



Hibernate

Version: 2.1.3

#####

| | |
|---|-----------|
| ##### | vi |
| 1. ##### # ##### Tomcat | 1 |
| 1.1. ##### # Hibernate | 1 |
| 1.2. ##### (persistent class) | 4 |
| 1.3. ##### | 5 |
| 1.4. ##### # | 6 |
| 1.5. ##### | 8 |
| 2. ##### | 10 |
| 2.1. ##### | 10 |
| 2.2. ##### # JMX | 12 |
| 2.3. ##### JCA | 12 |
| 3. ##### SessionFactory | 13 |
| 3.1. ##### | 13 |
| 3.2. ##### SessionFactory | 13 |
| 3.3. JDBC #####, ##### | 14 |
| 3.4. JDBC #####, ##### Hibernate | 14 |
| 3.5. ##### | 16 |
| 3.5.1. ##### SQL | 19 |
| 3.5.2. ##### # ##### (outer join) | 20 |
| 3.5.3. ##### | 20 |
| 3.5.4. ##### # CacheProvider | 21 |
| 3.5.5. Transaction strategy configuration | 21 |
| 3.5.6. ##### # JNDI (JNDI-bound SessionFactory) | 22 |
| 3.5.7. ##### # | 22 |
| 3.6. ##### (##### #, logging) | 22 |
| 3.7. ##### NamingStrategy (#####) | 22 |
| 3.8. ##### XML-##### | 23 |
| 4. ##### (Persistent Classes) | 25 |
| 4.1. ##### POJO-##### | 25 |
| 4.1.1. ##### # ##### | 26 |
| 4.1.2. ##### | 26 |
| 4.1.3. #####-##### (#####) | 26 |
| 4.1.4. ##### #-final ##### (#####) | 27 |
| 4.2. ##### | 27 |
| 4.3. ##### equals() # hashCode() | 27 |
| 4.4. Callback-##### | 28 |
| 4.5. ##### callback-##### | 29 |
| 4.6. ##### XDOclet ##### | 29 |
| 5. #####-##### | 32 |
| 5.1. ##### | 32 |
| 5.1.1. ##### (Doctype) | 32 |
| 5.1.2. hibernate-mapping | 33 |
| 5.1.3. class | 33 |
| 5.1.4. id | 35 |
| 5.1.4.1. generator | 36 |
| 5.1.4.2. ##### Hi/Lo | 37 |
| 5.1.4.3. ##### UUID | 37 |
| 5.1.4.4. Identity columns and Sequences | 37 |

| | |
|--|-----------|
| 5.1.4.5. Assigned Identifiers | 38 |
| 5.1.5. composite-id | 38 |
| 5.1.6. discriminator | 39 |
| 5.1.7. version (optional) | 39 |
| 5.1.8. timestamp (optional) | 40 |
| 5.1.9. property | 40 |
| 5.1.10. many-to-one | 41 |
| 5.1.11. one-to-one | 42 |
| 5.1.12. component, dynamic-component | 43 |
| 5.1.13. subclass | 44 |
| 5.1.14. joined-subclass | 44 |
| 5.1.15. map, set, list, bag | 45 |
| 5.1.16. import | 45 |
| 5.2. Hibernate Types | 46 |
| 5.2.1. Entities and values | 46 |
| 5.2.2. Basic value types | 46 |
| 5.2.3. Persistent enum types | 47 |
| 5.2.4. Custom value types | 48 |
| 5.2.5. Any type mappings | 48 |
| 5.3. SQL quoted identifiers | 49 |
| 5.4. Modular mapping files | 49 |
| 6. Collection Mapping | 50 |
| 6.1. Persistent Collections | 50 |
| 6.2. Mapping a Collection | 51 |
| 6.3. Collections of Values and Many-To-Many Associations | 52 |
| 6.4. One-To-Many Associations | 53 |
| 6.5. Lazy Initialization | 54 |
| 6.6. Sorted Collections | 55 |
| 6.7. Using an <idbag> | 56 |
| 6.8. Bidirectional Associations | 57 |
| 6.9. Ternary Associations | 58 |
| 6.10. Heterogeneous Associations | 58 |
| 6.11. Collection examples | 58 |
| 7. ##### | 61 |
| 7.1. ##### | 61 |
| 7.2. ##### | 62 |
| 7.3. ##### Map | 63 |
| 7.4. ##### | 63 |
| 7.5. ##### | 65 |
| 8. ##### | 66 |
| 8.1. ##### | 66 |
| 8.2. ##### | 68 |
| 9. Manipulating Persistent Data | 70 |
| 9.1. Creating a persistent object | 70 |
| 9.2. Loading an object | 70 |
| 9.3. Querying | 71 |
| 9.3.1. Scalar queries | 72 |
| 9.3.2. The Query interface | 73 |
| 9.3.3. Scrollable iteration | 74 |
| 9.3.4. Filtering collections | 74 |
| 9.3.5. Criteria queries | 75 |
| 9.3.6. Queries in native SQL | 75 |

| | |
|---|------------|
| 9.4. Updating objects | 75 |
| 9.4.1. Updating in the same Session | 75 |
| 9.4.2. Updating detached objects | 75 |
| 9.4.3. Reattaching detached objects | 77 |
| 9.5. Deleting persistent objects | 77 |
| 9.6. Flush | 77 |
| 9.7. Ending a Session | 78 |
| 9.7.1. Flushing the Session | 78 |
| 9.7.2. Committing the database transaction | 78 |
| 9.7.3. Closing the Session | 79 |
| 9.7.4. Exception handling | 79 |
| 9.8. Lifecycles and object graphs | 80 |
| 9.9. Interceptors | 81 |
| 9.10. Metadata API | 82 |
| 10. ##### # ##### (Transactions And Concurrency) | 83 |
| 10.1. #####, c##### (Sessions) # ##### (Factories) | 83 |
| 10.2. ##### # ##### (Threads and connections) | 83 |
| 10.3. ##### ##### | 84 |
| 10.4. ##### ##### | 84 |
| 10.4.1. ##### # ##### | 84 |
| 10.4.2. ##### # ##### | 85 |
| 10.4.3. ##### ## ##### | 85 |
| 10.5. ##### | 86 |
| 10.6. ##### | 87 |
| 11. HQL: #### ##### Hibernate (The Hibernate Query Language) | 89 |
| 11.1. Case Sensitivity | 89 |
| 11.2. The from clause | 89 |
| 11.3. Associations and joins | 89 |
| 11.4. The select clause | 90 |
| 11.5. Aggregate functions | 91 |
| 11.6. Polymorphic queries | 91 |
| 11.7. The where clause | 92 |
| 11.8. Expressions | 93 |
| 11.9. The order by clause | 95 |
| 11.10. The group by clause | 96 |
| 11.11. Subqueries | 96 |
| 11.12. HQL examples | 97 |
| 11.13. Tips & Tricks | 98 |
| 12. ##### # ##### Criteria | 100 |
| 12.1. ##### Criteria | 100 |
| 12.2. ##### | 100 |
| 12.3. ##### | 101 |
| 12.4. ##### | 101 |
| 12.5. ##### ## ##### | 101 |
| 12.6. ##### ## ##### | 102 |
| 13. ##### SQL ##### | 103 |
| 13.1. ##### SQL ##### | 103 |
| 13.2. ##### # ##### | 103 |
| 13.3. ##### SQL ##### | 103 |
| 14. Improving performance | 105 |
| 14.1. Understanding Collection performance | 105 |
| 14.1.1. Taxonomy | 105 |

| | |
|---|------------|
| 14.1.2. Lists, maps and sets are the most efficient collections to update | 105 |
| 14.1.3. Bags and lists are the most efficient inverse collections | 106 |
| 14.1.4. One shot delete | 106 |
| 14.2. Proxies for Lazy Initialization | 107 |
| 14.3. The Second Level Cache | 108 |
| 14.3.1. Cache mappings | 109 |
| 14.3.2. Strategy: read only | 109 |
| 14.3.3. Strategy: read/write | 109 |
| 14.3.4. Strategy: nonstrict read/write | 110 |
| 14.3.5. Strategy: transactional | 110 |
| 14.4. Managing the Session Cache | 110 |
| 14.5. The Query Cache | 111 |
| 15. Toolset Guide | 112 |
| 15.1. Schema Generation | 112 |
| 15.1.1. Customizing the schema | 112 |
| 15.1.2. Running the tool | 113 |
| 15.1.3. Properties | 114 |
| 15.1.4. Using Ant | 114 |
| 15.1.5. Incremental schema updates | 115 |
| 15.1.6. Using Ant for incremental schema updates | 115 |
| 15.2. Code Generation | 115 |
| 15.2.1. The config file (optional) | 116 |
| 15.2.2. The meta attribute | 117 |
| 15.2.3. Basic finder generator | 119 |
| 15.2.4. Velocity based renderer/generator | 119 |
| 15.3. Mapping File Generation | 120 |
| 15.3.1. Running the tool | 121 |
| 16. #####: #####/##### (Parent/Child) | 123 |
| 16.1. A note about collections | 123 |
| 16.2. Bidirectional one-to-many | 123 |
| 16.3. Cascading lifecycle | 124 |
| 16.4. Using cascading update() | 125 |
| 16.5. Conclusion | 127 |
| 17. #####: Weblog-##### | 128 |
| 17.1. ##### (Persistent Classes) | 128 |
| 17.2. Hibernate-##### | 129 |
| 17.3. Hibernate-### | 130 |
| 18. #####: ##### | 135 |
| 18.1. #####/##### (Employer/Employee) | 135 |
| 18.2. #####/##### (Author/Work) | 136 |
| 18.3. #####/#####/##### (Customer/Order/Product) | 138 |
| 19. ##### | 141 |

Hibernate. # ##### ## ## ##### ##### ##### ##### JIRA, ### ##### ##
#####. ##### ## ##### ##### Hibernate, ##### ##
#####.

