passed back to the application.

Note that `onSave()` is called after an identifier is assigned to the object, except when native key generation is used.

4.5. Validatable callback

If the persistent class needs to check invariants before its state is persisted, it may implement the following interface:

```
public interface Validatable {
        public void validate() throws ValidationFailure;
}
```

The object should throw a `ValidationFailure` if an invariant was violated. An instance of `Validatable` should not change its state from inside `validate()`.

Unlike the callback methods of the `Lifecycle` interface, `validate()` might be called at unpredictable times. The application should not rely upon calls to `validate()` for business functionality.

4.6. Using XDOclet markup

In the next chapter we will show how Hibernate mappings may be expressed using a simple, readable XML format. 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 mate;
    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="MATE_ID"
     */
    public Cat getMate() {
        return mate;
    }
    void setMate(Cat mate) {
        this.mate = mate;
    }

    /**
     * @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
     *  lazy="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;
    }
}
```

Chapter 5. Basic O/R Mapping

5.1. Mapping declaration

Object/relational mappings are 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 define XML mappings be 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 2.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat" table="CATS" discriminator-value="C">
                <id name="id" column="uid" type="long">
                        <generator class="hilo"/>
                </id>
                <discriminator column="subclass" type="character"/>
                <property name="birthdate" type="date"/>
                <property name="color" not-null="true"/>
                <property name="sex" not-null="true" update="false"/>
                <property name="weight"/>
                <many-to-one name="mate" column="mate_id"/>
                <set name="kittens">
                        <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.)

5.1.1. 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/net/sf/hibernate` or in `hibernate.jar`. Hibernate will always look for the DTD in its classpath first.

5.1.2. hibernate-mapping

This element has three optional attributes. The `schema` attribute specifies that tables referred to by this mapping belong to the named schema. If specified, tablenames will be qualified by the given schema name. 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"                         (1)
        default-cascade="none|save-update"          (2)
        auto-import="true|false"                    (3)
        package="package.name"                      (4)
 />
```

(1) `schema` (optional): The name of a database schema.
(2) `default-cascade` (optional - defaults to `none`): A default cascade style.
(3) `auto-import` (optional - defaults to `true`): Specifies whether we can use unqualified class names (of classes in this mapping) in the query language.
(4) `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.

5.1.3. class

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

```
<class
        name="ClassName"                            (1)
        table="tableName"                           (2)
        discriminator-value="discriminator_value"   (3)
        mutable="true|false"                        (4)
        schema="owner"                              (5)
        proxy="ProxyInterface"                      (6)
        dynamic-update="true|false"                 (7)
        dynamic-insert="true|false"                 (8)
        select-before-update="true|false"           (9)
        polymorphism="implicit|explicit"            (10)
        where="arbitrary sql where condition"       (11)
        persister="PersisterClass"                  (12)
        batch-size="N"                              (13)
        optimistic-lock="none|version|dirty|all"    (14)
        lazy="true|false"                           (15)
/>
```

(1) `name`: The fully qualified Java class name of the persistent class (or interface).
(2) `table`: The name of its database table.
(3) `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`.
(4) `mutable` (optional, defaults to `true`): Specifies that instances of the class are (not) mutable.
(5) `schema` (optional): Override the schema name specified by the root `<hibernate-mapping>` element.
(6) `proxy` (optional): Specifies an interface to use for lazy initializing proxies. You may specify the name of the class itself.
(7) `dynamic-update` (optional, defaults to `false`): Specifies that `UPDATE` SQL should be generated at runtime and contain only those columns whose values have changed.
(8) `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.

(9) `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.

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

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

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

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

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

(15) `lazy` (optional): Setting `lazy="true"` is a shortcut equalivalent to specifying the name of the class itself as the `proxy` interface.

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 be 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 `net.sf.hibernate.persister.EntityPersister` or you might even provide a completely new implementation of the interface `net.sf.hibernate.persister.ClassPersister` that implements persistence via, for example, stored procedure calls, serialization to flat files or LDAP. See `net.sf.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 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
- `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 outside of the session (ie. when `Session.update()` is used). Keep in mind that a version or timestamp property should never be null, no matter what `unsaved-value` strategy, or an instance will be detected as transient.

5.1.4. 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"                         (1)
        type="typename"                             (2)
        column="column_name"                        (3)
        unsaved-value="any|none|null|id_value"      (4)
        access="field|property|ClassName">          (5)

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

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

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

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

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

(5) `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 important! If the identfier property of your class does not default to `null`, then you should specify the actual default.

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

5.1.4.1. generator

The required `<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="uid" unsaved-value="0">
        <generator class="net.sf.hibernate.id.TableHiLoGenerator">
                <param name="table">uid_table</param>
                <param name="column">next_hi_value_column</param>
        </generator>
</id>
```

All generators implement the interface `net.sf.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. *Do not use this generator with connections enlisted with JTA or with a user-supplied connection.*

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

`uuid.hex`
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.

`uuid.string`
uses the same UUID algorithm. The UUID is encoded a string of length 16 consisting of (any) ASCII characters. *Do not use with PostgreSQL.*

`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.

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

5.1.4.2. 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, or when Hibernate is using an application server datasource to obtain connections enlisted with JTA. Hibernate must be able to fetch the "hi" value in a new transaction. A standard approach in an EJB environment is to implement the hi/lo algorithm using a stateless session bean.

5.1.4.3. UUID Algorithm

The UUIDs contain: 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.

Don't try to use `uuid.string` in PostgreSQL.

5.1.4.4. 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="uid">
        <generator class="sequence">
                <param name="sequence">uid_sequence</param>
        </generator>
</id>
```

```
<id name="id" type="long" column="uid" 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.

5.1.4.5. 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. Be very careful when using this feature to assign keys with business meaning (almost always a terrible design decision).

Due to its inherent nature, entities that use this generator cannot be saved via the Session's saveOrUpdate() method. Instead you have to explicitly specify to Hibernate if the object should be saved or updated by calling either the `save()` or `update()` method of the Session.

5.1.5. composite-id

```
<composite-id
        name="propertyName"
        class="ClassName"
        unsaved-value="any|none"
        access="field|property|ClassName">
```

```
        <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 will describe a much more convenient approach where the composite identifier is implemented as a seperate class in Section 7.4, “Components as composite identifiers”. The attributes described below apply only to this alternative approach:

- `name` (optional): A property of component type that holds the composite identifier (see next section).
- `class` (optional - defaults to the property type determined by reflection): The component class used as a composite identifier (see next section).
- `unsaved-value` (optional - defaults to `none`): Indicates that transient instances should be considered newly instantiated, if set to `any`.

5.1.6. 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"  (1)
        type="discriminator_type"      (2)
        force="true|false"             (3)
        insert="true|false"            (4)
/>
```

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

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

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

(4) `insert` (optional - defaults to `true`) set this to `false` if your discriminator column is also part of a mapped composite identifier.

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.

5.1.7. 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"                         (1)
        name="propertyName"                             (2)
        type="typename"                                 (3)
        access="field|property|ClassName"               (4)
        unsaved-value="null|negative|undefined"         (5)
/>
```

(1) `column` (optional - defaults to the property name): The name of the column holding the version number.
(2) `name`: The name of a property of the persistent class.
(3) `type` (optional - defaults to `integer`): The type of the version number.
(4) `access` (optional - defaults to `property`): The strategy Hibernate should use for accessing the property value.
(5) `unsaved-value` (optional - defaults to `undefined`): A version property value that indicates that an instance is newly instantiated (unsaved), distinguishing it from transient instances that were saved or loaded in a previous session. (`undefined` specifies that the identifier property value should be used.)

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

5.1.8. 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"           (1)
        name="propertyName"                 (2)
        access="field|property|ClassName"   (3)
        unsaved-value="null|undefined"      (4)
/>
```

(1) `column` (optional - defaults to the property name): The name of a column holding the timestamp.
(2) `name`: The name of a JavaBeans style property of Java type `Date` or `Timestamp` of the persistent class.
(3) `access` (optional - defaults to `property`): The strategy Hibernate should use for accessing the property value.
(4) `unsaved-value` (optional - defaults to `null`): A version property value that indicates that an instance is newly instantiated (unsaved), distinguishing it from transient instances that were saved or loaded in a previous session. (`undefined` specifies that the identifier property value should be used.)

Note that `<timestamp>` is equivalent to `<version type="timestamp">`.

5.1.9. property

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

```
<property
        name="propertyName"                 (1)
        column="column_name"                (2)
```

```
        type="typename"                     (3)
        update="true|false"                 (4)
        insert="true|false"                 (4)
        formula="arbitrary SQL expression"  (5)
        access="field|property|ClassName"   (6)
/>
```

(1) `name`: the name of the property, with an initial lowercase letter.
(2) `column` (optional - defaults to the property name): the name of the mapped database table column.
(3) `type` (optional): a name that indicates the Hibernate type.
(4) `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.
(5) `formula` (optional): an SQL expression that defines the value for a *computed* property. Computed properties do not have a column mapping of their own.
(6) `access` (optional - defaults to `property`): The strategy Hibernate should use for accessing the property value.

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 subclass of `PersistentEnum` (eg. `eg.Color`).
4. The name of a serializable Java class.
5. 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 `net.sf.hibernate.property.PropertyAccessor`.

5.1.10. 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. (Its really just an object reference.)

```
<many-to-one
        name="propertyName"                             (1)
        column="column_name"                            (2)
        class="ClassName"                               (3)
        cascade="all|none|save-update|delete"           (4)
        outer-join="true|false|auto"                    (5)
        update="true|false"                             (6)
        insert="true|false"                             (6)
        property-ref="propertyNameFromAssociatedClass"  (7)
        access="field|property|ClassName"               (8)
                unique="true|false"                                 (9)
/>
```

(1) `name`: The name of the property.
(2) `column` (optional): The name of the column.
(3) `class` (optional - defaults to the property type determined by reflection): The name of the associated class.
(4) `cascade` (optional): Specifies which operations should be cascaded from the parent object to the associated object.
(5) `outer-join` (optional - defaults to `auto`): enables outer-join fetching for this association when `hibernate.use_outer_join` is set.
(6) `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.
(7) `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.
(8) `access` (optional - defaults to `property`): The strategy Hibernate should use for accessing the property value.
(9) `unique` (optional): Enable the DDL generation of a unique constraint for the foreign-key column.

The `cascade` attribute permits the following values: `all`, `save-update`, `delete`, `none`. Setting a value other than `none` will propagate certain operations to the associated (child) object. See "Lifecycle Objects" below.

The `outer-join` attribute accepts three different values:

- `auto` (default) Fetch the association using an outerjoin if the associated class has no proxy
- `true` Always fetch the association using an outerjoin
- `false` Never fetch the association using an outerjoin

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

```
<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.

5.1.11. 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"                                 (1)
        class="ClassName"                                   (2)
        cascade="all|none|save-update|delete"               (3)
        constrained="true|false"                            (4)
        outer-join="true|false|auto"                        (5)
        property-ref="propertyNameFromAssociatedClass"      (6)
        access="field|property|ClassName"                   (7)
```

```
/>
```

(1) `name`: The name of the property.
(2) `class` (optional - defaults to the property type determined by reflection): The name of the associated class.
(3) `cascade` (optional) specifies which operations should be cascaded from the parent object to the associated object.
(4) `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 is also used by the schema export tool).
(5) `outer-join` (optional - defaults to `auto`): Enable outer-join fetching for this association when `hibernate.use_outer_join` is set.
(6) `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.
(7) `access` (optional - defaults to `property`): The strategy Hibernate should use for accessing the property value.

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 primar 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"/>
```

5.1.12. 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"                 (1)
        class="className"                   (2)
        insert="true|false"                 (3)
        upate="true|false"                  (4)
        access="field|property|ClassName">  (5)

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

(1) `name`: The name of the property.

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

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

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

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

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.

5.1.13. subclass

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

```
<subclass
        name="ClassName"                            (1)
        discriminator-value="discriminator_value"   (2)
        proxy="ProxyInterface"                      (3)
        lazy="true|false"                           (4)
        dynamic-update="true|false"
        dynamic-insert="true|false">

        <property .... />
        .....
</subclass>
```

(1) `name`: The fully qualified class name of the subclass.

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

(3) `proxy` (optional): Specifies a class or interface to use for lazy initializing proxies.
(4) `lazy` (optional): Setting `lazy="true"` is a shortcut equalivalent to specifying the name of the class itself as the `proxy` interface.

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.

5.1.14. joined-subclass

Alternatively, a subclass that is persisted to its own table (table-per-subclass mapping strategy) is declared using a `<joined-subclass>` element.

```
<joined-subclass
        name="ClassName"                    (1)
        proxy="ProxyInterface"              (2)
        lazy="true|false"                   (3)
        dynamic-update="true|false"
        dynamic-insert="true|false">

        <key .... >

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

(1) `name`: The fully qualified class name of the subclass.
(2) `proxy` (optional): Specifies a class or interface to use for lazy initializing proxies.
(3) `lazy` (optional): Setting `lazy="true"` is a shortcut equalivalent to specifying the name of the class itself as the `proxy` interface.

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-2.0.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>
```

5.1.15. map, set, list, bag

Collections are discussed later.