##, ##### #####, ##### Hibernate #####
JBoss Inc. (c#. <http://www.hibernate.org/SupportTraining/>). Hibernate ##### JBoss Professional
Open Source product suite.


```
<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>
  </ResourceParams>

```

```

<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>

```

#####, ##### # #####, ##### quickstart, # ##### # ##### TOMCAT/
webapp/quickstart. ### ##### # ##### ## ##### (servlets), # ##### ##### #####
http://localhost:8080/quickstart (##### ## ##, ##### # ##### web.xml). ##-
process().

Tomcat ##### DBCP ### ##### # ##### JDBC
Connection ## ##### JNDI ### java:comp/env/jdbc/quickstart. ##### # #####
#####, ##### # ##### Tomcat. #####
JDBC ##### (JDBC driver exception messages), #####-
JDBC ##### # # Hibernate. #####
"Tomcat & JDBC" ##### # Web.

Hibernate ### ##### (bound) #####
JNDI. ##### Hibernate ##### XML-####. #####, ##### Java-
(properties), #####, ## # #####. ##
XML, ##### # # #. ##### XML-
context classpath (WEB-INF/classes), ### ##### 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>

```

```
## ##### SQL #####, ##### SQL # #####, ## Hibernate
##### JDBC ##### (##### JNDI ##, # ##### Tomcat #####).
##### (dialect) #####, #####
##### SQL "#####" (##### ## #). Hibernate ##### ##
##### # ##### # ##### # ##### # ##### # #####
# #####.
```

```
SessionFactory -- ## ##### (single datastore). ##
##### XML-##### # #####, # ##### #-
##### (## ##### # #####) Configuration # SessionFactory.
```

```
##### # hibernate.cfg.xml, ##### Cat.hbm.xml ## ## XML-#####
##### (persistent class) Cat. ##### # #####
Hibernate, ## ##### ## POJO ##### (POJO class) # ##### (## # #-
#####). ##### # ##### # #####. ##### POJO #, #
##### # ##### (mapping metadata) ## ## POJO #####.
```

```
#####
```

```
## ##### "mapping" -- "##### # #####; #####; #####; #-
#####". # ##### # ##### # #####/#####
## ##### (POJO) # ##### (relational) #, #, #, #
## # ##### # ##### # ##### # ##### XML-##### (XML mapping file).
##### #, ##### # ##### # ##### Java # # # #
#####.
```

1.2. ##### (persistent class)

```
##### ## Hibernate ##### ##### -- ## Plain Old Java Objects (POJOs, ##-
## ##### Plain Ordinary Java Objects). POJO # ## JavaBean ##
(properties), ##### # getter- # setter-#####, ##### #
## #####:
```

```
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 ## ##### #####-## #####, ## Java ## # #####
 (##### String, char # Date) ##### #####, ##### ## Java
 Collections Framework. ## ##### # ##### (values), #####
 (collections of values) ## ##### (associations) # ##### (entities). #####
 (property) id ##### ## ##### (primary key) #####, #####-
 ##### # Cat. Hibernate ## ##
 #####, ## ##
 #####.

(persistent
 classes), ## ## ## ## ##.
 ## ##, Hibernate ## ## ## ## ##
 ## ##, ## ## (reflection) # ## ##
 ## ## (runtime class enhancement), ## ## CGLIB. ## ##
 ## ## Hibernate # ## ##, ## ##/#####
 ## ##.

1.3.

Cat.hbm.xml, ##### ## ##/#####
 ##### # ##### (object/relational mapping). #####
 ##### # #####/#####
 #####.

```

<?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>
    </class>

```

```

    <!-- 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>

```

(persistent) #####
 ##### (entities); ##### (components) ##-
 #####). #####
 #####: catA.getId().equals(catB.getId()) #####,
 ##### (database identity). Hibernate ##-
 #####
 ##### (native) ##### database sequence, hi/lo #####-
 #####). # ##-
 #####, ## ##### UUID-#####
 #####
 ##### CAT_ID ##### CAT, #####
 #####).

Cat #####/##### # #####. ##### name ## ##### # ##-
 #####
 ##### (SQL DDL #####), ##### Hibernate # #####-
 ##### SchemaExport. #####
 Hibernate ## #####, ## ##, ## # ##### # #####. ##### CAT # ##### # ##:

| ##### | ### | ##### |
|--------|-----------------------|----------|
| 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)

15, *Toolset Guide*, ##### SchemaExport.
 ##### SQL DDL #####,
 ##### (???) (custom column type constraints), #####
 (unique constraints) # #####.

1.4.

Hibernate Session. Session -- ## #####
 ##### (persistence manager), ## ##### # #####
 ## ##### Cat. ## # ##### Session ##
 SessionFactory, Session # Hibernate #####-##### (unit-of-work):

```

SessionFactory sessionFactory =
    new Configuration().configure().buildSessionFactory();

```

SessionFactory #####
 ##### XML-#### (hibernate.cfg.xml). ## #### SessionFactory,
 ## ##### (# #####) ##### #
 Configuration. ##. SessionFactory ##### (immutable), #####-
 ##### # ##### Configuration ##### ## #####
 SessionFactory. ## ## ##### SessionFactory # ##### #
 ##### ## ## #####?

SessionFactory ##### ##, #####, # #-
 ##### load-on-startup ##### (Java Servlet). ## ## ##
 # #####-##### ##, ## ## ##. ##
 ##### ## ## ## # ##### SessionFactory, ## ##
 Singleton. ##### ##: #####
 # SessionFactory.

HibernateUtil:

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

public class HibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            // Create the SessionFactory
            sessionFactory = new Configuration().configure().buildSessionFactory();
        } catch (HibernateException ex) {
            throw new RuntimeException("Configuration problem: " + ex.getMessage(), 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();
    }
}
```

SessionFactory, ##### ##, ##
 ##### ThreadLocal ##, ## Session #####. #####
 ##### ##, ## ## Java-#####
 thread-local #####.

SessionFactory ##### (threadsafe class), #####
 ##### # ##### # ##### (##### Session). #####
 ##### Session, ## #####-##### (non-threadsafe), ## #####-##
 ##### (single unit-of-work) # #####. ##### SessionFactory # #-

```
#####, ### ##### ##### #####, ##### ##### # #### #-
### (### #####) ## ##### ##### ##### # #####.
## ##### Hibernate Transaction API ## ##### #####
#####, # ##### ## JDBC #####. ## ##### # ##### # #####.
### ##### (container-managed transactions, JTA) ## #####-#####.
##### #####, ## ##### ## ##### #####.
```

```
#####  
##### HibernateUtil.currentSession(); #####  
##### ##  
##### # #####  
##### (unit-of-work), ##-  
## # #### (servlet filter), ## #### HTTP ##### (HTTP  
response). ##### HTTP #####, --  
### "#####" POJO #####: #.#. ### ##### (HTML), #####  
##### Hibernate ##### # ##-  
## #.
```

Hibernate #####. ##### -- ###
Hibernate Query Language (HQL). HQL -- ### # ##### # ##### -
###, ##### SQL:

```

Hibernate #####-#####-##### query by criteria API (##### #
#####), ##### API ##### #####. ## # ##### Hibernate
##### PreparedStatement # ##### (parameter binding) ## ## SQL-
##### # #####. # #####, ## ##### SQL ##### #-
##### ## ##### ## Hibernate Session, ##### JDBC Connection

```

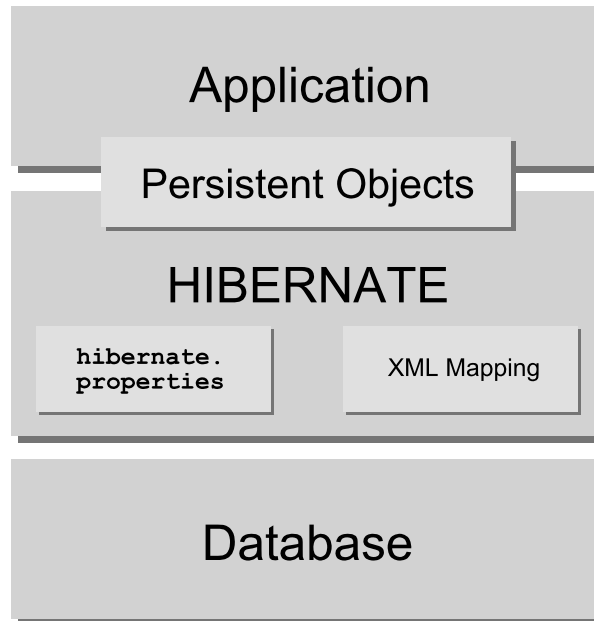
```
# #####  
##### Hibernate. #####  
#####  
##### # #####  
##### # #####  
##### # #####  
##### # #####
```

#####, ### Hibernate, ### ##### # ##### (data access layer), ##### # #####-
#####. ##### ### ##### ## ##### (persistence
mechanism). ##### ## ##### #####.

2.

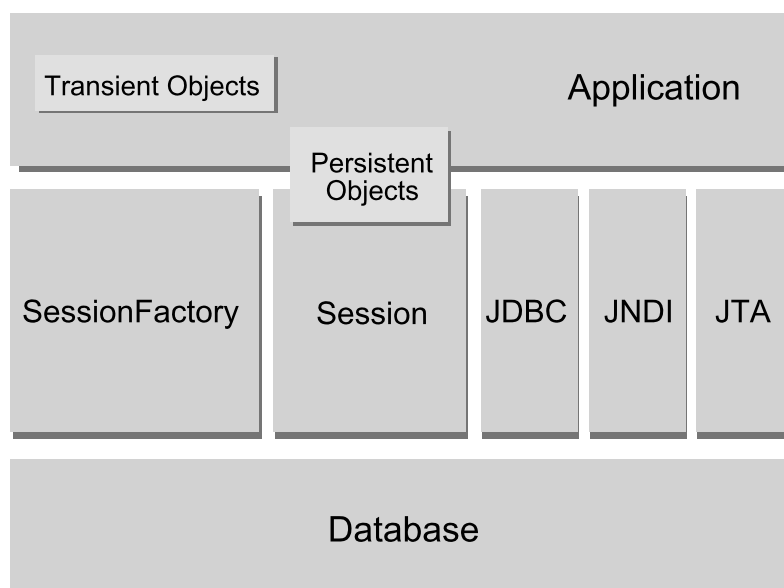
2.1.

(#####) ##### Hibernate:

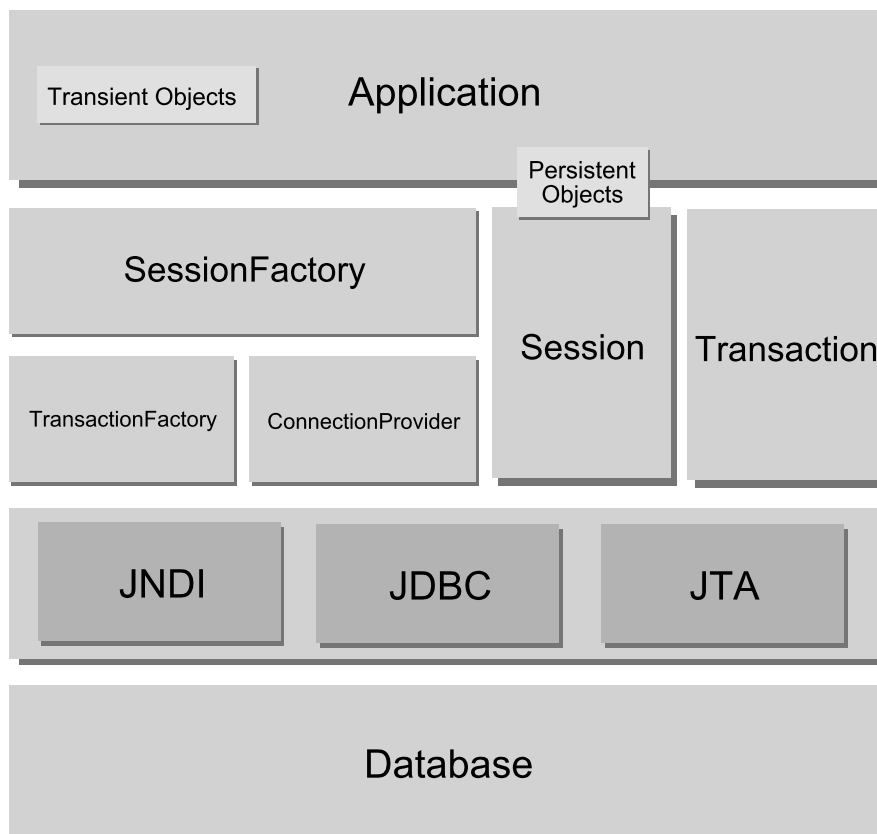


#####, ### Hibernate ##### (Database) # #####
(hibernate.properties, XML mapping) ### #####, #####
(Persistent Objects), ##### (Application).

(runtime architecture).
#. Hibernate ##### # #####
#####, ## #####. "#####"
Hibernate ##### JDBC-##### # #####-
#####. ##### Hibernate API:



"### # #####" ("full cream") ##### ## ##### JDBC/JTA API #
 #### ##### Hibernate ##### ## ## ##.



#####:

SessionFactory (net.sf.hibernate.SessionFactory)

(threadsafe, immutable) #####
 #####. ##### Session (Hibernate).
 SessionFactory ##### ConnectionProvider. SessionFactory #####
 #####, #####, #####, #####
 ##### (second-level cache) ##### JVM (JVM-level cache). #####
 ##### (process-level) #####
 ##### (cluster-level).

Session (net.sf.hibernate.Session)

 ##### JDBC-#####. #####
 Transaction. ##### (first-level cache) #####
 #####.

Persistent Objects and Collections

 ##### JavaBean/POJO #####
 ##### (Session). #####
 ##### (detached) #####
 ##### (layer) #####
 (data transfer objects) #####.

3. ##### SessionFactory

Hibernate ##### # #####, #####
#####. # #, # ##### # # #, #
Hibernate ##### # ##### hibernate.properties, #####
#####. #, # ##### # # classpath.

3.1.

net.sf.hibernate.cfg.Configuration #####
Java ##### # #####. ##### Configuration #####
(immutable) SessionFactory. #####
XML-##### (XML mapping files).

Configuration #####. # #
#####/##### # ##### XML-##### (#
classpath):

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

(mapping file), ##### getResourceAsStream() -- #####
#####):

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

Hibernate # # classpath ##### /
org/hibernate/autcion/Item.hbm.xml # /org/hibernate/autcion/Bid.hbm.xml. #####
#####.

Configuration #####:

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

Configuration ##### (configuration-time): #####
SessionFactory.

3.2. ##### SessionFactory

Configuration ##### (parse) # #####/#####
(factory) # ##### Session. # # #
(shared) #####:

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

#####, Hibernate #### #####
SessionFactory, ### #####, #### ## #####

3.3. JDBC #####,

SessionFactory ##### Session ## JDBC #####, #####-
#####. ### ##### JDBC #####
#####:

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

// do some data access work
```

Session ### ##### JDBC #####, #####-
#####

3.4. JDBC #####, ##### Hibernate

JDBC-#####, ##### SessionFactory. ###
SessionFactory, ### ##### JDBC-##-
#####:

1. ##### java.util.Properties ##### Configuration.setProperties().
2. ##### hibernate.properties # ##### classpath.
3. ##### java -Dproperty=value.
4. ##### XML-##### <property> # ##### hibernate.cfg.xml (#####).

Session ### ## #####:

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

Hibernate ##### # ##### net.sf.hibernate.cfg.Environment. ## ##-
JDBC #####.

Hibernate ##### (# ##### # ##) #####, #####
java.sql.DriverManager, #### ## #####:

3.1. Hibernate JDBC-c##### (Properties)

| ##### | ##### |
|-----------------------------------|------------------|
| hibernate.connection.driver_class | ##### JDBC-##### |
| hibernate.connection.url | JDBC URL |
| hibernate.connection.username | ### ##### |
| hibernate.connection.password | ##### |
| hibernate.connection.pool_size | ##### # ##### |

JDBC-##### ##### ## JNDI-##### #####, ##### ##### ##### ##### # ####-##### (container-managed transactions) ##### #####.

JDBC-##### ##### ##### ##### #####
 "hibernate.connection" # #####. #####, ## ##### charSet, #####
 ### hibernate.connection.charset.

#####, ##### (plugin) ##### JDBC-#####,
 ##### net.sf.hibernate.connection.ConnectionProvider. #####
 ##### net.sf.hibernate.connection.ConnectionProvider #####
 hibernate.connection.provider_class.

3.5.

Hibernate ## #####
 ##### (runtime). ## #####.

java -Dproperty=value ## ##-
 ##### # hibernate.properties, ## # ## Properties ##### # Configuration.

3.3. Hibernate Configuration Properties

| ### ##### | ##### |
|--------------------------------|--|
| hibernate.dialect | <p>### ##### Hibernate Dialect -- ##### ##### #####.</p> <p>##### full.classname.of.Dialect</p> |
| hibernate.default_schema | <p># ##### SQL ##### ## #####.</p> <p>##### SCHEMA_NAME</p> |
| hibernate.session_factory_name | <p>##### SessionFactory, ##### ##### # ##### # JNDI ##### ##, ## #####.</p> <p>##### jndi/composite/name</p> |
| hibernate.use_outer_join | <p>##### ##### (outer join fetching). ##- ##### (deprecated), ##### max_fetch_depth.</p> <p>##### true false</p> |
| hibernate.max_fetch_depth | <p>##### "#####" ##### ##### (one-to-one, many-to-one). 0 ##### ##### # ##### ##### (## #####). Set a</p> |

| ### ##### | ##### |
|--|--|
| | <p>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.</p> <p>##### ##### 0 # 3</p> |
| hibernate.jdbc.fetch_size | <p>##### ##### JDBC fetch size (##### Statement.setFetchSize()).</p> |
| hibernate.jdbc.batch_size | <p>## ##### ##### Hibernate #####-##### JDBC2 ##### (JDBC2 batch updates).</p> <p>##### ##### 5 # 30</p> |
| hibernate.jdbc.use_scrollable_resultset | <p>#### ##### Hibernate ##### JDBC2 scrollable resultsets. ### ##### # #####, ##### -##### JDBC-#####, ##### Hibernate ##-##### (connection metadata).</p> <p>##### true false</p> |
| hibernate.jdbc.use_streams_for_binary | <p>##### ##### /##### ##### ### ##### #/## JDBC. ### ##### (system-level property).</p> <p>##### true false</p> |
| hibernate.jdbc.use_get_generated_keys | <p>##### JDBC3 ##### PreparedStatement.getGeneratedKeys() ### ##, ##### (natively generated keys) ##### (SQL insert). ##### JDBC3+ ##### # JRE1.4+. ##### # false, ##### # ##### # ##### ##### Hibernate. ## ##### Hibernate ##### -##### #####.</p> <p>##### true false</p> |
| hibernate.cglib.use_reflection_optimizer | <p>##### CGLIB ##### -##### runtime reflection (##### #####, ## #####, ### ##, #####-##### CGLIB). ##### reflection ##### #####.</p> <p>##### true false</p> |
| hibernate.jndi.<propertyName> | <p>##### ##### propertyName # JNDI InitialContextFactory.</p> |

| ### ##### | ##### |
|-------------------------------|--|
| | (JVM-level cache = second-level cache) ##### classname.of.TransactionManagerLookup |
| hibernate.query.substitutions | #####/##### ## ##### Hibernate ##- ##### # SQL ##### (#####, ##### ##### #### ##### ##### ##### (function or literal names)). #####/##### ##### (tokens) ## ##### Hibernate # ##### SQL (##### ##### ##### ##### ### #####). ##### hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC |
| hibernate.show_sql | ##### ## SQL-##### ## #####. ##### true false |
| hibernate.hbm2ddl.auto | ##### ##### DDL-##### # ##- ## #####, ##### ##### SessionFactory. # ##### ##### create-drop, ##### ##- ## ##### #####, ##### SessionFactory ##### ##### (# #####). ##### update create create-drop |

3.5.1. ##### SQL

```
## ##### ##### ##### hibernate.dialect # ## #####, ##### ##### ##-
##### net.sf.hibernate.dialect.Dialect ## ##### ## #####. ## ## ## #####
#####, ##### ## ## ##### ##### ##### #####
native ## sequence, # ## ##### (pessimistic locking), ##### ##-
##### #: Session.lock() ## Query.setLockMode(). #####, ##### ## #####
####, Hibernate ##### ##### #####, #####, #####
## ## ##### ## #####.
```

3.4. ##### SQL, ##### Hibernate (hibernate.dialect)

| ##### | ##### |
|----------------|---|
| DB2 | net.sf.hibernate.dialect.DB2Dialect |
| MySQL | net.sf.hibernate.dialect.MySQLDialect |
| SAP DB | net.sf.hibernate.dialect.SAPDBDialect |
| Oracle (#####) | net.sf.hibernate.dialect.OracleDialect |
| Oracle 9 | net.sf.hibernate.dialect.Oracle9Dialect |
| Sybase | net.sf.hibernate.dialect.SybaseDialect |

| ##### | ##### |
|----------------------|--|
| Sybase Anywhere | net.sf.hibernate.dialect.SybaseAnywhereDialect |
| 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 |
| PostgreSQL | net.sf.hibernate.dialect.PostgreSQLDialect |
| HypersonicSQL | net.sf.hibernate.dialect.HSQLDialect |
| Microsoft SQL Server | net.sf.hibernate.dialect.SQLServerDialect |
| Ingres | net.sf.hibernate.dialect.IngresDialect |
| Informix | net.sf.hibernate.dialect.InformixDialect |
| FrontBase | net.sf.hibernate.dialect.FrontbaseDialect |

3.5.2. ##### # ##### (outer join)

(outer joins) # ANSI ## Oracle, ####-
 ## # (outer join fetching) #####
 #####, #####
 ##### (#####,
 ##### #. #####
 ##### # ##-
 #####, #####
 ##### many-to-one, one-to-many ## one-to-one, ##### SQL SELECT ##-
 #####.

"#####". ##, # "#####"
 "#####" #, #####, ##### # "#####"
 ## #.

##, ##### (# ##
 #####) outer-join # XML-##### (XML mapping).

#####, #####
 ##### hibernate.max_fetch_depth # 0. ##### # 1 ##
 ##### one-to-one # many-to-one, # ##-
 ## outer-join # auto. ##, one-to-many
 ##, #####
 ##### outer-join # true. #####
 Hibernate.

 ## (EAGER FETCH>), # HQL, ##### LEFT JOIN FECH, # Criteria API --
 Criteria.setFetchMode().

3.5.3.

Oracle ##### byte-#####, ##### ## JDBC- #####.

binary ### serializable, ### #####
hibernate.jdbc.use_streams_for_binary. ### ##### JVM.

3.5.4. ##### ##-#####. CacheProvider

##-##### JVM, ## ## ##### ## #####
(second-level cache) ##### ## ##### ##### ##, #####
net.sf.hibernate.cache.CacheProvider. ## ##### #####, #####
hibernate.cache.provider_class.

3.5.5. Transaction strategy configuration

Hibernate Transaction API, ## ##### (factory
class) ### ##### Transaction, ##### hibernate.transaction.factory_class.
Hibernate Transaction API ##### # ##### Hibernate ##-
(container managed), ## # # #####.

(first-level cache) -- ### Hibernate Session.

2 ##### (#####) #####:

net.sf.hibernate.transaction.JDBCTransactionFactory

JDBC-#####, ##### (##
#####)

net.sf.hibernate.transaction.JTATransactionFactory

JTA (#### ## #####, ##### Session #####
#####, #####).

(##### ## #####-
CORBA).

####, # JTA #####, ## ##### JVM ## #####
(mutable data), ## ##### JTA,
TransactionManager, ## ## ## ## J2EE #####:

3.5. ##### JTA

| ##### (Transaction Factory) | ##### |
|--|-----------|
| 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 |

3.5.6. ##### # JNDI (JNDI-bound SessionFactory)

SessionFactory, ##### # JNDI, ##### ##### # ##### #-
(##### Session).

SessionFactory # JNDI-##### ##, ##### ##
(#####,
java:comp/env/hibernate/SessionFactory), #####
hibernate.session_factory_name. #### ## ##### ## #####, SessionFactory ## ##### #
JNDI. (## ##### # ##### # ##### JNDI #####-##-#####, ##### Tomcat.)

SessionFactory # JNDI, Hibernate #####
hibernate.jndi.url, hibernate.jndi.class ### #####
(initial context). #### ##### ## #####, ## ##### InitialContext ## #####.

JNDI, ## EJB ## #####-##### ##, #####
SessionFactory ##### ## JNDI (JNDI lookup).

3.5.7.

hibernate.query.substitutions, ## ##### (tokens) ##
Hibernate. #####:

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

true # false, # SQL-#####, #####
(integer literals).

```
hibernate.query.substitutions toLowerCase=LOWER
```

SQL-##### LOWER.

3.6. ##### (##### #, logging)

Hibernate #####, ##### Apache commons-logging.

commons-loggins ##### # Apache Log4j (#### #
log4j.jar # ## classpath), #### # ##### JDK1.4 (#### #
###). ## ##### Log4j # ##### http://jakarta.apache.org. ##
Log4j ## ##### log4j.properties # ## classpath, ##### #-
Hibernate # ##### src/.

Hibernate. ####
#####-
#####. #####. ##
SQL-##### (hibernate.show_sql), ##
#####

3.7. ##### NamingStrategy (#####)

net.sf.hibernate.cfg.NamingStrategy ##### "#####"

Hibernate #### #####

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

XML-####

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

4. ##### (Persistent Classes)

(Persistent #lasses) - ### #####, ##### #####-#####
(#####, #####, ### # #####). #####, ### ##
#####, ##### # ##### (##### - #####
#####, ##### ### #####, #####, # #####; #####
- ##### # #####, #.#. ##### # JVM).

Hibernate, #####-
POJO (Plain Old Java Object) ##### # #####.

4.1. ##### POJO-#####

#####, ##### #####-
#####.

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

public class Cat {
    private Long id; // #####
    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;
    }
}

```

#####:

4.1.1. ##### #

Cat #####. ##### ORM (Object Relational Model - #####) #####. #####. ## ##, ## #####. ##### # ##-#####. Hibernate ##### # ##### JavaBeans # #####-##### getFoo, isFoo # setFoo.

JavaBeans, ##### # #####. ##### (public) - Hibernate #####, ##### # ##### "## #####", protected ## private.

4.1.2.

Cat ##### (#####). ## #####-##### Hibernate ##### (##### ## public) ## ##, ##### Hibernate ##### # ##### Constructor.newInstance().

4.1.3. #####-##### (#####)

Cat ##### "id". ## #####-##### # #####. #####, # #####-### ##, ## #####-##### ##, # ##### java.lang.String # java.util.Date. ##### ##, ## #####-#####-##### ## ## ## (##. #####, #####).

#####-##### ## ##### # #####. ## ## ##### # ## Hibernate #####, #####-#####-##### ## #####.

##, #####
#####. #

- ##### (##. "#####")
- Session.saveOrUpdate()

persistent-##### # #####
##, ##### ##### (##### null)

4.1.4. ##### ##-final ##### (#####)

Hibernate ##### proxy-#####, ##### ##

#####, ##### persistent-##### set/get #####.

Hibernate ## ##### # ##### # final-#####, ##
proxy-#####, ##
#####.

4.2.

#####. C#####
#####-##### 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. ##### equals() # hashCode()

equals() # hashCode() #####
(#####, #
Set).

#####, ##### Hibernate-#####
##. Hibernate ##### JVM-##### (a == b , ##### equals() ##
#####!

"##### JVM-#####", ## ## # #####
#####, "5", ## #

##, ##### ##
(####. ##.).


```

    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 - #####  
(2) onUpdate - ##### (##### #  
##### Session.update())  
(3) onDelete - #####  
(4) onLoad - #####
```

```
##### onSave(), onDelete() # onUpdate() ##### ##### ##### ##### # #####-
### #####, ### ##### ##### ##### ##### # mapping-#####. #####
onLoad() ##### ##### transient-##### (#.#. #####, ##### ## #####-
##### # ### ## #####, ####. ##.) ## ##### ##### #####. ####
##### ## ##### ##### #####, #.#. Hibernate #####
##### ## onLoad(). ##### onLoad(), onSave() # onUpdate() ##### #####
##### ##### #####.
```

```
##### # ####, ### ##### onUpdate() ##### #####, ##### ##### ##### #
Session.update(),## ## ##### #####.
```

```
##### onSave(),onUpdate()### onDelete()##### true,## ##### ##### ### ####-
##. #### ### ##### ##### ##### ##### ##### CallbackException,## #####
#####,# ##### ##### ##### # #####.
```

```
#### ## ##### (native key generation), ## ##### onSave() #####
##### ##, ### #####.
```

4.5. ##### callback-#####

```
#####  
##,##### # persistent-##### Validatable:
```

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

```
##### ValidationFailure #####-## ## #####  
##### , ##### Validatable ## ##### #  
##### (validate()).
```

```
# ##### ## callback-##### ##### Lifecycle, ##### ##### validate() ## #####.
#####-##### ##### ## ##### validate().
```

4.6. ##### XD Oclet