5.1.16. 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"             (1)
        rename="ShortName"            (2)
/>
```

(1) `class`: The fully qualified class name of of any Java class.

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

5.2. Hibernate Types

5.2.1. 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, 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.

All Hibernate types except collections support null semantics.

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.

5.2.2. Basic value types

The *basic 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`
: A type mapping from `java.math.BigDecimal` 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 or implement `PersistentEnum`.

`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.)

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 `net.sf.hibernate.Hibernate`. For example, `Hibernate.STRING` represents the `string` type.

5.2.3. Persistent enum types

An *enumerated* type is a common Java idiom where a class has a constant (small) number of immutable instances. You may create a persistent enumerated type by implementing `net.sf.hibernate.PersistentEnum`, defining the operations `toInt()` and `fromInt()`:

```
package eg;
import net.sf.hibernate.PersistentEnum;

public class Color implements PersistentEnum {
    private final int code;
    private Color(int code) {
        this.code = code;
    }
    public static final Color TABBY = new Color(0);
    public static final Color GINGER = new Color(1);
    public static final Color BLACK = new Color(2);

    public int toInt() { return code; }

    public static Color fromInt(int code) {
        switch (code) {
            case 0: return TABBY;
            case 1: return GINGER;
            case 2: return BLACK;
            default: throw new RuntimeException("Unknown color code");
        }
    }
}
```

The Hibernate type name is simply the name of the enumerated class, in this case `eg.Color`.

5.2.4. 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 `net.sf.hibernate.UserType` or `net.sf.hibernate.CompositeUserType` and declare properties using the fully qualified classname of the type. Check out `net.sf.hibernate.test.DoubleStringType` to see the kind of things that are possible.

```
<property name="twoStrings" type="net.sf.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.

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 `MonetoryAmount` 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 monetory values.

5.2.5. Any type mappings

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).

```
<any name="anyEntity" id-type="long" meta-type="eg.custom.Class2TablenameType">
    <column name="table_name"/>
    <column name="id"/>
</any>
```

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`. If the meta-type returns instances of `java.lang.Class`, nothing else is required. On the other hand, if it is a basic type like `string` or `character`, you must specify the mapping from values to classes.

```
<any name="anyEntity" 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"                     (1)
        id-type="idtypename"                    (2)
        meta-type="metatypename"                (3)
        cascade="none|all|save-update"          (4)
        access="field|property|ClassName"       (5)
>
        <meta-value ... />
        <meta-value ... />
        .....
        <column .... />
        <column .... />
        .....
</any>
```

(1) `name`: the property name.

(2) `id-type`: the identifier type.

(3) `meta-type` (optional - defaults to `class`): a type that maps `java.lang.Class` to a single database column or, alternatively, a type that is allowed for a discriminator mapping.

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

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

The old `object` type that filled a similar role in Hibernate 1.2 is still supported, but is now semi-deprecated.

5.3. 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>
```

5.4. Modular mapping files

It is possible to define `subclass` and `joined-subclass` mappings in seperate 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. Use of this feature makes the ordering of the mapping documents important!

```

<hibernate-mapping>
        <subclass name="eg.subclass.DomesticCat" extends="eg.Cat" discriminator-value="D">
             <property name="name" type="string"/>
        </subclass>
</hibernate-mapping>
```

Chapter 6. Collection Mapping

6.1. Persistent Collections

This section does not contain much example Java code. We assume you already know how to use Java's collections framework. If so, there's not really anything more to know - with a single caveat, you may use Java collections the same way you always have.

Hibernate can persist instances of `java.util.Map`, `java.util.Set`, `java.util.SortedMap`, `java.util.SortedSet`, `java.util.List`, and any array of persistent entities or values. Properties of type `java.util.Collection` or `java.util.List` may also be persisted with "bag" semantics.

Now the caveat: persistent collections do not retain any extra semantics added by the class implementing the collection interface (eg. iteration order of a `LinkedHashSet`). The persistent collections actually behave like `HashMap`, `HashSet`, `TreeMap`, `TreeSet` and `ArrayList` respectively. Furthermore, the Java type of a property holding a collection must be the interface type (ie. `Map`, `Set` or `List`; never `HashMap`, `TreeSet` or `ArrayList`). This restriction exists because, when you're not looking, Hibernate sneakily replaces your instances of `Map`, `Set` and `List` with instances of its own persistent implementations of `Map`, `Set` or `List`. (So also be careful when using == on your collections.)

```
Cat cat = new DomesticCat();
Cat kitten = new DomesticCat();
....
Set kittens = new HashSet();
kittens.add(kitten);
cat.setKittens(kittens);
session.save(cat);
kittens = cat.getKittens(); //Okay, kittens collection is a Set
(HashSet) cat.getKittens(); //Error!
```

Collections obey the usual rules for value types: no shared references, created and deleted along with containing entity. Due to the underlying relational model, they do not support null value semantics; Hibernate does not distinguish between a null collection reference and an empty collection.

Collections 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. You shouldn't have to worry much about any of this. Just use Hibernate's collections the same way you use ordinary Java collections, but make sure you understand the semantics of bidirectional associations (discussed later) before using them.

Collection instances are distinguished in the database by a foreign key to the owning entity. This foreign key is referred to as the *collection key* . The collection key is mapped by the `<key>` element.

Collections may contain almost any other Hibernate type, including all basic types, custom types, entity types and components. This is an important definition: An object in a collection can either be handled with "pass by value" semantics (it therefore fully depends on the collection owner) or it can be a reference to another entity with an own lifecycle. Collections may not contain other collections. The contained type is referred to as the *collection element type*. Collection elements are mapped by `<element>`, `<composite-element>`, `<one-to-many>`, `<many-to-many>` or `<many-to-any>`. The first two map elements with value semantics, the other three are used to map entity associations.

All collection types except `Set` and bag have an *index* column - a column that maps to an array or `List` index or `Map` key. The index of a `Map` may be of any basic type, an entity type or even a composite type (it may not be a

collection). The index of an array or list is always of type `integer`. Indexes are mapped using `<index>`, `<index-many-to-many>`, `<composite-index>` or `<index-many-to-any>`.

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.

6.2. Mapping a Collection

Collections are declared by the `<set>`, `<list>`, `<map>`, `<bag>`, `<array>` and `<primitive-array>` elements. `<map>` is representative:

```
<map
    name="propertyName"                                          (1)
    table="table_name"                                           (2)
    schema="schema_name"                                         (3)
    lazy="true|false"                                            (4)
    inverse="true|false"                                         (5)
    cascade="all|none|save-update|delete|all-delete-orphan"      (6)
    sort="unsorted|natural|comparatorClass"                      (7)
    order-by="column_name asc|desc"                              (8)
    where="arbitrary sql where condition"                        (9)
    outer-join="true|false|auto"                                 (10)
    batch-size="N"                                               (11)
    access="field|property|ClassName"                            (12)
>

    <key .... />
    <index .... />
    <element .... />
</map>
```

(1) `name` the collection property name

(2) `table` (optional - defaults to property name) the name of the collection table (not used for one-to-many associations)

(3) `schema` (optional) the name of a table schema to override the schema declared on the root element

(4) `lazy` (optional - defaults to `false`) enable lazy initialization (not used for arrays)

(5) `inverse` (optional - defaults to `false`) mark this collection as the "inverse" end of a bidirectional association

(6) `cascade` (optional - defaults to `none`) enable operations to cascade to child entities

(7) `sort` (optional) specify a sorted collection with `natural` sort order, or a given comparator class

(8) `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`

(9) `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)

(10) `outer-join` (optional) specify that the collection should be fetched by outer join, whenever possible. Only one collection may be fetched by outer join per SQL `SELECT`.

(11) `batch-size` (optional, defaults to `1`) specify a "batch size" for lazily fetching instances of this collection.

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

The mapping of a `List` or array requires a seperate table column holding the array or list index (the `i` in `foo[i]`). If your relational model doesn't have an index column, e.g. if you're working with legacy data, use an unordered `Set` instead. This seems to put people off who assume that `List` should just be a more convenient way of accessing an unordered collection. Hibernate collections strictly obey the actual semantics attached to the `Set`, `List` and `Map` interfaces. `List` elements don't just spontaneously rearrange themselves!

On the other hand, people who planned to use the `List` to emulate *bag* semantics have a legitimate grievance here. A bag is an unordered, unindexed collection which may contain the same element multiple times. The Java collections framework lacks a `Bag` interface, hence you have to emulate it with a `List`. Hibernate lets you map properties of type `List` or `Collection` with the `<bag>` element. Note that bag semantics are not really part of the `Collection` contract and they actually conflict with the semantics of the `List` contract (however, you can sort the bag arbitrarily, discussed later in this chapter).

Note: Large Hibernate bags mapped with `inverse="false"` are inefficient and should be avoided; Hibernate can't create, delete or update rows individually, because there is no key that may be used to identify an individual row.

6.3. Collections of Values and Many-To-Many Associations

A collection table is required for any collection of values and any collection of references to other entities mapped as a many-to-many association (the natural semantics for a Java collection). The table requires (foreign) key column(s), element column(s) and possibly index column(s).

The foreign key from the collection table to the table of the owning class is declared using a `<key>` element.

```
<key column="column_name"/>
```

(1) `column` (required): The name of the foreign key column.

For indexed collections like maps and lists, we require an `<index>` element. For lists, this column contains sequential integers numbered from zero. Make sure that your index really starts from zero if you have to deal with legacy data. For maps, the column may contain any values of any Hibernate type.

```
<index
        column="column_name"                (1)
        type="typename"                     (2)
/>
```

(1) `column` (required): The name of the column holding the collection index values.
(2) `type` (optional, defaults to `integer`): The type of the collection index.

Alternatively, a map may be indexed by objects of entity type. We use the `<index-many-to-many>` element.

```
<index-many-to-many
        column="column_name"                (1)
        class="ClassName"                   (2)
/>
```

(1) `column` (required): The name of the foreign key column for the collection index values.
(2) `class` (required): The entity class used as the collection index.

For a collection of values, we use the `<element>` tag.

```
<element
        column="column_name"                (1)
        type="typename"                     (2)
/>
```

(1) `column` (required): The name of the column holding the collection element values.
(2) `type` (required): The type of the collection element.

A collection of entities with its own table corresponds to the relational notion of *many-to-many association*. A many to many association is the most natural mapping of a Java collection but is not usually the best relational model.

```
<many-to-many
        column="column_name"                                (1)
        class="ClassName"                                   (2)
        outer-join="true|false|auto"                        (3)
    />
```

(1) `column` (required): The name of the element foreign key column.
(2) `class` (required): The name of the associated class.
(3) `outer-join` (optional - defaults to `auto`): enables outer-join fetching for this association when `hibernate.use_outer_join` is set.

Some examples, first, a set of strings:

```
<set name="names" table="NAMES">
    <key column="GROUPID"/>
    <element column="NAME" type="string"/>
</set>
```

A bag containing integers (with an iteration order determined by the `order-by` attribute):

```
<bag name="sizes" table="SIZES" order-by="SIZE ASC">
    <key column="OWNER"/>
    <element column="SIZE" type="integer"/>
</bag>
```

An array of entities - in this case, a many to many association (note that the entities are lifecycle objects, `cascade="all"`):

```
<array name="foos" table="BAR_FOOS" cascade="all">
    <key column="BAR_ID"/>
    <index column="I"/>
    <many-to-many column="FOO_ID" class="org.hibernate.Foo"/>
</array>
```

A map from string indices to dates:

```
<map name="holidays" table="holidays" schema="dbo" order-by="hol_name asc">
    <key column="id"/>
    <index 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="car_components">
    <key column="car_id"/>
    <index column="posn"/>
    <composite-element class="org.hibernate.car.CarComponent">
            <property name="price" type="float"/>
            <property name="type" type="org.hibernate.car.ComponentType"/>
            <property name="serialNumber" column="serial_no" type="string"/>
    </composite-element>
</list>
```

6.4. One-To-Many Associations

A *one to many association* links the tables of two classes *directly*, with no intervening collection table. (This implements a *one-to-many* relational model.) This relational model loses some of the semantics of Java collections:

- No null values may be contained in a map, set or list
- 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 `Foo` to `Bar` requires the addition of a key column and possibly an index column to the table of the contained entity class, `Bar`. These columns are mapped using the `<key>` and `<index>` elements described above.

The `<one-to-many>` tag indicates a one to many association.