```
# ##### ## #####, ### ##### ##### ##### # ##### ## #
##### mapping-##### # #####, ##### ## ##### XML #####. ##### ##### Hibernate
##### ##### ## ##### ##### # ##### ## #####, ##### XDoclet
@hibernate.tags. # #### ##### ## ## ##### ##### ##### #.#., ##### ##, ## #####
XDoclet #####. ## ## #####. ## ## ##### ##### ##### Cat # XDoclet #####.
```

```

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

/**
 * @hibernate.class
 * table="CATS"
 */
public class Cat {
    private Long id; // #####
    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;
}
}
```

5. #####-#####

5.1.

XML #####. #####

Java, #####
java-#####, #

Hibernate ##### XML

XDoclet, Middlegen # AndromDA.

#####:

```
<?xml version="1.0" encoding="windows-1251"?>
<!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">
        <!-- ##### -->
    </class>

</hibernate-mapping>
```


Hibernate #

not-null.

5.1.1. ### (Doctype)

XML ##### (doctype).
DTD ##### URL, # hibernate-
x.x.x/src/net/sf/hibernate # # hibernate.jar. Hibernate ##### DTD #
classpath.

- ##### <hibernate-mapping>.
- (6) proxy (#####): ##### ## (#####) #####.
 - (7) dynamic-update (#####, ## ##### false): ##### # true Hibernate ##### UPDATE #####, ## ##### #####.
 - (8) dynamic-insert (#####, ## ##### false): ##### # true Hibernate ##### INSERT #####, ## ##### ##### null.
 - (9) select-before-update (#####, ## ##### false): ##### Hibernate ##### SQL UPDATE #####. # ##### (# #####, ##### ##### # ##### # ##-##### update()), ## ##### # Hibernate ##### SQL SELECT ##### # ##, ## UPDATE #####.
 - (10) polymorphism (#####, ## ##### implicit): ##### (explicit) ## (implicit) #####.
 - (11) where (#####) ##### WHERE SQL #####, ##### #####.
 - (12) persister (#####): ##### ClassPersister.
 - (13) batch-size (#####, ## ##### 1) ##### "##### (batch size)" ## #####.
 - (14) optimistic-lock (#####, ## ##### version): ##### (optimistic locking)
 - (15) lazy (#####): ##### lazy="true" ##### ##-##### ## (##### proxy).

java ##### # #####. #####
<subclass>. ## #####
(static inner class), ## #####
(##### eg. Foo\$Bar)

(immutable classes), # ##### mutable="false" ## #####
#####. ## Hibernate #####

proxy #####
##. ##### Hibernate ##### CGLIB #####, # ##

"#####"

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

persister ##### net.sf.hibernate.persister.EntityPersister

name ## #####, #####, ### ##### ## ##### #####.

unsaved-value #####! ##### ##### ##### ## null, ##
#####.

<composite-id> ### ##### # ##### (legacy) #####
c #####. ## ##### ## ##### ##### # #####
#####.

5.1.4.1. generator

<generator>'# ##### Java ##### ##
#####. ## #####
<param> ## ##### ## #####
##.

```
<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>
```

net.sf.hibernate.id.IdentifierGenerator. ###
#####; #####
##, Hibernate #####
(#####) ###:

increment

long, short ### int, #####
#####.

identity

identity ##### # in DB2, MySQL, MS SQL Server, Sybase # HypersonicSQL. ###
long, short ### int.

sequence

sequence # DB2, PostgreSQL, Oracle, SAP DB, McKoi ### generator # Interbase. ###
long, short ### int.

hilo

hi/lo ##### ##
short ### int, ##### # ##### (## hibernate_unique_key #
next_hi #####), ## hi. ##### hi/lo #####

JTA #####.

seqhilo

hi/lo ## ## long, short ### int, #
(sequence) #####.

uuid.hex

128-##### UUID #####
(##### IP-#####). UUID - ##### # 32

#####.

uuid.string

UUID #####, ##### ## ## ##### #####
16 (#####-##) ANSI #####. ## ##### # PostgreSQL.

native

identity, sequence ## hilo, # ##### ## ##### #####.

assigned

#####. #####, ## ##-
save().

foreign

##, #####. ##### # ##-
<one-to-one> ## ##.

5.1.4.2. ##### Hi/Lo

hilo # seqhilo ##### ## ##### hi/lo, #####
#####. ##### "#####-
###" ##### # ##, ## ##### "hi" #####. #####
(Oracle-style), # ##, ##.

```
<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>
```

hilo # ##### (Connection) #
Hibernate, ## ## #####, ##### Hiberante #####
datasource ##### JTA. Hiberante ##### "hi" ##-
#####. ##### # EJB, ##### session stateless bean
hi/lo.

5.1.4.3. ##### UUID

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 separate class in ##### 7.4, «##### ### #####». 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)
/>
```

- (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.

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 column(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)
/>
```

- (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.

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 primary key value as the `Employee` instance referred 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)
  update="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 INSERTS?
- (4) `update`: Do the mapped columns appear in SQL UPDATES?
- (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 distinguishes 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 equivalent 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 hierarchy 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 equivalent 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 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 reachability - 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 `MonetaryAmount` class is a good candidate for a `CompositeUserType`, even though it could easily be mapped as a component. One motivation for this is abstraction. With a custom type, your mapping documents would be future-proofed against possible changes in your way of representing monetary values.

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 separate mapping documents, directly beneath `hibernate-mapping`. This allows you to extend a class hierarchy 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>

```

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 separate 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, ##### 1.4, «##### # #####», for an example implementation).
- In an application with a separate business tier, the business logic must "prepare" all collections that will be needed by the web tier before returning. This means that the business tier should load all the data and return all the data already initialized to the presentation/web tier that is required for a particular use case. Usually, the application calls `Hibernate.initialize()` for each collection that will be needed in the web tier (this call must occur before the session is closed) or retrieves the collection eagerly using a Hibernate query with a `FETCH` clause.
- You may also attach a previously loaded object to a new `Session` with `update()` or `lock()` before accessing uninitialized 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 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 let's 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
```

7.

```
# Hibernate ##### # ##### # ##### # ##### # ##### # #####, ### ##### #####.
```

7.1.

#####, ##### #####, # ## ###. #####
#####-##### (## ##### # #####
#####). #####, ##### #####:

```
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;
    }
}
```

```
##### Name, ##### #### # ##### Person. ##### #, ## #  
Name ##### # getter / setter # #, ## # # -
```

#####

Hibernate

```

<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" ##### -->
    <property name="initial" />
    <property name="first" />
    <property name="last" />
  </component>
</class>

```

pid, birthday, initial, first # last.

 #####
 #####
 ##### null #####. #####, Hibernate
 ##### (null), #####
 (null). #####

Hibernate (#####
 (many-to-one), #####
 #####. Hibernate #####

<component> ##### <parent>, #####
 #####

```

<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" /> <!-- ##### ## Person -->
    <property name="initial" />
    <property name="first" />
    <property name="last" />
  </component>
</class>

```

7.2.

(#.#. ##### Name). #####
 ##### <element> ## <composite-element>.

```

<set name="someNames" table="some_names" lazy="true">
  <key column="id" />
  <composite-element class="eg.Name"> <!-- ##### "class" ##### -->
    <property name="initial" />
    <property name="first" />
    <property name="last" />
  </composite-element>
</set>

```

#####: ##### Set #####

#####:

- ## ##### java.io.Serializable.
- ## ##### equals() # hashCode(), # #####
#####.

IdentifierGenerator #####. #####
#####.

##. #####
unsaved-value #####, #####
#####.

saveOrUpdate() ##### / ##### (cascade save / delete). #####, ## ##
unsaved-value ##### <version> (## <timestamp>), #####
(transient) #####. # #####, #####
(#####) ##### # ## #####
#####.

(##### # ##### ##, ## # # <component>):

```
<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>
```


#####:

```
<many-to-one name="foo" class="eg.Foo">
<!-- ##### "class" ##### -->
  <column name="foo_string"/>
  <column name="foo_short"/>
  <column name="foo_date"/>
</many-to-one>
```

<column> ##### (multi-column) #####
(custom types). ## ##### column. #####
Foo #####:

```
<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>
```

#####, <one-to-many>, ## #####.

Foo ### #####, ### ##### ##### #####.

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

7.5.

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>
```

<dynamic-component> ##### <component>. #####
 #####, # #####, #####
 #####. (##### ## #####, #####
 DOM #####.)

8.

8.1. ###

Hibernate #####.

- table per class hierarchy
- table per subclass
- table per concrete class (# #####)

#####, ## ##### table-per-concrete class. Hibernate
<subclass> ##### # <joined-subclass> ##### #
<class>

Payment # ##### CreditCardPayment, CashPayment, ChequePayment, ##-
#####. ##### table-per-hierarchy ##### ##:

```
<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>
```


#####: ##### NOT NULL.

table-per-subclass ##### ##:

```
<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>
```

```
</joined-subclass>
</class>
```

#####. ### #####. ##### # ##### ##-
(# ##### # # ##### # ##-
#####).

table-per-subclass # Hibernate ## #####-#####. #####
#####-##### # ##### # ##### table-
per-subclass, #####-##### # #####. #####
Hibernate ##### # #####, ##, #####, ##### # #####
#####.

#####, ##### # Payment #####
<many-to-one>.

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

table-per-concrete-class #####.

```
<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>
```

#####. #####, ### ##### # ##### Payment. ###-
Hibernate'##### (implicit polymorphism). ### ##
##, ### ##### Payment #####.

Payment ##### <any>.

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

meta-type ## #####
Payment.

```

<any name="payment"
    meta-type="PaymentMetaType"
    id-type="long">
    <column name="PAYMENT_TYPE"/> <!-- CREDIT, CASH ### CHEQUE -->
    <column name="PAYMENT_ID"/>
</any>

```

#####, ##### ##### ##### ## ##### #####. #### #####
 ##### # ##### <class> (# Payment ##### # ##), ##### ## ##### ##### # ##### ##-
 ##### ##### ##### ##### table-per-class ### table-per-subclass! (### ##### #####
 ##### ##### ##### ##### # ##### Payment.)

```

<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>

```

#####, ### ## ##### ## Payment ####. #### ##### ##### ## ##### Payment, #####-
 ### from Payment, Hibernate ##### ##### ##### CreditCardPayment (# ### #####-
 ##, ### ## ## ##### ##### Payment), CashPayment # ChequePayment ## ##
 NonelectronicTransaction. 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.