```
<one-to-many class="ClassName"/>
```

(1) `class` (required): The name of the associated class.

Example:

```
<set name="bars">
    <key column="foo_id"/>
    <one-to-many class="org.hibernate.Bar"/>
</set>
```

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 `<key>` column of a `<one-to-many>` association is declared `NOT NULL`, Hibernate may cause constraint violations when it creates or updates the association. To prevent this problem, *you must use a bidirectional association* with the many valued end (the set or bag) marked as `inverse="true"`. See the discussion of bidirectional associations later in this chapter.

6.5. Lazy Initialization

Collections (other than arrays) may be lazily initialized, meaning they load their state from the database only when the application needs to access it. Initialization happens transparently to the user so the application would not normally need to worry about this (in fact, transparent lazy initialization is the main reason why Hibernate needs its own collection implementations). However, if the application tries something like this:

```
s = sessions.openSession();
User u = (User) s.find("from User u where u.name=?", userName, Hibernate.STRING).get(0);
Map permissions = u.getPermissions();
s.connection().commit();
s.close();

Integer accessLevel = (Integer) permissions.get("accounts");  // Error!
```

It could be in for a nasty surprise. Since the permissions collection was not initialized when the `Session` was committed, the collection will never be able to load its state. The fix is to move the line that reads from the collection to just before the commit. (There are other more advanced ways to solve this problem, however.)

Alternatively, use a non-lazy collection. Since lazy initialization can lead to bugs like that above, non-laziness is the default. However, it is intended that lazy initialization be used for almost all collections, especially for collections of entities (for reasons of efficiency).

Exceptions that occur while lazily initializing a collection are wrapped in a `LazyInitializationException`.

Declare a lazy collection using the optional `lazy` attribute:

```
<set name="names" table="NAMES" lazy="true">
    <key column="group_id"/>
    <element column="NAME" type="string"/>
</set>
```

In some application architectures, particularly where the code that accesses data using Hibernate, and the code that uses it are in different application layers, it can be a problem to ensure that the `Session` is open when a collection is initialized. They 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. Of course, this places heavy demands upon 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. The servlet filter has to be able to access the `Session` for this approach. We recommend that a `ThreadLocal` variable be used to hold the current `Session` (see chapter 1, Section 1.4, “Playing with cats”, for an example implementation).

- In an application with a seperate 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.

- You may also attach a previously loaded object to a new `Session` with `update()` or `lock()` before accessing unitialized collections (or other proxies). Hibernate can not do this automatically, as it would introduce ad hoc transaction semantics!

You can use the `filter()` method of the Hibernate Session API to get the size of a collection without initializing it:

```
( (Integer) s.filter( collection, "select count(*)" ).get(0) ).intValue()
```

`filter()` or `createFilter()` are also used to efficiently retrieve subsets of a collection without needing to initialize the whole collection.

6.6. 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" lazy="true">
    <key column="year_id"/>
    <index 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="name asc">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" order-by="hol_date, hol_name" lazy="true">
    <key column="year_id"/>
    <index 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 `filter()`.

```
sortedUsers = s.filter( group.getUsers(), "order by this.name" );
```

6.7. 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 (slightly experimental) 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" lazy="true">
    <collection-id column="ID" type="long">
        <generator class="hilo"/>
    </collection-id>
    <key column="PERSON1"/>
    <many-to-many column="PERSON2" class="eg.Person" outer-join="true"/>
</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 `identity` identifier generation strategy is not supported for `<idbag>` collection identifiers.

6.8. 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

Please note that Hibernate does not support bidirectional one-to-many associations with an indexed collection (list, map or array) as the "many" end, you have to use a set or bag mapping.

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). Here's an example of a bidirectional many-to-many association from a class back to *itself* (each category can have many items and each item can be in many categories):

```
<class name="org.hibernate.auction.Category">
    <id name="id" column="ID"/>
    ...
    <bag name="items" table="CATEGORY_ITEM" lazy="true">
        <key column="CATEGORY_ID"/>
        <many-to-many class="org.hibernate.auction.Item" column="ITEM_ID"/>
    </bag>
</class>

<class name="org.hibernate.auction.Item">
    <id name="id" column="ID"/>
    ...

    <!-- inverse end -->
    <bag name="categories" table="CATEGORY_ITEM" inverse="true" lazy="true">
        <key column="ITEM_ID"/>
        <many-to-many class="org.hibernate.auction.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.update(item);                     // No effect, nothing will be saved!
session.update(category);                 // The relationship will be saved
```

The non-inverse side is used to save the in-memory representation to the database. We would get an unnecessary INSERT/UPDATE and probably even a foreign key violation if both would trigger changes! The same is of course also true for bidirectional one-to-many associations.

You may map 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="eg.Parent">
```

```
    <id name="id" column="id"/>
    ....
    <set name="children" inverse="true" lazy="true">
        <key column="parent_id"/>
        <one-to-many class="eg.Child"/>
    </set>
</class>

<class name="eg.Child">
    <id name="id" column="id"/>
    ....
    <many-to-one name="parent" class="eg.Parent" column="parent_id"/>
</class>
```

Mapping one end of an association with `inverse="true"` doesn't affect the operation of cascades, both are different concepts!

6.9. Ternary Associations

There are two possible approaches to mapping a ternary association. One approach is to use composite elements (discussed below). Another is to use a `Map` with an association as its index:

```
<map name="contracts" lazy="true">
    <key column="employer_id"/>
    <index-many-to-many column="employee_id" class="Employee"/>
    <one-to-many column="contract_id" class="Contract"/>
</map>
```

```
<map name="connections" lazy="true">
    <key column="node1_id"/>
    <index-many-to-many column="node2_id" class="Node"/>
    <many-to-many column="connection_id" class="Connection"/>
</map>
```

6.10. Heterogeneous Associations

The `<many-to-any>` and `<index-many-to-any>` elements provide for true heterogeneous associations. These mapping elements work in the same way as the `<any>` element - and should also be used rarely, if ever.

6.11. 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 `eg.Child` instances. If each child has at most one parent, the most natural mapping is a one-to-many association:

```
<hibernate-mapping>

    <class name="eg.Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" lazy="true">
            <key column="parent_id"/>
            <one-to-many class="eg.Child"/>
        </set>
    </class>

    <class name="eg.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="eg.Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" inverse="true" lazy="true">
            <key column="parent_id"/>
            <one-to-many class="eg.Child"/>
        </set>
    </class>

    <class name="eg.Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
        <many-to-one name="parent" class="eg.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
```

On the other hand, if a child might have multiple parents, a many-to-many association is appropriate:

```
<hibernate-mapping>

    <class name="eg.Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" lazy="true" table="childset">
            <key column="parent_id"/>
            <many-to-many class="eg.Child" column="child_id"/>
        </set>
    </class>

    <class name="eg.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
```

Chapter 7. Component Mapping

The notion of a *component* is re-used in several different contexts, for different purposes, throughout Hibernate.

7.1. Dependent objects

A component is a contained object that is persisted as a value type, not an entity. 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.hex"/>
    </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. 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.hex"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name">
        <parent name="namedPerson"/> <!-- reference back to the Person -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class>
```

7.2. 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 compon-

ents 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>
```

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.

7.3. Components as Map indices

The `<composite-index>` 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.

7.4. 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.

You can't use an `IdentifierGenerator` to generate composite keys. Instead the application must assign its own

identifiers.

Since a composite identifier must be assigned to the object before saving it, we can't use `unsaved-value` of the identifier to distinguish between newly instantiated instances and instances saved in a previous session.

You may instead implement `Interceptor.isUnsaved()` if you wish to use `saveOrUpdate()` or cascading save / update. As an alternative, you may also set the `unsaved-value` attribute on a `<version>` (or `<timestamp>`) element to specify a value that indicates a new transient instance. In this case, the version of the entity is used instead of the (assigned) identifier and you don't have to implement `Interceptor.isUnsaved()` yourself.

Use the `<composite-id>` tag (same attributes and elements as `<component>`) in place of `<id>` for the declaration of a composite identifier class:

```
<class name="eg.Foo" table"FOOS">
    <composite-id name="compId" class="eg.FooCompositeID">
        <key-property name="string"/>
        <key-property name="short"/>
        <key-property name="date" column="date_" type="date"/>
    </composite-id>
    <property name="name"/>
    ....
</class>
```

Now, any foreign keys into the table `FOOS` are also composite. You must declare this in your mappings for other classes. An association to `Foo` would be declared like this:

```
<many-to-one name="foo" class="eg.Foo">
<!-- the "class" attribute is optional, as usual -->
    <column name="foo_string"/>
    <column name="foo_short"/>
    <column name="foo_date"/>
</many-to-one>
```

This new `<column>` tag is also used by multi-column custom types. Actually it is an alternative to the `column` attribute everywhere. A collection with elements of type `Foo` would use:

```
<set name="foos">
    <key column="owner_id"/>
    <many-to-many class="eg.Foo">
        <column name="foo_string"/>
        <column name="foo_short"/>
        <column name="foo_date"/>
    </many-to-many>
</set>
```

On the other hand, `<one-to-many>`, as usual, declares no columns.

If `Foo` itself contains collections, they will also need a composite foreign key.

```
<class name="eg.Foo">
    ....
    ....
    <set name="dates" lazy="true">
        <key>   <!-- a collection inherits the composite key type -->
            <column name="foo_string"/>
            <column name="foo_short"/>
            <column name="foo_date"/>
        </key>
        <element column="foo_date" type="date"/>
    </set>
</class>
```

7.5. Dynamic components

You may even map a property of type `Map`:

```
<dynamic-component name="userAttributes">
    <property name="foo" column="FOO"/>
    <property name="bar" column="BAR"/>
    <many-to-one name="baz" class="eg.Baz" column="BAZ"/>
</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.)

Chapter 8. Inheritance Mapping

8.1. The Three Strategies

Hibernate supports the three basic inheritance mapping strategies.

- table per class hierarchy
- table per subclass
- table per concrete class (some limitations)

It is even possible to use different mapping strategies for different branches of the same inheritance hierarchy, but the same limitations apply as apply to table-per-concrete class mappings. Hibernate does not support mixing `<subclass>` mappings and `<joined-subclass>` mappings inside the same `<class>` element.

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">
        ...
    </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 may not have `NOT NULL` constraints.

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"/>
        ...
    </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).

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.

For either of these two mapping strategies, a polymorphic association to `Payment` is mapped using `<many-to-one>`.

```
<many-to-one name="payment"
    column="PAYMENT"
    class="Payment"/>
```

The table-per-concrete-class strategy is very different.

```
<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>
```

Three tables were required. Notice that nowhere do we mention the `Payment` interface explicitly. Instead, we make use of Hibernate's *implicit polymorphism*. Also notice that properties of `Payment` are mapped in each of the subclasses.

In this case, a polymorphic association to `Payment` is mapped using `<any>`.

```
<any name="payment"
        meta-type="class"
        id-type="long">
    <column name="PAYMENT_CLASS"/>
    <column name="PAYMENT_ID"/>
</any>
```

It would be better if we defined a `UserType` as the `meta-type`, to handle the mapping from type discriminator strings to `Payment` subclass.

```
<any name="payment"
        meta-type="PaymentMetaType"
        id-type="long">
```

```
    <column name="PAYMENT_TYPE"/> <!-- CREDIT, CASH or CHEQUE -->
    <column name="PAYMENT_ID"/>
</any>
```

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 table-per-class or table-per-subclass 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`.

8.2. Limitations

Hibernate assumes that an association maps to exactly one foreign key column. Multiple associations per foreign key are tolerated (you might need to specify `inverse="true"` or `insert="false" update="false"`), but there is no way to map any association to multiple foreign keys. This means that:

- when an association is modified, it is always the same foreign key that is updated
- when an association is fetched lazily, a single database query is used
- when an association is fetched eagerly, it may be fetched using a single outer join

In particular, it implies that polymorphic one-to-many associations to classes mapped using the table-per-concrete-class strategy are *not supported*. (Fetching this association would require multiple queries or multiple joins.)

The following table shows the limitations of table-per-concrete-class mappings, and of implicit polymorphism,

in Hibernate.

Table 8.1. Features of inheritance mappings

Inheritance strategy	Polymorphic many-to-one	Polymorphic one-to-one	Polymorphic one-to-many	Polymorphic many-to-many	Polymorphic `load()/get()`	Polymorphic queries	Polymorphic joins
table-per-class-hierarchy	`<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-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 (implicit polymorphism)	`<any>`	*not supported*	*not supported*	`<many-to-any>`	*use a query*	`from Payment p`	*not supported*

Chapter 9. Manipulating Persistent Data

9.1. Creating a persistent object

An object (entity instance) is either *transient* or *persistent* with respect to a particular `Session`. Newly instantiated objects are, of course, transient. The session offers services for saving (ie. persisting) transient instances:

```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
Long generatedId = (Long) sess.save(fritz);
```

```
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) );
```

The single-argument `save()` generates and assigns a unique identifier to `fritz`. The two-argument form attempts to persist `pk` using the given identifier. We generally discourage the use of the two-argument form since it may be used to create primary keys with business meaning. It is most useful in certain special situations like using Hibernate to persist a BMP entity bean.

Associated 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.

9.2. Loading an object

The `load()` methods of `Session` give you a way to retrieve a persistent instance if you already know its identifier. One version takes a class object and will load the state into a newly instantiated object. The second version allows you to supply an instance into which the state will be loaded. The form which takes an instance is particularly useful if you plan to use Hibernate with BMP entity beans and is provided for exactly that purpose. You may discover other uses. (DIY instance pooling etc.)

```
Cat fritz = (Cat) sess.load(Cat.class, generatedId);
```

```
// you need to wrap primitive identifiers
long pkId = 1234;
DomesticCat pk = (DomesticCat) sess.load( Cat.class, new Long(pkId) );
```

```
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()` returns an object that is an uninitialized proxy and does not actually hit the database until you invoke a method of the object. This behaviour is very useful if you wish to create an association to an object without actually loading it from the database.

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 also load an objects using an SQL `SELECT ... FOR UPDATE`. See the next section for a discussion of Hibernate `LockModes`.

```
Cat cat = (Cat) sess.get(Cat.class, id, LockMode.UPGRADE);
```

Note that any associated instances or contained collections are *not* selected `FOR UPDATE`.

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)
```

9.3. Querying

If you don't know the identifier(s) of the object(s) you are looking for, use the `find()` methods of `Session`. Hibernate supports a simple but powerful object oriented query language.

```
List cats = sess.find(
    "from Cat as cat where cat.birthdate = ?",
    date,
    Hibernate.DATE
);

List mates = sess.find(
    "select mate from Cat as cat join cat.mate as mate " +
    "where cat.name = ?",
    name,
    Hibernate.STRING
);

List cats = sess.find( "from Cat as cat where cat.mate.bithdate is null" );

List moreCats = sess.find(
    "from Cat as cat where " +
    "cat.name = 'Fritz' or cat.id = ? or cat.id = ?",
    new Object[] { id1, id2 },
    new Type[] { Hibernate.LONG, Hibernate.LONG }
);

List mates = sess.find(
    "from Cat as cat where cat.mate = ?",
    izi,
    Hibernate.entity(Cat.class)
);

List problems = sess.find(
    "from GoldFish as fish " +
    "where fish.birthday > fish.deceased or fish.birthday is null"
);
```

The second argument to `find()` accepts an object or array of objects. The third argument accepts a Hibernate type or array of Hibernate types. These given types are used to bind the given objects to the `?` query placeholders (which map to IN parameters of a JDBC `PreparedStatement`). Just as in JDBC, you should use this binding mechanism in preference to string manipulation.

The `Hibernate` class defines a number of static methods and constants, providing access to most of the built-in types, as instances of `net.sf.hibernate.type.Type`.

If you expect your query to return a very large number of objects, but you don't expect to use them all, you might get better performance from the `iterate()` methods, which return a `java.util.Iterator`. The iterator will load objects on demand, using the identifiers returned by an initial SQL query (n+1 selects total).

```
// fetch ids
Iterator iter = sess.iterate("from eg.Qux q order by q.likeliness");
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;
    }
}
```

Unfortunately `java.util.Iterator` does not declare any exceptions, so any SQL or Hibernate exceptions that occur are wrapped in a `LazyInitializationException` (a subclass of `RuntimeException`).

The `iterate()` method also performs better if you expect that many of the objects are already loaded and cached by the session, or if the query results contain the same objects many times. (When no data is cached or repeated, `find()` is almost always faster.) Heres an example of a query that should be called using `iterate()`:

```
Iterator iter = sess.iterate(
    "select customer, product " +
    "from Customer customer, " +
    "Product product " +
    "join customer.purchases purchase " +
    "where product = purchase.product"
);
```

Calling the previous query using `find()` would return a very large JDBC `ResultSet` containing the same data many times.

Hibernate queries sometimes return tuples of objects, in which case each tuple is returned as an array:

```
Iterator foosAndBars = sess.iterate(
    "select foo, bar from Foo foo, Bar bar " +
    "where bar.date = foo.date"
);
while ( foosAndBars.hasNext() ) {
    Object[] tuple = (Object[]) foosAndBars.next();
    Foo foo = tuple[0]; Bar bar = tuple[1];
    ....
}
```

9.3.1. Scalar queries

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.

```
Iterator results = sess.iterate(
        "select cat.color, min(cat.birthdate), count(cat) from Cat cat " +
        "group by cat.color"
);
while ( results.hasNext() ) {
    Object[] row = results.next();
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    .....
}
```

```
Iterator iter = sess.iterate(
    "select cat.type, cat.birthdate, cat.name from DomesticCat cat"
);
```

```
List list = sess.find(
    "select cat, cat.mate.name from DomesticCat cat"
);
```

9.3.2. The Query interface

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 obtain an instance of `net.sf.hibernate.Query`:

```
Query q = sess.createQuery("from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.list();
```

You may even define a named query in the mapping document. (Remember to use a `CDATA` section if your query contains characters that could be interpreted as markup.)

```
<query name="eg.DomesticCat.by.name.and.minimum.weight"><![CDATA[
    from eg.DomesticCat as cat
        where cat.name = ?
        and cat.weight > ?
] ]></query>
```

```
Query q = sess.getNamedQuery("eg.DomesticCat.by.name.and.minimum.weight");
q.setString(0, name);
q.setInt(1, minWeight);
List cats = q.list();
```

The query interface supports the use of named parameters. Named parameters are identifiers of the form `:name` in the query string. There are methods on `Query` for binding values to named parameters or JDBC-style `?` parameters. *Contrary to JDBC, Hibernate numbers parameters from zero.* 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();
```

9.3.3. Scrollable iteration

If your JDBC driver supports scrollable `ResultSets`, the `Query` interface may be used to obtain a `ScrollableResults` which allows more 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) );

}
```

9.3.4. 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.filter(
    pk.getKittens(), "where this.color = ?", Color.BLACK, Hibernate.enum(Color.class)
);
```

The returned collection is considered a bag.

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.filter(
    pk.getKittens(), "select this.mate where this.color = eg.Color.BLACK"
);
```

9.3.5. Criteria queries

HQL is extremely powerful but some people prefer to build queries dynamically, using an object oriented API, rather than embedding strings in their Java code. For these people, Hibernate provides an intuitive `Criteria` query API.

```
Criteria crit = session.createCriteria(Cat.class);
crit.add( Expression.eq("color", eg.Color.BLACK) );
crit.setMaxResults(10);
List cats = crit.list();
```

If you are uncomfortable with SQL-like syntax, this is perhaps the easiest way to get started with Hibernate. This API is also more extensible than HQL. Applications might provide their own implementations of the `Criterion` interface.

9.3.6. Queries in native SQL

You may express a query in SQL, using `createSQLQuery()`. 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.

9.4. Updating objects

9.4.1. Updating in the same Session

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). 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.

9.4.2. Updating 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 transaction isolation.) This approach requires a slightly different programming model to the one described in the last section. Hibernate supports this model by providing the method `Session.update()`.

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catId);
Cat potentialMate = new Cat();
firstSession.save(potentialMate);

// in a higher tier 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 update it, an exception would have been thrown.

The application should individually `update()` transient instances reachable from the given transient instance if and *only* if it wants their state also updated. (Except for lifecycle objects, discussed later.)

Hibernate users have requested a general purpose method that either saves a transient instance by generating a new identifier or update the persistent state associated with its current identifier. The `saveOrUpdate()` method now implements this functionality.

Hibernate distinguishes "new" (unsaved) instances from "existing" (saved or loaded in a previous session) instances by the value of their identifier (or version, or timestamp) property. The `unsaved-value` attribute of the `<id>` (or `<version>`, or `<timestamp>`) mapping specifies which values should be interpreted as representing a "new" instance.

```
<id name="id" type="long" column="uid" unsaved-value="null">
    <generator class="hilo"/>
</id>
```

The allowed values of `unsaved-value` are:

- `any` - always save
- `none` - always update
- `null` - save when identifier is null (this is the default)
- valid identifier value - save when identifier is null or the given value
- `undefined` - the default for `version` or `timestamp`, then identifier check is used

```
// 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()` or `saveOrUpdate()`. 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 the object has no identifier property, `save()` it
- if the object's identifier matches the criteria specified by `unsaved-value`, `save()` it
- if the object is versioned (`version` or `timestamp`), then the version will take precedence to identifier check, unless the versions `unsaved-value="undefined"` (default value)
- if another object associated with the session has the same identifier, throw an exception

9.4.3. Reattaching detached objects

The `lock()` method allows the application to reassociate an unmodified object with a new session.

```
//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);
```

9.5. Deleting persistent objects

`Session.delete()` will remove an object's state from the database. Of course, your application might still hold a reference to it. So it's best to think of `delete()` as making a persistent instance transient.

```
sess.delete(cat);
```

You may also delete many objects at once by passing a Hibernate query string to `delete()`.

You may now delete objects in any order you like, without risk of foreign key constraint violations. Of course, it is still possible to violate a `NOT NULL` constraint on a foreign key column by deleting objects in the wrong order.

9.6. Flush

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

- from some invocations of `find()` or `iterate()`
- from `net.sf.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 `Session.find(..)` methods 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. This is most useful in the case of "readonly" transactions, where it might be used to achieve a (very) slight performance increase.

```
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);
// execute some queries....
sess.find("from Cat as cat left outer join cat.kittens kitten");
//change to izi is not flushed!
...
tx.commit(); //flush occurs
```

9.7. Ending a Session

Ending a session involves four distinct phases:

- flush the session
- commit the transaction
- close the session
- handle exceptions

9.7.1. Flushing the Session

If you happen to be using the `Transaction` API, you don't need to worry about this step. It will be performed implicitly when the transaction is committed. Otherwise you should call `Session.flush()` to ensure that all changes are synchronized with the database.

9.7.2. Committing the database transaction

If you are using the Hibernate `Transaction` API, this looks like:

```
tx.commit(); // flush the Session and commit the transaction
```

If you are managing JDBC transactions yourself you should manually `commit()` the JDBC connection.

```
sess.flush();
sess.connection().commit();  // not necessary for JTA datasource
```

If you decide *not* to commit your changes:

```
tx.rollback();  // rollback the transaction
```

or:

```
// not necessary for JTA datasource, important otherwise
sess.connection().rollback();
```

If you rollback the transaction you should immediately close and discard the current session to ensure that Hibernate's internal state is consistent.

9.7.3. Closing the Session

A call to `Session.close()` marks the end of a session. The main implication of `close()` is that the JDBC connection will be relinquished by the session.

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

```
sess.flush();
sess.connection().commit();  // not necessary for JTA datasource
sess.close();
```

If you provided your own connection, `close()` returns a reference to it, so you can manually close it or return it to the pool. Otherwise `close()` returns it to the pool.

9.7.4. Exception handling

If the `Session` throws an exception (including any `SQLException`), you should immediately rollback the transaction, call `Session.close()` and discard the `Session` instance. Certain methods of `Session` will *not* leave the session in a consistent state.

The following exception handling idiom is recommended:

```
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();
    // do some work
    ...
    tx.commit();
}
catch (Exception e) {
    if (tx!=null) tx.rollback();
    throw e;
}
finally {
    sess.close();
}
```

Or, when manually managing JDBC transactions:

```
Session sess = factory.openSession();
try {
    // do some work
    ...
    sess.flush();
```

```
    sess.connection().commit();
}
catch (Exception e) {
    sess.connection().rollback();
    throw e;
}
finally {
    sess.close();
}
```

Or, when using a datasource enlisted with JTA:

```
UserTransaction ut = .... ;
Session sess = factory.openSession();
try {
    // do some work
    ...
    sess.flush();
}
catch (Exception e) {
    ut.setRollbackOnly();
    throw e;
}
finally {
    sess.close();
}
```

9.8. Lifecyles and object graphs

To save or update all objects in a graph of associated objects, you must either

- `save()`, `saveOrUpdate()` or `update()` each individual object OR
- map associated objects using `cascade="all"` or `cascade="save-update"`.

Likewise, to delete all objects in a graph, either

- `delete()` each individual object OR
- map associated objects using `cascade="all"`, `cascade="all-delete-orphan"` or `cascade="delete"`.

Recommendation:

- 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"`.
- Otherwise, `save()` and `delete()` it explicitly from application code. If you really want to save yourself some extra typing, use `cascade="save-update"` and explicit `delete()`.

Mapping an association (many-to-one, or collection) with `cascade="all"` marks the association as a *parent/child* style relationship where save/update/deletion of the parent results in save/update/deletion of the child(ren). Futhermore, a mere reference to a child from a persistent parent will result in save / update of the child. The 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="all-delete-orphan"`. The precise semantics of cascading operations are as follows:

- If a parent is saved, all children are passed to `saveOrUpdate()`
- If a parent is passed to `update()` or `saveOrUpdate()`, all children are passed to `saveOrUpdate()`
- If a transient 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 transient child is dereferenced by a persistent parent, *nothing special happens* (the application should

explicitly delete the child if necessary) unless `cascade="all-delete-orphan"`, in which case the "orphaned" child is deleted.

Hibernate does not fully implement "persistence by reachability", which would imply (inefficient) persistent garbage collection. However, due to popular demand, Hibernate does support the notion of entities becoming persistent when referenced by another persistent object. Associations marked `cascade="save-update"` behave in this way. If you wish to use this approach throughout your application, its easier to specify the `default-cascade` attribute of the `<hibernate-mapping>` element.