Hiberante #####, ### ##### ##### ## ##### ##### foreign key. #####-
 ##### ##### ## ## ##### # ##### ## foreign key ##### (##### #####
 inverse="true" ### insert="false" update="false"), ## ## ##### ##### ##### ##-
 ### ##### ## ##### foreign keys. ### ##### ## ##:

- ##### #####, ##### ##### ## ## ## foreign key.
- ##### ##### ##### ## ##### ##### ## ## ## ##.

- ##### "#####" (eagerly) ##### ## #####, ##### # ## ## outer join.

table-per-concrete-class ## #####. (##### ## ##### join'##.)

table-per-concrete-class # ##### # Hibernate.

8.1.

| ##### #####- ### | Polymorph ic many- to-one | Polymorph ic one- to-one | Polymorph ic one- to-many | Polymorph ic many- to-many | Polymorph ic load()/get () | Polymorph ic queries | Polymorph ic 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 (##### #####- ####) | <any> | ## #####- ##### | ## #####- ##### | <many-to-any> | use a query | from Payment p | ## #####- ##### |

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) );

}
```

The behaviour of `scroll()` is similar to `iterate()`, except that objects may be initialized selectively by `get(int)`, instead of an entire row being initialized at once.

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 explicitly `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. Lifecycles 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). Furthermore, 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] );
    }
}

```

10. ##### # ##### (Transactions And Concurrency)

Hibernate ## #####. ### #####-#####
(#####). ##### ## #####. ##### # #####
JTA, ##### Session ##### JTA ####-
#####. # ##### Hibernate ##### # JDBC, #####-
#####.

10.1. #####, c##### (Sessions) # ##### (Factories)

SessionFactory ### #####, ##### (threadsafe).
#####. Session

(non-threadsafe). ## #####-##### # #####-
#####. #####, ### Hibernate # #####, #####
SessionFactory #####:

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

service ##### Session, ##### (#####
flush()), ##### (##### commit()), ##### (#####
close()) # # #####. (SessionFactory # ###. ##### # JNDI # #
#####-##### Singleton'#.)

stateless session bean #####. SessionFactory ##### #
setSessionContext(). ##### Session, ##### (flush()) #
(close()). (##### JTA, ##### #
container-managed transactions.)

(Hibernate Transaction) ### ##, ##### commit()
Hibernate Transaction ##### # ##### (## #####-
JTA #####).

##, ## # flush(). #####-
##. ##### # #
Hibernate JDBC #####, ##### #
Hibernate!

#####. ### "#####" ##### #
#####.

10.2. ##### # ##### (Threads and connections)

Hibernate:

- ##### Session ### Transaction ### #-
#####.

```
##### ##### ##### ##### ## ##### ##### # ##### # #### # #####-
#####. ##### #### # ## ## ##### ##### ##### ##### ## #####
##### ##### ##### (##### Session). ##### ##### ## #####
#####:
```

```
##### # #####
    foo.getId().equals( bar.getId() )
```

```
##### # JVM
foo==bar
```

```
### ##### (Session) ### #####. ####, ### #####
##### # ##### # ##### # ##### # ##### # ##### # #####, ###
##### # ##### ##### JVM.
```

```
##### Hibernae # #### ##### ## #####.
##### # #####-#####, ## # ##### # #-
#####. ##### # # ##### # ##### = # #####
#####.
```

**10.4. #####
#####**

#####.##### ##### #### ##### # ##### #
#. # #- # (enterprise) ##### #
#.

-##### ##### # #####
#####. #####-
#####. #####
(#####) # ##### # #####. # #####.

```
#####  
# #####. Hibernate #####  
#####
```

10.4.1. ##### # #####

```
#### ##### Session # ## #####
#####
```

```
// ##### foo ### ##### # #####
session.reconnect();
foo.setProperty("bar");
session.flush();
session.connection().commit();
session.disconnect();
```

```
##### foo ### ### ##### # ##### ### #####. ### # ##### Session #### JDBC ##-
#####,, ## ##### ##### # #####.
```

```
##### ##### ##### ##### ##### ##### ##### # # #####, ##### #####-
##### ##### (## ##### ##### #####), ### ## HttpSession ##### #####-
## ##, ##### ## #####. ## ## ##### (Session) ## ## (#####) ##### #####-
##### ##### # ##### ## #####, ## ##### ##### ##### ##### ##
##### ##### ##### #####/#####. ## ##### ##### ## ## ##### ##
## ## ##### ##### ##### ##### (##### #####).

```

10.4.2. ##### # #####

```
##### # ##### # ##### # ##### # #####. ##### ## ##, #-
## # ## # ##### # ##### # ##### # #####. #####-
##### # ##### (detached) ##### # ##### # #####,
##### # ##### ## ##### Session.update() ### Session.saveOrUpdate().
```

```
// foo- #####, ##### #####
foo.setProperty("bar");
session = factory.openSession();
session.saveOrUpdate(foo);
session.flush();
session.connection().commit();
session.close();
```

```
#####  ### ## ##### lock() ##### update() # ##### LockMode.READ (##### #####
##### # ##### #####) ##### ## ##### # ###, ### ##### ## ### #####.
```

10.4.3. ##### ##

```
#####c##### # ##### (Session), #####
##### # ##### # ##### # #####. #####
##### # ##### # ##### (#####-#,
Hibernate # # # # #). ##### #
#####. ##### # ##### # Entity EJB.
```

```
// foo - ##### # ##### (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();
```

#####, #### # ##### # ##### # ##### ##### ##### # #####, # ## # ##-
#####, ## ##### ##### # ##### # ##### # ##### # ##### # ##-
###.

10.5.

#####-
##, ##### # # ##### # ##### # ##### # ##### # #####. (### #, ##### # ##-
(Session) # ##### # HttpSession.) ### # ##### # ##### # #####
#####

1. ##### (commit) ##### (Transaction) ### JDBC ##### # #####
2. ##### # Session ## JDBC.

#####. ##### Session.disconnect() ##### #
JDBC ##### # ##### # ##### JDBC ##### # ## (unless you provided the
connection).

Session.reconnect() ##### # ##### # ##### # #####. ##### # ##### # ##-
#####, ### # ##### # ##### # ##### # #####, ##### # #, ## # #####
Session.lock() ### # ##### # ##### # ##### # ##### # #####. ### # ## #
(lock) #, ##### # #.

#####:

```
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();
```

#####:

```
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);    // #####, ### foo ## #####.
    bar.getFooTable().put( foo.getName(), foo );
}

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

```

##, ### ##### ##### ##### (Transaction) # ##### Session ##### #####-
 #-#####. ##### ##### ##### ##### ##### ##### # ##### #####. #####
 ##### ## ##### ## ##### ## ##### ## ##### ##### (unit-of-work).

10.6.

 #####. ##### ##### ##### ##### ##### ##### # JDBC ##### # #####-
 ##### ##### ##### ##### #####. ### ## #####, ##### ##### # ##### #####
 ##### ##### ##### ##### #####, ### ##### ##### # ##### #####
 #####.

Hibernate ##### ##### ##### ##### #####, ## ##### ## ##### ##### #
 #####!

LockMode ##### ##### #####, ##### ##### ##### Hibernate.
 #####

- LockMode.WRITE ##### ##### ##### Hiberante ##### ## #####.
- LockMode.UPGRADE ##### ##### ##### ##### ##### ##### SELECT ...
FOR UPDATE ### ## #####, ##### #####.
- LockMode.UPGRADE_NOWAIT ##### ##### ##### ##### ##### ##### SELECT ...
FOR UPDATE NOWAIT # Oracle.
- LockMode.READ ##### ##### ##### Hibernate ##### ##### ## #####
Repeatable Read ### Serializable. ##### ##### #####.
- LockMode.NONE ##### #####. ### ##### # ##### ##-
#####. ##### ##### # ##### update() ##
saveOrUpdate() ### ## # ##### #####.

"#####" ##### ##### ## #####:

- ##### Session.load() # ##### LockMode.
- ##### Session.lock().
- ##### Query.setLockMode().

Session.load() ### ##### # UPGRADE ### UPGRADE_NOWAIT, # ##### ##### ## ##
 ### # ##### ##### # ##### SELECT ... FOR UPDATE. #### load() ###
 ### ##### ##### # ##### # ##### # #####, Hibernate ##-
 ##### lock() ### #####.

```
Session.lock() ##### ##### ##### ##### ##### ##### ##### ##### READ,
UPGRADE ### UPGRADE_NOWAIT. (# ##### UPGRADE ### UPGRADE_NOWAIT, ##### SELECT ... FOR
UPDATE.)
```

```
#### ##### ## ##### ##### ##### #####, Hibernate ##### #####-
### ##### (#####). ### ##### #####-
### #####.
```

11. HQL: ##### Hibernate (The Hibernate Query Language)

Hibernate is equipped with an extremely powerful query language that (quite intentionally) looks very much like SQL. But don't be fooled by the syntax; HQL is fully object-oriented, understanding notions like inheritance, 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 mates of other Cats. 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 Hibernate's main selling points. Here are some example queries very similar to queries that I used on a recent project. Note that most queries you will write are much simpler than these!

The following query returns the order id, number of items and total value of the order for all unpaid orders for a particular customer and given minimum total value, ordering the results by total value. In determining the prices, it uses the current catalog. The resulting SQL query, against the `ORDER`, `ORDER_LINE`, `PRODUCT`, `CATALOG` and `PRICE` tables has four inner joins and an (uncorrelated) subselect.

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog.effectiveDate < sysdate
    and catalog.effectiveDate >= all (
        select cat.effectiveDate
        from Catalog as cat
        where cat.effectiveDate < sysdate
    )
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

What a monster! Actually, in real life, I'm not very keen on subqueries, so my query was really more like this:

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog = :currentCatalog
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

The next query counts the number of payments in each status, excluding all payments in the `AWAITING_APPROVAL` status where the most recent status change was made by the current user. It translates to an SQL query with two inner joins and a correlated subselect against the `PAYMENT`, `PAYMENT_STATUS` and `PAYMENT_STATUS_CHANGE` tables.

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
    join payment.statusChanges as statusChange
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or (
        statusChange.timeStamp = (
            select max(change.timeStamp)
            from PaymentStatusChange change
            where change.payment = payment
        )
        and statusChange.user <> :currentUser
```



```
)
group by status.name, status.sortOrder
order by status.sortOrder
```

If I would have mapped the `statusChanges` collection as a list, instead of a set, the query would have been much simpler to write.

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or payment.statusChanges[ maxIndex(payment.statusChanges) ].user <> :currentUser
group by status.name, status.sortOrder
order by status.sortOrder
```

The next query uses the MS SQL Server `isNull()` function to return all the accounts and unpaid payments for the organization to which the current user belongs. It translates to an SQL query with three inner joins, an outer join and a subselect against the `ACCOUNT`, `PAYMENT`, `PAYMENT_STATUS`, `ACCOUNT_TYPE`, `ORGANIZATION` and `ORG_USER` tables.

```
select account, payment
from Account as account
    left outer join account.payments as payment
where :currentUser in elements(account.holder.users)
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

For some databases, we would need to do away with the (correlated) subselect.

```
select account, payment
from Account as account
    join account.holder.users as user
    left outer join account.payments as payment
where :currentUser = user
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

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();
```

12. ##### # ##### Criteria

Hibernate ##### # ##### API ### #####. #####
API ##### # ##### HQL #####. # #####, ##### #-
(projection) # #####.

12.1. ##### Criteria

net.sf.hibernate.Criteria ##### ## #####. ##-
Criteria ##### Session.

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

12.2.

(criterion) #####
net.sf.hibernate.expression.Criterion. ##### net.sf.hibernate.expression.Expression #####-
Criterion.

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

(expressions)

```
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();
```

(criterion) (##### Expression), ## # #####
SQL #####.

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

{alias} #####.

12.3.

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.

createCriteria()

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

#####, ## createCriteria() ##### Criteria, ##### kittens.

(##. ##).

```
List cats = sess.createCriteria(Cat.class)
    .createAlias("kittens", "kt")
    .createAlias("mate", "mt")
    .add( Expression.eqProperty("kt.name", "mt.name") )
    .list();
```

(createAlias() ## ##### Criteria.)

#####, ## kittens ##### Cat #####
#####! ##### kittens #####
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.

setFetchMode().

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

mate # kittens ##### outer join.

12.6. ##### ##

net.sf.hibernate.expression.Example ##### ## ##### (criterion) ##-
#####.

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .list();
```

##, ##### # #####. ## ##, #####
null ##### ## ## #####.

Example.

```
Example example = Example.create(cat)
    .excludeZeroes() //##### # #####
    .excludeProperty("color") //##### "color"
    .ignoreCase() //##### ## #####
    .enableLike(); //##### like ## #
List results = session.createCriteria(Cat.class)
    .add(example)
    .list();
```

#####.

```
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .createCriteria("mate")
        .add( Example.create( cat.getMate() ) )
    .list();
```

13. ##### SQL

SQL. ## ##### ##
CONNECT # Oracle. ##
SQL/JDBC
Hibernate.

13.1. ##### SQL

SQL ##### ##### ## ## Query #####, ##### ##### ## HQL #####.
##, ## ## SQL ##### ##
Session.createSQLQuery().

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

createSQLQuery():

- ##### SQL #####
- ##### #####
- ##### #####

SQL ##### ## ##### # ##### ##### ##-
(# ##### ##### ## Cat). ## ##### ##### ##### # ##### ##### ##-
(String) ## ##### # ##### ##### Class #####.

13.2. ##### #

{cat.*} ##### # ##### ##### ##### "### #####". ## #####
##, ## ## ##### ##### Hibernate ##### ##### ##-
##-
#####. # ##### ## ##### (Cat) ## ##### (cat_log) ##### ##
#####. ##### ## ##, ### ##### #
where.

```
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();
```

#####: ##### ## ##### ##, ## ##### ##### # ##### #
#####!

13.3. ##### SQL

```
##### SQL ##### ##### ##### # ##### # ##### # ##### # #,
##### HQL #####.
```

```
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>
```

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 efficiently. 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 than 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 completely stupid, so it knows not to do that in the case of an newly-empty collection (if you called `list.clear()`, for example). In this case, Hibernate will issue a single `DELETE` and we are done!

Suppose we add a single element to a collection of size twenty and then remove two elements. Hibernate will issue one `INSERT` statement and two `DELETE` statements (unless the collection is a bag). This is certainly desirable.

However, suppose that we remove eighteen elements, leaving two and then add three 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 inheritance model. If you wish to avoid these problems your persistent classes must each implement an interface that declares its business methods. You should specify these interfaces in the mapping file. eg.

```
<class name="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 always force initialization by calling `cat.getSex()` or `cat.getKittens().size()`, for example. But that is confusing to readers of the code and is not convenient for generic code. The static methods `Hibernate.initialize()` and `Hibernate.isInitialized()` provide the application with a convenient way of working with lazily initialized collections or proxies. `Hibernate.initialize(cat)` will force the initialization of a proxy, `cat`, as long as its `Session` is still open. `Hibernate.initialize(cat.getKittens())` has a similar effect for the collection of kittens.

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`.

14.1. Cache Providers

| Cache | Provider class | Type | Cluster Safe | Query Cache Supported |
|--|--|---|------------------------------|-----------------------|
| Hashtable (not intended for production use) | <code>net.sf.hibernate.cache.HashtableCacheProvider</code> | memory | | yes |
| EHCache | <code>net.sf.ehcache.hibernate.Provider</code> | memory, disk | | yes |
| OSCache | <code>net.sf.hibernate.cache.OSCacheProvider</code> | memory, disk | | yes |
| SwarmCache | <code>net.sf.hibernate.cache.SwarmCacheProvider</code> | clustered (ip multicast) | yes (clustered invalidation) | |
| JBoss TreeCache | <code>net.sf.hibernate.cache.TreeCacheProvider</code> | 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 (preferably?), 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.

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();
```

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 utility 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>
```

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
```

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:

15.3. SchemaExport Connection Properties

| Property Name | Description |
|--|-------------------|
| <code>hibernate.connection.driver_class</code> | jdbc driver class |
| <code>hibernate.connection.url</code> | jdbc url |
| <code>hibernate.connection.username</code> | database user |
| <code>hibernate.connection.password</code> | user password |
| <code>hibernate.dialect</code> | 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
```

15.4. SchemaUpdate Command Line Options

| Option | Description |
|--|--------------------------------------|
| <code>--quiet</code> | don't output the script to stdout |
| <code>--properties=hibernate.properties</code> | 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 separate 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
```

15.5. Code Generator Command Line Options

| Option | Description |
|--|--|
| <code>--output=<i>output_dir</i></code> | root directory for generated code |
| <code>--config=<i>config_file</i></code> | 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-empty-concrete-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;
    }
}

```

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 <code>toString()</code> |
| implement-equals | include a <code>equals()</code> and <code>hashCode()</code> method in this class. |
| use-in-equals | include this property in the <code>equals()</code> and <code>hashCode()</code> method. |
| bound | add <code>ChangeListener</code> support for a property |
| constrained | bound + <code>VetoChangeListener</code> 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 <code>Object</code> . |
| 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 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.

15.7. MapGenerator Command Line Options

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

16. #####: #####/##### (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 only 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 children 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 ##### 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 identifier 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, that's all very well for the case of a generated identifier, but what about assigned identifiers and composite identifiers? This is more difficult, since `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.)

17. #####: Weblog-#####

17.1. ##### (Persistent Classes)

weblog # #####, ##### ##### # #####. #####-##### ##### ##### ##### ##### ##### ##### #####/##### (parent/child), ## ## ##### ##### bag (ordered bag) ##### (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-#####

XML-#####, # #####, #####, #####, #####.

```

<?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-###

 #####, #####
 ### 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;
}

```

```
}  
}
```

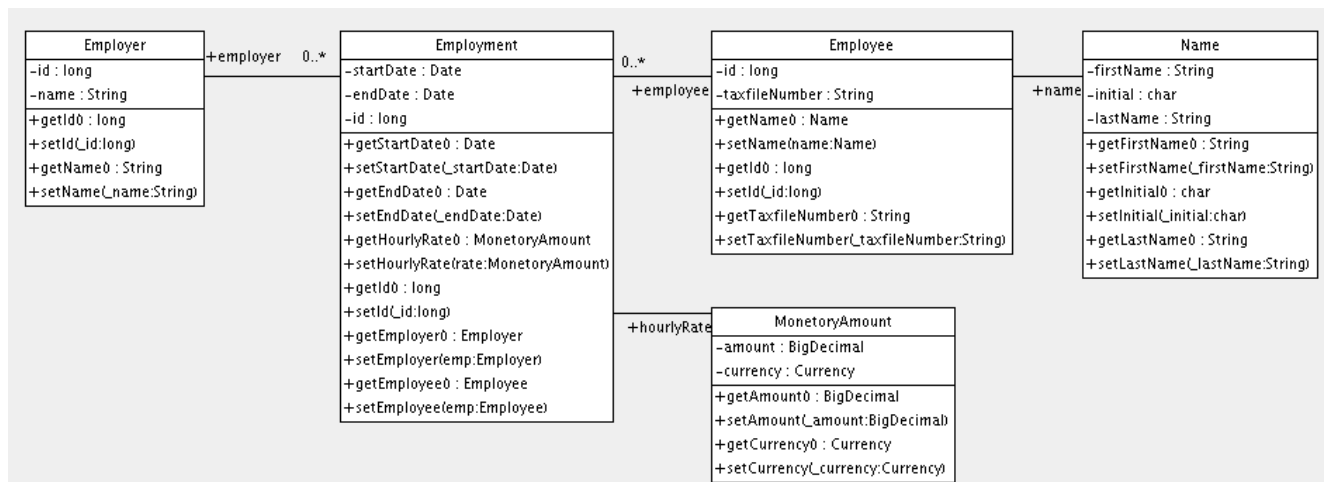
18. #####:

#####

18.1. #####/##### (Employer/Employee)

Employer # ##### Employee ##-
(entity class) Employment, #####. ##-
##, #####

(components), MonetaryAmounts # Name #####.



#####