9.9. 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.

```
package net.sf.hibernate.test;

import java.io.Serializable;
import java.util.Date;
import java.util.Iterator;

import net.sf.hibernate.Interceptor;
import net.sf.hibernate.type.Type;

public class AuditInterceptor implements Interceptor, Serializable {

    private int updates;
    private int creates;

    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) {
        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 postFlush(Iterator entities) {
        System.out.println("Creations: " + creates + ", Updates: " + updates);
    }

    public void preFlush(Iterator entities) {
        updates=0;
        creates=0;
    }

    ......
    ......

}
```

The interceptor would be specified when a session is created.

```
Session session = sf.openSession( new AuditInterceptor() );
```

9.10. Metadata API

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 = ......;
Long id = (Long) catMeta.getIdentifier(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
// TODO: what about components?
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 10. Transactions And Concurrency

Hibernate is not itself a database. It is a lightweight object-relational mapping tool. Transaction management is delegated to the underlying database connection. If the connection is enlisted with JTA, operations performed by the `Session` are atomically part of the wider JTA transaction. Hibernate can be seen as a thin adapter to JDBC, adding object- oriented semantics.

10.1. Configurations, Sessions and Factories

A `SessionFactory` is an expensive-to-create, threadsafe object intended to be shared by all application threads. A `Session` is an inexpensive, non-threadsafe object that should be used once, for a single business process, and then discarded. For example, when using Hibernate in a servlet-based application, servlets could obtain a `SessionFactory` using

```
SessionFactory sf = (SessionFactory)getServletContext().getAttribute("my.session.factory");
```

Each call to a service method could create a new `Session`, `flush()` it, `commit()` its connection, `close()` it and finally discard it. (The `SessionFactory` may also be kept in JNDI or in a static *Singleton* helper variable.)

In a stateless session bean, a similar approach could be used. The bean would obtain a `SessionFactory` in `setSessionContext()`. Then each business method would create a `Session`, `flush()` it and `close()` it. Of course, the application should not `commit()` the connection. (Leave that to JTA, the database connection participates automatically in container-managed transactions.)

We use the Hibernate `Transaction` API as discussed previously, a single `commit()` of a Hibernate `Transaction` flushes the state and commits any underlying database connection (with special handling of JTA transactions).

Ensure you understand the semantics of `flush()`. Flushing synchronizes the persistent store with in-memory changes but *not* vice-versa. Note that for all Hibernate JDBC connections/transactions, the transaction isolation level for that connection applies to all operations executed by Hibernate!

The next few sections will discuss alternative approaches that utilize versioning to ensure transaction atomicity. These are considered "advanced" approaches to be used with care.

10.2. Threads and connections

You should observe the following practices when creating Hibernate Sessions:

- Never create more than one concurrent `Session` or `Transaction` instance per database connection.
- Be extremely careful when creating more than one `Session` per database per transaction. The `Session` itself keeps track of updates made to loaded objects, so a different `Session` might see stale data.
- The `Session` is *not* threadsafe! Never access the same `Session` in two concurrent threads. A `Session` is usually only a single unit-of-work!

10.3. Considering object identity

The application may concurrently access the same persistent state in two different units-of-work. 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`, the two notions are equivalent. 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).

This approach leaves Hibernate and the database to worry about concurrency. The application never needs to synchronize on any business object, as long as it sticks to a single thread per `Session` or object identity (within a `Session` the application may safely use `==` to compare objects).

10.4. Optimistic concurrency control

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.

Maintaining isolation of business processes becomes the partial responsibility of the application tier, hence we call this process a long running *application transaction*. A single application transaction usually spans several database transactions. It will be atomar if only one of these database transactions (the last one) stores the updated data, all others simply read data.

The only approach that is consistent with high concurrency and high scalability is optimistic concurrency control with versioning. Hibernate provides for three possible approaches to writing application code that uses optimistic concurrency.

10.4.1. Long session with automatic versioning

A single `Session` instance and its persistent instances are used for the whole application transaction.

The `Session` uses optimistic locking with versioning to ensure that many database transactions appear to the application as a single logical application transaction. 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.

```
// foo is an instance loaded earlier by the Session
session.reconnect();
foo.setProperty("bar");
session.flush();
session.connection().commit();
session.disconnect();
```

The `foo` object still knows which `Session` it was loaded it. As soon as the `Session` has a JDBC connection, we commit the changes to the object.

This pattern is problematic if our `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 propably use this strategy only for a few request/response cycles. This is indeed recommended, as the `Session` will soon also have stale data.

10.4.2. Many sessions with 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 "reassociates" them using `Session.update()` or `Session.saveOrUpdate()`.

```
// foo is an instance loaded by a previous Session
foo.setProperty("bar");
session = factory.openSession();
session.saveOrUpdate(foo);
session.flush();
session.connection().commit();
session.close();
```

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.

10.4.3. Application version checking

Each interaction with the database occurs in a new `Session` that reloads all persistent instances from the database before manipulating them. This approach forces the application to carry out its own version checking to ensure application transaction isolation. (Of course, Hibernate will still *update* version numbers for you.) 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();
int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() );
if ( oldVersion!=foo.getVersion ) throw new StaleObjectStateException();
foo.setProperty("bar");
session.flush();
session.connection().commit();
session.close();
```

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.

10.5. Session disconnection

The first approach described above is to maintain a single `Session` for a whole business process thats spans user think time. (For example, a servlet might keep a `Session` in the user's `HttpSession`.) For performance reasons you should

1. commit the `Transaction` (or JDBC connection) and then
2. disconnect the `Session` from the JDBC connection

before waiting for user activity. The method `Session.disconnect()` will disconnect the session from the JDBC connection and return the connection to the pool (unless you provided the connection).

`Session.reconnect()` obtains a new connection (or you may supply one) and restarts the session. After reconnection, to force a version check on data you aren't updating, you may call `Session.lock()` on any objects that might have been updated by another transaction. You don't need to lock any data that you *are* updating.

Heres an example:

```
SessionFactory sessions;
List fooList;
Bar bar;
....
Session s = sessions.openSession();

Transaction tx = null;
try {
    tx = s.beginTransaction();

    fooList = s.find(
            "select foo from eg.Foo foo where foo.Date = current date"
        // uses db2 date function
    );
    bar = (Bar) s.create(Bar.class);

    tx.commit();
}
catch (Exception e) {
    if (tx!=null) tx.rollback();
    s.close();
    throw e;
}
s.disconnect();
```

Later on:

```
s.reconnect();

try {
    tx = s.beginTransaction();

    bar.setFooTable( new HashMap() );
    Iterator iter = fooList.iterator();
    while ( iter.hasNext() ) {
        Foo foo = (Foo) iter.next();
        s.lock(foo, LockMode.READ);    //check that foo isn't stale
        bar.getFooTable().put( foo.getName(), foo );
    }

    tx.commit();
}
catch (Exception e) {
    if (tx!=null) tx.rollback();
    throw e;
}
finally {
    s.close();
}
```

You can see from this how the relationship between `Transactions` and `Sessions` is many-to-one, A `Session` represents a conversation between the application and the database. The `Transaction` breaks that conversation up into atomic units of work at the database level.

10.6. 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.

Chapter 11. HQL: The Hibernate Query Language

Hibernate is equiped 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.

11.1. 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 `net.sf.hibernate.eg.FOO` is not `net.sf.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.

11.2. 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`.

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 eg.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 eg.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`).

11.3. Associations and joins

We may also assign aliases to associated entities, or even to elements of a collection of values, using a `join`.

```
from eg.Cat as cat
    inner join cat.mate as mate
    left outer join cat.kittens as kitten

from eg.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 eg.Cat as cat
    join cat.mate as mate
    left join cat.kittens as kitten
```

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.

```
from eg.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.

Note that, in the current implementation, only one collection role may be fetched in a query (everything else would be non-performant). Note also that the `fetch` construct may not be used in queries called using `scroll()` or `iterate()`. Finally, note that `full join fetch` and `right join fetch` are not meaningful.

11.4. The select clause

The `select` clause picks which objects and properties to return in the query result set. Consider:

```
select mate
from eg.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 eg.Cat cat
```

You may even select collection elements, using the special `elements` function. The following query returns all kittens of any cat.

```
select elements(cat.kittens) from eg.Cat cat
```

Queries may return properties of any value type including properties of component type:

```
select cat.name from eg.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 eg.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 eg.DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

assuming that the class `Family` has an appropriate constructor.

11.5. 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 eg.Cat cat
```

Collections may also appear inside aggregate functions in the `select` clause.

```
select cat, count( elements(cat.kittens) )
from eg.Cat cat group by cat
```

The supported aggregate functions are

- `avg(...), sum(...), min(...), max(...)`
- `count(*)`
- `count(...), count(distinct ...), count(all...)`

The `distinct` and `all` keywords may be used and have the same semantics as in SQL.

```
select distinct cat.name from eg.Cat cat

select count(distinct cat.name), count(cat) from eg.Cat cat
```

11.6. Polymorphic queries

A query like:

```
from eg.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 eg.Named n, eg.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()`.)

11.7. The where clause

The `where` clause allows you to narrow the list of instances returned.

```
from eg.Cat as cat where cat.name='Fritz'
```

returns instances of `Cat` named 'Fritz'.

```
select foo
from eg.Foo foo, eg.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 eg.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 eg.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 eg.Cat cat, eg.Cat rival where cat.mate = rival.mate

select cat, mate
from eg.Cat cat, eg.Cat mate
where cat.mate = mate
```

The special property (lowercase) `id` may be used to reference the unique identifier of an object. (You may also use its property name.)

```
from eg.Cat as cat where cat.id = 123

from eg.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`.

```
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 eg.Cat cat where cat.class = eg.DomesticCat
```

You may also specify properties of components or composite user types (and of components of components, etc). Never try to use a path-expression that ends in a property of component type (as opposed to a property of a component). For example, if `store.owner` is an entity with a component `address`

```
store.owner.address.city    // okay
store.owner.address         // error!
```

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 eg.AuditLog log, eg.Payment payment
where log.item.class = 'eg.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.

11.8. 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`
- string concatenation ||
- SQL scalar functions like `upper()` and `lower()`
- Parentheses `( )` indicate grouping
- `in`, `between`, `is null`
- JDBC IN parameters `?`
- named parameters `:name`, `:start_date`, `:x1`
- SQL literals `'foo'`, `69`, `'1970-01-01 10:00:01.0'`
- Java `public static final` constants `eg.Color.TABBY`

`in` and `between` may be used as follows:

```
from eg.DomesticCat cat where cat.name between 'A' and 'B'

from eg.DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )
```

and the negated forms may be written

```
from eg.DomesticCat cat where cat.name not between 'A' and 'B'

from eg.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 eg.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 eg.Cat cat where cat.kittens.size > 0

from eg.Cat cat where size(cat.kittens) > 0
```

For indexed collections, you may refer to the minimum and maximum indices using `minIndex` and `maxIndex`. Similarly, you may refer to the minimum and maximum elements of a collection of basic type using `minElement` and `maxElement`.

```
from Calendar cal where cal.holidays.maxElement > current date
```

There are also functional forms (which, unlike the constructs above, are not case sensitive):

```
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 eg.Cat as mother, eg.Cat as kit
where kit in elements(foo.kittens)

select p from eg.NameList list, eg.Person p
where p.name = some elements(list.names)

from eg.Cat cat where exists elements(cat.kittens)

from eg.Player p where 3 > all elements(p.scores)

from eg.Show show where 'fizard' in indices(show.acts)
```

Note that these constructs - `size`, `elements`, `indices`, `minIndex`, `maxIndex`, `minElement`, `maxElement` - have certain usage restrictions:

- in a `where` clause: only for databases with subselects
- in a `select` clause: only `elements` and `indices` make sense

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 eg.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
    )
```

11.9. The order by clause

The list returned by a query may be ordered by any property of a returned class or components:

```
from eg.DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

The optional `asc` or `desc` indicate ascending or descending order respectively.

11.10. 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 eg.Cat cat
group by cat.color

select foo.id, avg( elements(foo.names) ), max( indices(foo.names) )
from eg.Foo foo
group by foo.id
```

Note: You may use the `elements` and `indices` constructs inside a select clause, even on databases with no subselects.

A `having` clause is also allowed.

```
select cat.color, sum(cat.weight), count(cat)
from eg.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 (ie. not in MySQL).

```
select cat
from eg.Cat cat
    join cat.kittens kitten
group by cat
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.

11.11. 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 eg.Cat as fatcat
where fatcat.weight > (
    select avg(cat.weight) from eg.DomesticCat cat
)

from eg.DomesticCat as cat
where cat.name = some (
    select name.nickName from eg.Name as name
)

from eg.Cat as cat
where not exists (
    from eg.Cat as mate where mate.mate = cat
)

from eg.DomesticCat as cat
where cat.name not in (
    select name.nickName from eg.Name as name
)
```

11.12. HQL examples

Hibernate queries can be quite powerful and complex. In fact, the power of the query language is one of Hi-

bernate'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
```

11.13. 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 in class 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();
```

Chapter 12. Criteria Queries

Hibernate now features an intuitive, extensible criteria query API. For now, this API is less powerful and than the more mature HQL query facilities. In particular, criteria queries do not support projection or aggregation.

12.1. Creating a `Criteria` instance

The interface `net.sf.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();
```

12.2. Narrowing the result set

An individual query criterion is an instance of the interface `net.sf.hibernate.expression.Criterion`. The class `net.sf.hibernate.expression.Expression` defines factory methods for obtaining certain built-in `Criterion` types.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "Fritz%") )
    .add( Expression.between("weight", minWeight, maxWeight) )
    .list();
```

Expressions may be grouped logically.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "Fritz%") )
    .add( Expression.or(
        Expression.eq( "age", new Integer(0) ),
        Expression.isNull("age")
    ) )
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.in( "name", new String[] { "Fritz", "Izi", "Pk" } ) )
    .add( Expression.disjunction()
        .add( Expression.isNull("age") )
            .add( Expression.eq("age", new Integer(0) ) )
            .add( Expression.eq("age", new Integer(1) ) )
            .add( Expression.eq("age", new Integer(2) ) )
    ) )
    .list();
```

There are quite a range of built-in criterion types (`Expression` subclasses), but one that is especially useful lets you specify SQL directly.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.sql("lower({alias}.name) like lower(?)", "Fritz%", Hibernate.STRING) )
    .list();
```

The `{alias}` placeholder with be replaced by the row alias of the queried entity.

12.3. Ordering the results

You may order the results using `net.sf.hibernate.expression.Order`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "F%")
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

12.4. Associations

You may easily specify constraints upon related entities by navigating associations using `createCriteria()`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "F%")
    .createCriteria("kittens")
        .add( Expression.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( Expression.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 `returnMaps()`.

```
List cats = sess.createCriteria(Cat.class)
    .createCriteria("kittens", "kt")
        .add( Expression.eq("name", "F%") )
    .returnMaps()
    .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");
}
```

12.5. Dynamic association fetching

You may specify association fetching semantics at runtime using `setFetchMode()`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "Fritz%") )
    .setFetchMode("mate", FetchMode.EAGER)
```

```
    .setFetchMode("kittens", FetchMode.EAGER)
    .list();
```

This query will fetch both `mate` and `kittens` by outer join.

12.6. Example queries

The class `net.sf.hibernate.expression.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();
```

Chapter 13. Native SQL Queries

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 the CONNECT keyword in Oracle. This also allows for a cleaner migration path from a direct SQL/JDBC based application to Hibernate.

13.1. Creating a SQL based `Query`

SQL queries are exposed through the same `Query` interface, just like ordinary HQL queries. The only difference is the use of `Session.createSQLQuery()`.

```
Query sqlQuery = sess.createSQLQuery("select {cat.*} from cats {cat}", "cat", Cat.class);
sqlQuery.setMaxResults(50);
List cats = sqlQuery.list();
```

The three parameters provided to `createSQLQuery()` are:

- the SQL query string
- a table alias name
- the persistent class returned by the query

The alias name is used inside the sql string to refer to the properties of the mapped class (in this case `Cat`). You may retrieve multiple objects per row by supplying a `String` array of alias names and a `Class` array of corresponding classes.

13.2. Alias and property references

The `{cat.*}` notation used above is a shorthand for "all properties". You may even list the properties explicity, but you must let Hibernate provide SQL column aliases for each property. The placeholders for these column aliases are the property name qualified by the table alias. In the following example, we retrieve `Cats` 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.

```
String sql = "select cat.originalId as {cat.id}, "
    + "  cat.mateid as {cat.mate}, cat.sex as {cat.sex}, "
    + "  cat.weight*10 as {cat.weight}, cat.name as {cat.name}"
    + "     from cat_log cat where {cat.mate} = :catId"
List loggedCats = sess.createSQLQuery(sql, "cat", Cat.class)
    .setLong("catId", catId)
    .list();
```

Note: if you list each property explicitly, you must include all properties of the class *and its subclasses*!

13.3. 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.

```
List people = sess.getNamedQuery("mySqlQuery")
```

```
    .setMaxResults(50)
    .list();
```

```
<sql-query name="mySqlQuery">
    <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 'Hiber%'
</sql-query>
```

Chapter 14. Improving performance

14.1. 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.

14.1.1. 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"`.)

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.)

14.1.2. Lists, maps 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 sets are the most performant collection types. (With the caveat that a set might be less efficient for some collections of values.)

Sets are expected to be the most common kind of collection in Hibernate applications.

There is an undocumented feature in this release of Hibernate. The `<idbag>` mapping implements bag semantics for a collection of values or a many to many association and is more efficient that any other style of collection in this case!

14.1.3. 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();
```

14.1.4. One shot delete

Occasionally, deleting collection elements one by one can be extremely inefficient. Hibernate isn't completly 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.

We have already shown how you can use lazy initialization for persistent collections in the chapter about collection mappings. A similar effect is achievable for ordinary object references, using CGLIB proxies. We have also mentioned how Hibernate caches persistent objects at the level of a `Session`. More aggressive caching strategies may be configured upon a class-by-class basis.

In the next section, we show you how to use these features, which may be used to achieve much higher performance, where necessary.

14.2. Proxies for Lazy Initialization

Hibernate implements lazy initializing proxies for persistent objects using runtime bytecode enhancement (via the excellent CGLIB library).

The mapping file declares a class or interface to use as the proxy interface for that class. The recommended approach is to specify the class itself:

```
<class name="eg.Order" proxy="eg.Order">
```

The runtime type of the proxies will be a subclass of `Order`. Note that the proxied class must implement a default constructor with at least package visibility.

There are some gotchas to be aware of when extending this approach to polymorphic classes, eg.

```
<class name="eg.Cat" proxy="eg.Cat">
    ......
    <subclass name="eg.DomesticCat" proxy="eg.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);  // required 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 inheritence 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="eg.Cat" proxy="eg.ICat">
    ......
    <subclass name="eg.DomesticCat" proxy="eg.IDomesticCat">
        .....
    </subclass>
</class>
```

where `Cat` implements the interface `ICat` and `DomesticCat` implements the interface `IDomesticCat`. Then proxies for instances of `Cat` and `DomesticCat` may be returned by `load()` or `iterate()`. (Note that `find()` does not return proxies.)

```
ICat cat = (ICat) session.load(Cat.class, catid);
Iterator iter = session.iterate("from cat in class eg.Cat where cat.name='fritz'");
ICat fritz = (ICat) iter.next();
```

Relationships are also lazily initialized. This means you must declare any properties to be of type `ICat`, not `Cat`.

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()`.

Exceptions that occur while initializing a proxy are wrapped in a `LazyInitializationException`.

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 lazyily 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.

14.3. 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).

By default, Hibernate uses EHCache for JVM-level caching. (JCS support is now deprecated and will be removed in a future version of Hibernate.) You may choose a different implementation by specifying the name of a class that implements `net.sf.hibernate.cache.CacheProvider` using the property `hibernate.cache.provider_class`.

Table 14.1. Cache Providers

Cache	Provider class	Type	Cluster Safe	Query Cache Supported
Hashtable (not intended for production use)	`net.sf.hibernate.cache.HashtableCacheProvider`	memory		yes
EHCache	`net.sf.hibernate.cache.EhCacheProvider`	memory, disk		yes
OSCache	`net.sf.hibernate.cache.OSCacheProvider`	memory, disk		yes
SwarmCache	`net.sf.hibernate.cache.SwarmCacheProvider`	clustered (ip multicast)	yes (clustered invalidation)	
JBoss TreeCache	`net.sf.hibernate.cache.TreeCacheProvider`	clustered (ip multicast), transactional	yes (replication)	

14.3.1. 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"  (1)
/>
```

(1) `usage` specifies the caching strategy: `transactional`, `read-write`, `nonstrict-read-write` or `read-only`

Alternatively (preferrably?), you may specify `<class-cache>` and `<collection-cache>` elements in `hibernate.cfg.xml`.

The `usage` attribute specifies a *cache concurrency strategy*.

14.3.2. 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. Its even perfectly safe for use in a cluster.

```
<class name="eg.Immutable" mutable="false">
    <cache usage="read-only"/>
    ....
</class>
```

14.3.3. 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>
```

14.3.4. 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.

14.3.5. 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.

Table 14.2. Cache Concurrency Strategy Support

Cache	read-only	nonstrict-read-write	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

14.4. Managing the `Session` Cache

Whenever you pass an object to `save()`, `update()` or `saveOrUpdate()` and whenever you retrieve an object using `load()`, `find()`, `iterate()`, or `filter()`, 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 cache.

```
Iterator cats = sess.iterate("from eg.Cat as cat"); //a huge result set
```

```
while ( cats.hasNext() ) {
    Cat cat = (Cat) iter.next();
    doSomethingWithACat(cat);
    sess.evict(cat);
}
```

Hibernate will evict associated entities automatically if the association is mapped with `cascade="all"` or `cascade="all-delete-orphan"`.

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.

14.5. 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 by setting the property `hibernate.cache.use_query_cache=true`. This causes the creation of two cache regions - one holding cached query result sets (`net.sf.hibernate.cache.QueryCache`), the other holding timestamps of most recent updates to queried tables (`net.sf.hibernate.cache.UpdateTimestampsCache`). Note that the query cache does not cache the state of any entities in the result set; it caches only identifier values and results of value type. So the query cache is usually 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 may call `Query.setForceCacheRefresh()` to `true`. This is particularly useful in cases where underlying data may have been updated via a seperate process (i.e., not modified through Hibernate) and allows the application to selectively refresh the query cache regions based on its knowledge of those events. This is an alternative to eviction of a query cache region. If you need fine-grained refresh control for many queries, use this function instead of a new region for each query.

Chapter 15. Toolset Guide

Roundtrip engineering with Hibernate is possible using a set of commandline tools maintained as part of the Hibernate project, along with Hibernate support built into XDoclet, Middlegen and AndroMDA.

The Hibernate main package comes bundled with the most important tool (it can even be used from "inside" Hibernate on-the-fly):

- DDL schema generation from a mapping file (aka `SchemaExport`, `hbm2ddl`)

Other tools directly provided by the Hibernate project are delivered with a separate package, *Hibernate Extensions*. This package includes tools for the following tasks:

- Java source generation from a mapping file (aka `CodeGenerator`, `hbm2java`)
- mapping file generation from compiled Java classes or from Java source with XDoclet markup (aka `MapGenerator`, `class2hbm`)

There's actually another utitily living in Hibernate Extensions: `ddl2hbm`. It is considered deprecated and will no longer be maintained, Middlegen does a better job for the same task.

Third party tools with Hibernate support are:

- Middlegen (mapping file generation from an existing database schema)
- AndroMDA (MDA (Model-Driven Architecture) approach generating code for persistent classes from UML diagrams and their XML/XMI representation)

These 3rd party tools are not documented in this reference. Please refer to the Hibernate website for up-to-date information (a snapshot of the site is included in the Hibernate main package).

15.1. Schema Generation

DDL may be generated from your mapping files by a command line utility. A batch file is located in the `hibernate-x.x.x/bin` directory of the core Hibernate package.

The generated schema include 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.

15.1.1. Customizing the schema

Many Hibernate mapping elements define an optional attribute named `length`. You may set the length of a column with this attribute. (Or, for numeric/decimal data types, the precision.)

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).

Some tags accept an `index` attribute for specifying the name of an index for that column. A `unique-key` attribute can be used to group columns in a single unit key constraint. Currently, the specified value of the `unique-`

`key` attribute is *not* used to name the constraint, only to group the columns in the mapping file.

Examples:

```
<property name="foo" type="string" length="64" not-null="true"/>

<many-to-one name="bar" foreign-key="fk_foo_bar" not-null="true"/>

<element column="serial_number" type="long" not-null="true" unique="true"/>
```

Alternatively, these elements also accept a child `<column>` element. This is particularly useful for multi-column types:

```
<property name="foo" type="string">
    <column name="foo" length="64" not-null="true" sql-type="text"/>
</property>

<property name="bar" type="my.customtypes.MultiColumnType"/>
    <column name="fee" not-null="true" index="bar_idx"/>
    <column name="fi" not-null="true" index="bar_idx"/>
    <column name="fo" not-null="true" index="bar_idx"/>
</property>
```

The `sql-type` attribute allows the user to override the default mapping of Hibernate type to SQL datatype.

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>
```

Table 15.1. Summary

Attribute	Values
`length`	number
`not-null`	`true|false`
`unique`	`true|false`
`index`	`index_name`
`unique-key`	`unique_key_name`
`foreign-key`	`foreign_key_name`
`sql-type`	`column_type`
`check`	SQL expression

15.1.2. Running the tool

The `SchemaExport` tool writes a DDL script to standard out and/or executes the DDL statements.