```
<hibernate-mapping>

<class name="Employer" table="employers">
    <id name="id">
        <generator class="sequence">
            <param name="sequence">employer_id_seq</param>
        </generator>
    </id>
    <property name="name"/>
</class>

<class name="Employment" table="employment_periods">

    <id name="id">
        <generator class="sequence">
            <param name="sequence">employment_id_seq</param>
        </generator>
    </id>
    <property name="startDate" column="start_date"/>
    <property name="endDate" column="end_date"/>

    <component name="hourlyRate" class="MonetaryAmount">
        <property name="amount">
            <column name="hourly_rate" sql-type="NUMERIC(12, 2)"/>
        </property>
        <property name="currency" length="12"/>
    </component>

    <many-to-one name="employer" column="employer_id" not-null="true"/>
    <many-to-one name="employee" column="employee_id" not-null="true"/>

</class>
```

```

<class name="Employee" table="employees">
  <id name="id">
    <generator class="sequence">
      <param name="sequence">employee_id_seq</param>
    </generator>
  </id>
  <property name="taxfileNumber"/>
  <component name="name" class="Name">
    <property name="firstName"/>
    <property name="initial"/>
    <property name="lastName"/>
  </component>
</class>
</hibernate-mapping>

```

#####, ##### 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)

Work #####, ##### Author #
 #####, Person. ## ##### Work # Author, ### #####
 ###-## ##### (many-to-many). ### ##### Author # Person, ##
 #####-# ##### (one-to-one). #####, ### #####
 #####: ##### Author ##### (extend) Person.

```
<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>
</hibernate-mapping>
```



```

        </set>

    </class>

    <class name="Person" table="persons">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>

```

#####. ##### works, authors # persons #####-
 ### # #####, ##### # #####. ##### author_work, #####-###
 ##### (association table), ##### # #####. ### #####, #####
 ### ##### 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)
)

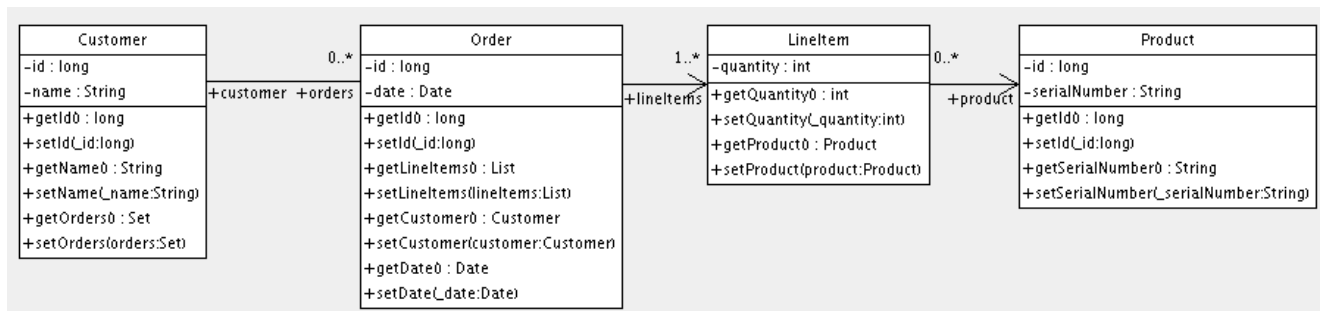
create table persons (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

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)

Customer; ##### Order; # #####-
 ## ##### LineItem, # ##### Product. ##### Customer # Order #####-
 ##### (one-to-many); ## ## ##### Order / LineItem /
 Product? ## ##### LineItem, ### ##### (association
 class), ## ##### (many-to-many) ##### Order #
 Product. # Hibernate ##### # #####
 (composite element), # ##### LineItem -- #####.



#####:

```

<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 # products, #####
##### # #####. ##### line_items, #####
### #####.

```

```

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
```

19.

```
##### "#####" (fine-grained) ##### # ##### ## # ##### #####
##### <component>.
##### ##### ##### ##### ##### #####, #####, ###, ##-
##### ##. ##### ##### # ##### ##### ##### # ##### #####.

### ##### (persistent) #####, #####-#####.
# Hibernate #####-##### #####, ## ##### #####, ## #####-
### ## #####. ## #####, ##### '#####'
(synthetic), #.#. ## ##### ##### ##### # #####; #####, #####
##### #####. ## #####, #####
java.lang.Long ### java.lang.String.

##### ##### # #####.
## ##### #####-#####. ##### com.eg.Foo #
##### com/eg/Foo.hbm.xml. ## ##### # #####.

##### #.
##### #, ## #.

##### #.
### #####, SQL-##### ANSI #####.
##### # #####-#####, #####.

##### (bind variables).
### # # JDBC, ##### " ? ". ##### # #####-
##### #, ## ##### # #####! ### #, #####
##### (named parameters) #.

## ##### JDBC-#####.
Hibernate ##### JDBC-#####, ## ##### #####-
##### # #####. ## # # #####-
## ##### (connections providers), ##### # #####-
##### net.sf.hibernate.connection.ConnectionProvider.

##### (custom types).
##### # # # Java-###, ##### # #####-#####, ##### # #####
##### #, ## # # ##### (accessors) ##### #
##### # # # (component). ## ##### # #####-
## net.sf.hibernate.UserType. ##### # ##### #
##### Java-### # Hibernate-### #.

##### JDBC-### # "#####" #####.
# ##### #, ## #####
##### (#####/#####), ##### #
##### JDBC-#####. ## ##### #
## # "#####" ##### # # #####-
### JDBC-#####. ##### JDBC-#####-
###, ## #####, ## # Hibernate-##### #
JDBC-#####. ##### # #####
#####.
```

```
##### # ##### Session.flush().
##### ## ##### ##### ##### ##### ##### (##### #####
##### ##### # #####) # ##### #####. ##### ##### ##### ##### #####
### ##### ## #####. ##### ## ##### ##### #####
Session.flush() ##### ##### ##### ##### ##### ##### ##-
##### ##### # ##### ##### ##### #####.

# ##### # #####, ##### ##### saveOrUpdate().
### ##### servlet / session bean #####, ## ##### #####
##### # session bean#, # servlet / JSP ####. ##### Hibernate-#####
## #####. ##### Session.update() ### Session.saveOrUpdate() ###
#####.

# ##### # #####, ##### ##### (session
disconnection).
### #####, ##### ##### (Database
Transactions) ##### ## #####. ### ##, ##### #-
##### (long running Application
Transactions), ##### ## ##### (single unit-
of-work). ##### (Application Transaction) ##### #
#####-#####. ##### (Detached Objects), #-
##, # #####, ##### (Session.disconnect()) Hibernate-#####
##### JDBC-##### (Session.reconnect()) ###
#####. ##### Hibernate-#####
##### (Application Transaction usecase); ##### #
##### (stale data), #.#. Hibernate-#####.

## ##### (exceptions) ### (recoverable).
### #####, ### "#####" #####. #####
##### (Transaction.rollback()) # ##### Hibernate-#####
(Session.close()). #####, Hibernate ## #####
##### (#) ##### ##### # #####. ###
## #####, ## Session.load() ### #
#####; #####, ##### find() ## get(). ##### #-
#####, ##### (recoverable), #####, StaleObjectStateException #
ObjectNotFoundException.

### ##### "#####" ##### (lazy fetching).
##### (eager, outer-join) #####. #####
(proxy) ##### /### "#####" ##### #, ##### ## #-
##### JVM (JVM-level cache). ### #, #####
##### (cache hit), #####
##### (eager fetching), ##### outer-join="false". # ##### #
##### outer-join #####, ##### HQL ##### left join fetch.

##### #####-##### ## Hibernate.
##### ## (Hibernate) ###, # ##### #
(data-access code). ##### DAO # Thread Local Session. ##
##### JDBC-###, #####, #####
# Hibernate ##### UserType. (##### "#####")
###; ## ## ##### # #####!)
```

equals() # hashCode(), #####-##### (unique business key).

```
##### ## ##### ##### ## ##### #####, ## ##### ##### equals() # hashCode() ## ##### (shl: # ## ##### ##### ## ## ##-##### # ##### ##### ##### ##### identity, #.#. # ##### ## '=='). ##### ##, ##### (identity) Java-#####. ##### ## #####, ## ##### ##### ##### (### ##### # #####, ##### #). #####, ##### ## ##-##### # ##, ## ##### ## ##### ##, Hibernate ##### ##, #####-##### # ##. ## ##### # java.util.Set'# # #####, ## ##-###-### (hash code) #####, ##### (contract) ## hashCode(). ## equals() # hashCode(), #####-####, #.#. #####-#####. #####, ## ##### # ##### ## ## ## ## ##### # java.util.Set'#, ## ## ##### ## (#.#. #####-#### ## ##### ##). ##### ## ##### equals(), ##-## ##### (lazy loading) # ##### ## # #####, ##### ## ## (proxy).
```

```
## ##### # #####.
##### ##### many-to-many, #####. #####,
##### # "#####" (link table). # #####
#####, ##### ##### one-to-many ## "#####
#####" (link class). ## ##### ## #####, #####-
##### one-to-many # many-to-one. ## ##### #####
#####, ##### ## #####, ##### ##
## ## ##.
```