`java -cp` *hibernate_classpaths* `net.sf.hibernate.tool.hbm2ddl.SchemaExport` *options mapping_files*

Table 15.2. `SchemaExport` Command Line Options

Option	Description
`--quiet`	don't output the script to stdout
`--drop`	only drop the tables
`--text`	don't export to the database
`--output=my_schema.ddl`	output the ddl script to a file
`--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=x`	set an end of line delimiter for the script

You may even embed `SchemaExport` in your application:

```
Configuration cfg = ....;
new SchemaExport(cfg).create(false, true);
```

15.1.3. 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:

Table 15.3. SchemaExport Connection Properties

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

15.1.4. Using Ant

You can call `SchemaExport` from your Ant build script:

```
<target name="schemaexport">
    <taskdef name="schemaexport"
        classname="net.sf.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>
```

15.1.5. 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* `net.sf.hibernate.tool.hbm2ddl.SchemaUpdate` *options mapping_files*

Table 15.4. `SchemaUpdate` Command Line Options

Option	Description
`--quiet`	don't output the script to stdout
`--properties=hibernate.properties`	read database properties from a file

You may embed `SchemaUpdate` in your application:

```
Configuration cfg = ....;
new SchemaUpdate(cfg).execute(false);
```

15.1.6. Using Ant for incremental schema updates

You can call `SchemaUpdate` from the Ant script:

```
<target name="schemaupdate">
    <taskdef name="schemaupdate"
        classname="net.sf.hibernate.tool.hbm2ddl.SchemaUpdateTask"
        classpathref="class.path"/>

    <schemaupdate
        properties="hibernate.properties"
        quiet="no">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaupdate>
</target>
```

15.2. Code Generation

The Hibernate code generator may be used to generate skeletal Java implementation classes from a Hibernate mapping file. This tool is included in the Hibernate Extensions package (a seperate download).

`hbm2java` parses the mapping files and generates fully working Java source files from these. Thus with `hbm2java` one could "just" provide the `.hbm` files, and then don't worry about hand-writing/coding the Java files.

`java -cp` *hibernate_classpaths* `net.sf.hibernate.tool.hbm2java.CodeGenerator` *options mapping_files*

Table 15.5. Code Generator Command Line Options

Option	Description
`--output=`*output_dir*	root directory for generated code
`--config=`*config_file*	optional file for configuring hbm2java

15.2.1. The config file (optional)

The config file provides for a way to specify multiple "renderers" for the source code and to declare `<meta>` attributes that is "global" in scope. See more about this in the `<meta>` attribute section.

```
<codegen>
    <meta attribute="implements">codegen.test.IAuditable</meta>
    <generate renderer="net.sf.hibernate.tool.hbm2java.BasicRenderer"/>
    <generate
        package="autofinders.only"
        suffix="Finder"
        renderer="net.sf.hibernate.tool.hbm2java.FinderRenderer"/>
</codegen>
```

This config file declares a global meta attribute "implements" and specify two renderers, the default one (BasicRenderer) and a renderer that generates Finder's (See more in "Basic Finder generation" below).

The second renderer is provided with a package and suffix attribute.

The package attribute specifies that the generated source files from this renderer should be placed here instead of the package scope specified in the `.hbm` files.

The suffix attribute specifies the suffix for generated files. E.g. here a file named `Foo.java` would be `FooFinder.java` instead.

It is also possible to send down arbitrary parameters to the renders by adding `<param>` attributes to the `<generate>` elements.

hbm2java currently has support for one such parameter, namely `generate-concrete-empty-classes` which informs the BasicRenderer to only generate empty concrete classes that extends a base class for all your classes. The following config.xml example illustrate this feature

```
<codegen>
  <generate prefix="Base" renderer="net.sf.hibernate.tool.hbm2java.BasicRenderer"/>
  <generate renderer="net.sf.hibernate.tool.hbm2java.BasicRenderer">
    <param name="generate-concrete-empty-classes">true</param>
    <param name="baseclass-prefix">Base</param>
  </generate>
</codegen>
```

Notice that this config.xml configure 2 (two) renderers. One that generates the Base classes, and a second one that just generates empty concrete classes.

15.2.2. The `meta` attribute

The `<meta>` tag is a simple way of annotating the `hbm.xml` with information, so tools have a natural place to store/read information that is not directly related to the Hibernate core.

You can use the `<meta>` tag to tell `hbm2java` to only generate "protected" setters, have classes always implement a certain set of interfaces or even have them extend a certain base class and even more.

The following example:

```
<class name="Person">
    <meta attribute="class-description">
        Javadoc for the Person class
        @author Frodo
    </meta>
    <meta attribute="implements">IAuditable</meta>
    <id name="id" type="long">
        <meta attribute="scope-set">protected</meta>
        <generator class="increment"/>
    </id>
    <property name="name" type="string">
        <meta attribute="field-description">The name of the person</meta>
    </property>
</class>
```

will produce something like the following (code shortened for better understanding). Notice the Javadoc comment and the protected set methods:

```
// default package

import java.io.Serializable;
import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder;
import org.apache.commons.lang.builder.ToStringBuilder;

/**
 *         Javadoc for the Person class
 *         @author Frodo
 *
 */
public class Person implements Serializable, IAuditable {

    /** identifier field */
    public Long id;

    /** nullable persistent field */
    public String name;

    /** full constructor */
    public Person(java.lang.String name) {
        this.name = name;
    }

    /** default constructor */
    public Person() {
    }

    public java.lang.Long getId() {
        return this.id;
    }

    protected void setId(java.lang.Long id) {
```

```
        this.id = id;
    }

    /**
     * The name of the person
     */
    public java.lang.String getName() {
        return this.name;
    }

    public void setName(java.lang.String name) {
        this.name = name;
    }

}
```

Table 15.6. Supported meta tags

Attribute	Description
`class-description`	inserted into the javadoc for classes
`field-description`	inserted into the javadoc for fields/properties
`interface`	If true an interface is generated instead of an class.
`implements`	interface the class should implement
`extends`	class the class should extend (ignored for subclasses)
`generated-class`	overrule the name of the actual class generated
`scope-class`	scope for class
`scope-set`	scope for setter method
`scope-get`	scope for getter method
`scope-field`	scope for actual field
`use-in-tostring`	include this property in the `toString()`
`implement-equals`	include a `equals()` and `hashCode()` method in this class.
`use-in-equals`	include this property in the `equals()` and `hashCode()` method.
`bound`	add propertyChangeListener support for a property
`constrained`	bound + vetoChangeListener support for a property
`gen-property`	property will not be generated if false (use with care)
`property-type`	Overrides the default type of property. Use this with any tag's to specify the concrete type instead of just Object.
`class-code`	Extra code that will inserted at the end of the class
`extra-import`	Extra import that will inserted at the end of all other imports
`finder-method`	see "Basic finder generator" below
`session-method`	see "Basic finder generator" below

Attributes declared via the <meta> tag are per default "inherited" inside an hbm.xml file.

What does that mean? It means that if you e.g want to have all your classes implement IAuditable then you just add an <meta attribute="implements">IAuditable</meta> in the top of the hbm.xml file, just after <hibernate-mapping>. Now all classes defined in that hbm.xml file will implement IAuditable! (Except if a class also has an "implements" meta attribute, because local specified meta tags always overrules/replaces any inherited meta tags).

Note: This applies to *all* <meta>-tags. Thus it can also e.g. be used to specify that all fields should be declare protected, instead of the default private. This is done by adding <meta attribute="scope-field">protected</meta> at e.g. just under the <class> tag and all fields of that class will be protected.

To avoid having a <meta>-tag inherited then you can simply specify inherit="false" for the attribute, e.g. <meta attribute="scope-class" inherit="false">public abstract</meta> will restrict the "class-scope" to the current class, not the subclasses.

15.2.3. Basic finder generator

It is now possible to have hbm2java generate basic finders for Hibernate properties. This requires two things in the hbm.xml files.

The first is an indication of which fields you want to generate finders for. You indicate that with a meta block inside a property tag such as:

```
<property name="name" column="name" type="string">
     <meta attribute="finder-method">findByName</meta>
</property>
```

The finder method name will be the text enclosed in the meta tags.

The second is to create a config file for hbm2java of the format:

```
<codegen>
    <generate renderer="net.sf.hibernate.tool.hbm2java.BasicRenderer"/>
    <generate suffix="Finder" renderer="net.sf.hibernate.tool.hbm2java.FinderRenderer"/>
</codegen>
```

And then use the param to hbm2java --config=xxx.xml where xxx.xml is the config file you just created.

An optional parameter is meta tag at the class level of the format:

```
<meta attribute="session-method">
    com.whatever.SessionTable.getSessionTable().getSession();
</meta>
```

Which would be the way in which you get sessions if you use the *Thread Local Session* pattern (documented in the Design Patterns area of the Hibernate website).

15.2.4. Velocity based renderer/generator

It is now possible to use velocity as an alternative rendering mechanism. The follwing config.xml shows how to configure hbm2java to use its velocity renderer.

```

```

```
<codegen>
 <generate renderer="net.sf.hibernate.tool.hbm2java.VelocityRenderer">
  <param name="template">pojo.vm</param>
 </generate>
</codegen>
```

The parameter named `template` is a resource path to the velocity macro file you want to use. This file must be available via the classpath for hbm2java. Thus remember to add the directory where pojo.vm is located to your ant task or shell script. (The default location is `./tools/src/velocity`)

Be aware that the current `pojo.vm` generates only the most basic parts of the java beans. It is not as complete and feature rich as the default renderer - primarily a lot of the `meta` tags are not supported.

15.3. Mapping File Generation

A skeletal mapping file may be generated from compiled persistent classes using a command line utility called `MapGenerator`. This utility is part of the Hibernate Extensions package.

The Hibernate mapping generator provides a mechanism to produce mappings from compiled classes. It uses Java reflection to find *properties* and uses heuristics to guess an appropriate mapping from the property type. The generated mapping is intended to be a starting point only. There is no way to produce a full Hibernate mapping without extra input from the user. However, the tool does take away some of the repetitive "grunt" work involved in producing a mapping.

Classes are added to the mapping one at a time. The tool will reject classes that it judges are are not *Hibernate persistable*.

To be *Hibernate persistable* a class

- must not be a primitive type
- must not be an array
- must not be an interface
- must not be a nested class
- must have a default (zero argument) constructor.

Note that interfaces and nested classes actually are persistable by Hibernate, but this would not usually be intended by the user.

`MapGenerator` will climb the superclass chain of all added classes attempting to add as many Hibernate persistable superclasses as possible to the same database table. The search stops as soon as a property is found that has a name appearing on a list of *candidate UID names*.

The default list of candidate UID property names is: `uid`, `UID`, `id`, `ID`, `key`, `KEY`, `pk`, `PK`.

Properties are discovered when there are two methods in the class, a setter and a getter, where the type of the setter's single argument is the same as the return type of the zero argument getter, and the setter returns `void`. Furthermore, the setter's name must start with the string `set` and either the getter's name starts with `get` or the getter's name starts with `is` and the type of the property is boolean. In either case, the remainder of their names must match. This matching portion is the name of the property, except that the initial character of the property name is made lower case if the second letter is lower case.

The rules for determining the database type of each property are as follows:

1. If the Java type is `Hibernate.basic()`, then the property is a simple column of that type.

2. For `hibernate.type.Type` custom types and `PersistentEnum` a simple column is used as well.
3. If the property type is an array, then a Hibernate array is used, and `MapGenerator` attempts to reflect on the array element type.
4. If the property has type `java.util.List`, `java.util.Map`, or `java.util.Set`, then the corresponding Hibernate types are used, but `MapGenerator` cannot further process the insides of these types.
5. If the property's type is any other class, `MapGenerator` defers the decision on the database representation until all classes have been processed. At this point, if the class was discovered through the superclass search described above, then the property is an `many-to-one` association. If the class has any properties, then it is a `component`. Otherwise it is serializable, or not persistable.

15.3.1. Running the tool

The tool writes XML mappings to standard out and/or to a file.

When invoking the tool you must place your compiled classes on the classpath.

`java -cp` *hibernate_and_your_class_classpaths* `net.sf.hibernate.tool.class2hbm.MapGenerator` *options and classnames*

There are two modes of operation: command line or interactive.

The interactive mode is selected by providing the single command line argument `--interact`. This mode provides a prompt response console. Using it you can set the UID property name for each class using the `uid=XXX` command where `XXX` is the UID property name. Other command alternatives are simply a fully qualified class name, or the command done which emits the XML and terminates.

In command line mode the arguments are the options below interspersed with fully qualified class names of the classes to be processed. Most of the options are meant to be used multiple times; each use affects subsequently added classes.

Table 15.7. MapGenerator Command Line Options

Option	Description
`--quiet`	don't output the O-R Mapping to stdout
`--setUID=uid`	set the list of candidate UIDs to the singleton uid
`--addUID=uid`	add uid to the front of the list of candidate UIDs
`--select=`*mode*	mode use select mode *mode*(e.g., *distinct* or *all*) for subsequently added classes
`--depth=<small-int>`	limit the depth of component data recursion for subsequently added classes
`--output=my_mapping.xml`	output the O-R Mapping to a file
full.class.Name	add the class to the mapping
`--abstract=`*full.class.Name*	see below

The abstract switch directs the map generator tool to ignore specific super classes so that classes with common inheritance are not mapped to one large table. For instance, consider these class hierarchies:

```
Animal-->Mammal-->Human
```

```
Animal-->Mammal-->Marsupial-->Kangaroo
```

If the `--abstract`switch is *not* used, all classes will be mapped as subclasses of `Animal`, resulting in one large table containing all the properties of all the classes plus a discriminator column to indicate which subclass is actually stored. If `Mammal` is marked as `abstract`, `Human` and `Marsupial` will be mapped to separate `<class>` declarations and stored in separate tables. `Kangaroo` will still be a subclass of `Marsupial` unless `Marsupial` is also marked as `abstract`.

Chapter 16. 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!

16.1. 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.

16.2. 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.

The underlying cause 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();
```

16.3. 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()`.

16.4. Using cascading `update()`

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 (synthetic) identifier properties of type `java.lang.Long`. Hibernate will use the identifier property value to determine which of the children are new. (You may also use the version or timestamp property, see Section 9.4.2, “Updating detached objects”.)

The `unsaved-value` attribute is used to specify the identifier value of a newly instantiated instance. `unsaved-value` defaults to "null", which is perfect for a `Long` identifier type. If we would have used a primitive identitifier property, we would need to specify

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

for the `Child` mapping. (There is also an `unsaved-value` attribute for version and timestamp property mappings.)

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, thats all very well for the case of a generated identifier, but what about assigned identifiers and composite identifiers? This is more difficult, since `unsaved-value` can't distinguish between a newly instantiated object (with an identifier assigned by the user) and an object loaded in a previous session. In these cases, you will probably need to give Hibernate a hint; either

- define `unsaved-value="null"` or `unsaved-value="negative"` on a `<version>` or `<timestamp>` property mapping for the class.
- set `unsaved-value="none"` and explicitly `save()` newly instantiated children before calling `update(parent)`
- set `unsaved-value="any"` and explicitly `update()` previously persistent children before calling `update(parent)`

`none` is the default `unsaved-value` for assigned and composite identifiers.

There is one further possibility. There is a new `Interceptor` method named `isUnsaved()` which lets the application implement its own strategy for distinguishing newly instantiated objects. For example, you could define a base class for your persistent classes.

```
public class Persistent {
    private boolean _saved = false;
    public void onSave() {
        _saved=true;
    }
    public void onLoad() {
        _saved=true;
    }
    ......
    public boolean isSaved() {
        return _saved;
    }
}
```

(The `saved` property is non-persistent.) Now implement `isUnsaved()`, along with `onLoad()` and `onSave()` as follows.

```
public Boolean isUnsaved(Object entity) {
    if (entity instanceof Persistent) {
        return new Boolean( !( (Persistent) entity ).isSaved() );
    }
    else {
        return null;
    }
}

public boolean onLoad(Object entity,
    Serializable id,
    Object[] state,
```

```
    String[] propertyNames,
    Type[] types) {

    if (entity instanceof Persistent) ( (Persistent) entity ).onLoad();
    return false;
}

public boolean onSave(Object entity,
    Serializable id,
    Object[] state,
    String[] propertyNames,
    Type[] types) {

    if (entity instanceof Persistent) ( (Persistent) entity ).onSave();
    return false;
}
```

16.5. Conclusion

There is quite a bit to digest here and it might look confusing first time around. However, in practice, it all works out quite 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. (However, they *may* have a surrogate primary key, using an `<idbag>` mapping.)

Chapter 17. Example: Weblog Application

17.1. 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;
    }
}
```

17.2. Hibernate Mappings

The XML mappings should now be quite straightforward.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping package="eg">

    <class
        name="Blog"
        table="BLOGS"
        lazy="true">

        <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"
            lazy="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 2.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.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>
```

17.3. 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 net.sf.hibernate.HibernateException;
import net.sf.hibernate.Query;
import net.sf.hibernate.Session;
import net.sf.hibernate.SessionFactory;
import net.sf.hibernate.Transaction;
import net.sf.hibernate.cfg.Configuration;
import net.sf.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.save(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)"
            );
```

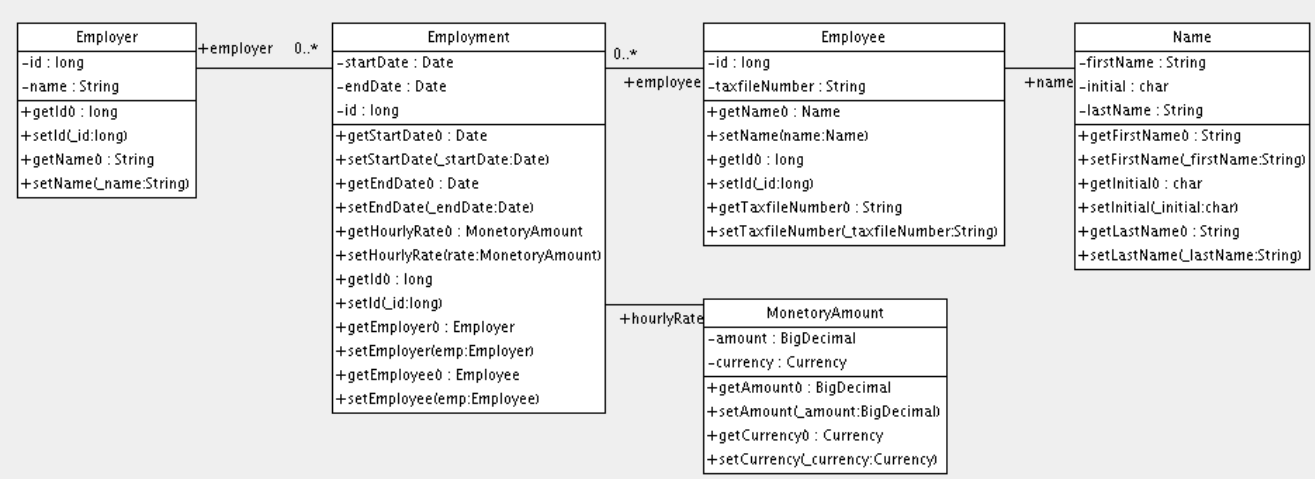
```
            q.setMaxResults(max);
            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }

    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.list().get(0);
            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;
    }
}
```

Chapter 18. Example: Various Mappings

This chapters shows off some more complex association mappings.

18.1. 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 monetory 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="MonetoryAmount">
            <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">
```

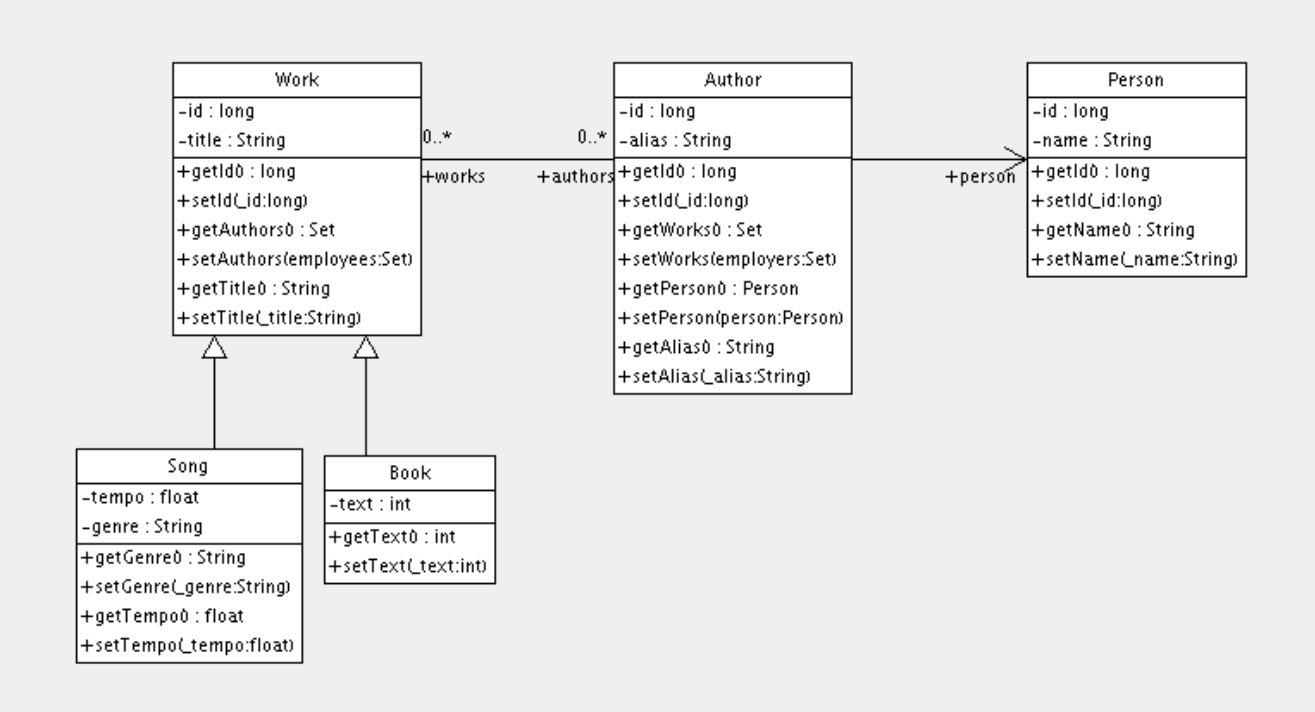
```
            <generator class="sequence">
                <param name="sequence">employee_id_seq</param>
            </generator>
        </id>
        <property name="taxfileNumber"/>
        <component name="name" class="Name">
            <property name="firstName"/>
            <property name="initial"/>
            <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
```

18.2. 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" lazy="true">
            <key>
                <column name="work_id" not-null="true"/>
            </key>
            <many-to-many class="Author">
                <column name="author_id" not-null="true"/>
            </many-to-many>
        </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" lazy="true">
            <key column="author_id"/>
            <many-to-many class="Work" column="work_id"/>
```

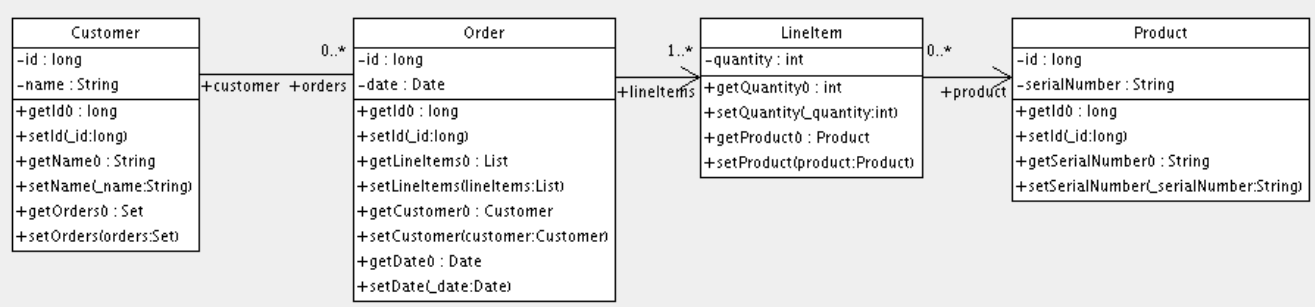
```
        </set>

    </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
```

18.3. 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" lazy="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" lazy="true">
            <key column="order_id"/>
            <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
```

Chapter 19. 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) and of a non-primitive type. For maximum flexibility, use `java.lang.Long` or `java.lang.String`.

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 `net.sf.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 `net.sf.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 (eg. mass update / delete) 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 need to use direct JDBC, it might be worth opening a Hibernate `Session` and using that SQL 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 `saveOrUpdate()`.

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.update()`

or `Session.saveOrUpdate()` to update the persistent state of an object.

In a two tiered architecture, consider using session disconnection.

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. This Application Transaction might span several client requests and response cycles. Either use Detached Objects or, in two tiered architectures, simply disconnect the Hibernate Session from the JDBC connection and reconnect it for each subsequent request. Never use a single Session for more than one Application Transaction usecase, otherwise, you will run into 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 `find()` instead. Some exceptions are recoverable, for example the `StaleObjectStateException` and `ObjectNotFoundException`.

Prefer lazy fetching for associations.

Use eager (outer-join) fetching sparingly. Use proxies and/or lazy collections for most associations to classes that are not cached at the JVM-level. For associations to cached classes, where there is a high probability of a cache hit, explicitly disable eager fetching using `outer-join="false"`. When an outer-join fetch is appropriate to a particular use case, use a query with a `left join`.

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!)

Implement `equals()` and `hashCode()` using a unique business key.

If you compare objects outside of the Session scope, you have to implement `equals()` and `hashCode()`. Inside the Session scope, Java object identity is guaranteed. If you implement these methods, never ever use the database identifier! A transient object doesn't have an identifier value and Hibernate would assign a value when the object is saved. If the object is in a Set while being saved, the hash code changes, breaking the contract. To implement `equals()` and `hashCode()`, use a unique business key, that is, compare a unique combination of class properties. Remember that this key has to be stable and unique only while the object is in a Set, not for the whole lifetime (not as stable as a database primary key). Never use collections in the `equals()` comparison (lazy loading) and be careful with other associated classes that might be proxied.

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.