HIBERNATE

What is Persistence?

- Storing an object beyond the process (program) that created it is called persistence.
- Objects are created in Java program. They are in RAM.
- Objects are to be persisted to tables in relational database, which is on hard disk.

Paradigm Mismatch or Impedance Mismatch

- Object contains data in the form of attributes.
- Objects may be associated with other objects. They hold references to other objects.
- Object may be derived from other objects – inheritance.
- Data in the database is in the form of rows and columns.
- A table references other tables using foreign key.
- There is no support for inheritance in relational model.

How persistence is handled in Java?

- We use JDBC API to convert objects to rows in table.
- To retrieve data from database and to project it as a collection, we need to retrieve rows, convert rows to objects and place them in a collection.
- JDBC needs considerable amount of code.

AddUser.java

```
package jdbc;
import java.sql.Connection;
import java.sql.Date;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
public class AddUser {
public static void main(String[] args) {
        User u = new User();
        u.setUname("Srikanth");
        u.setEmail("srikanthpragada@gmail.com");
        u.setDj( Date.valueOf("2009-7-19"));
        u.setPwd("password");
        Connection con = null;
        PreparedStatement ps = null;
        try {
         Class.forName("oracle.jdbc.driver.OracleDriver");
         con = DriverManager.getConnection
           ("jdbc:oracle:thin:@localhost:1521:xe","hibws","hibws");
         ps = con.prepareStatement
                    ("insert into users values(?,?,?,?)");
         ps.setString(1, u.getUname());
         ps.setString(2, u.getPwd());
         ps.setString(3, u.getEmail());
         ps.setDate(4, u.getDj());
         ps.executeUpdate();
         con.close();
        }
        catch(Exception ex) {
            System.out.println( ex.getMessage());
        }
    }
}
```

What is ORM?

- Stands for Object-Relational Mapping
- Maps objects to relational tables automatically and transparently
- Uses metadata or XML entries to map objects to relational table
- Provides API for performing CRUD (create, read, update and delete)
- Provides a language for queries
- Provides a way to interact with transaction manager, dirty checking, caching and optimizing fetching

Why ORM?

Productivity
Allows you to concentrate on business logic leaving persistence plumbing.

Maintainability
Fewer lines of code, allows code to be more maintainable.

Performance
Provides better performance with better fetching strategy and caching.

Vendor independence
Abstracts application from underlying database differences.

Hibernate as ORM

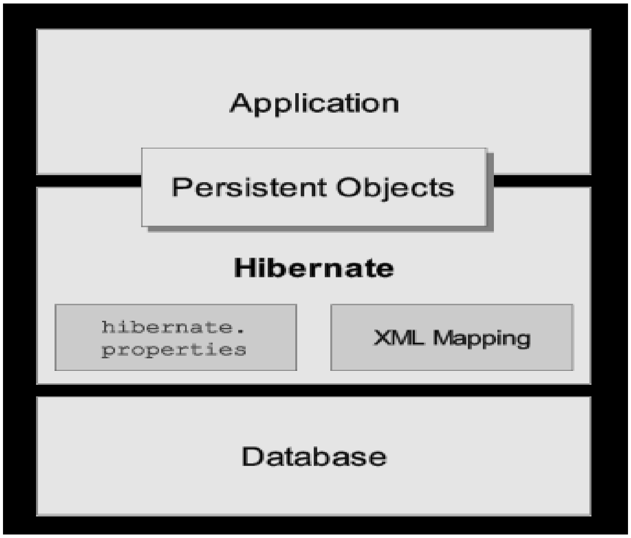
- ❑ Hibernate is Object/relational mapping framework for enabling transparent POJO persistence
- ❑ Developed by Gavin King
- ❑ It is a professional Open Source project and a critical component of the JBoss Enterprise Middleware System (JEMS) suite of products
- ❑ Latest version is 4.3, released in Nov, 2013
- ❑ Available at hibernate.org
- ❑ Hibernate implements JSR 220 (EJB 3.0) and JSR 317 (JPA 2.0)
- ❑ Lets you build persistent objects following common OO programming concepts
    - o Association
    - o Inheritance
    - o Polymorphism
    - o Composition

Why Hibernate?

- ❑ Allows developers focus on domain object modelling, not the persistence plumbing
- ❑ Provides better performance through object caching and configurable fetching strategies.
- ❑ Better integration with Java EE – Hibernate is one of the implementations of JPA.
- ❑ Provides flexible and sophisticated query facilities as
    - o Hibernate Query Language (HQL)
    - o Query By Example (QBE)
    - o Native SQL
    - o Filters
    - o Criteria based query

Note: **NHibernate** is Hibernate for .Net. Its current release is **3.3.3**. It is available at **nhforge.org**.

Hibernate High Level Architecture

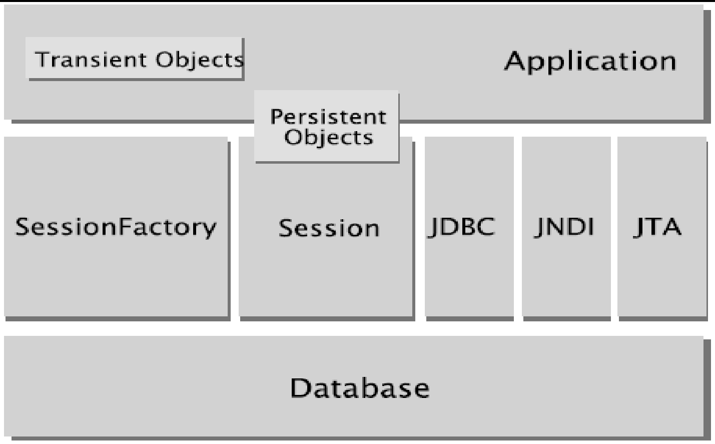


Hibernate Goals

- ❑ Clear cut domain model
- ❑ Domain model should only be concerned about modelling the business process, not persistence, transaction management and authorization
- ❑ Transparent and automated persistence
- ❑ Complete separation of concerns between domain model objects and the persistence mechanism. Persistent solution does not involve writing SQL
- ❑ Metadata in XML
- ❑ Reduction in LOC
- ❑ Importance of domain object model

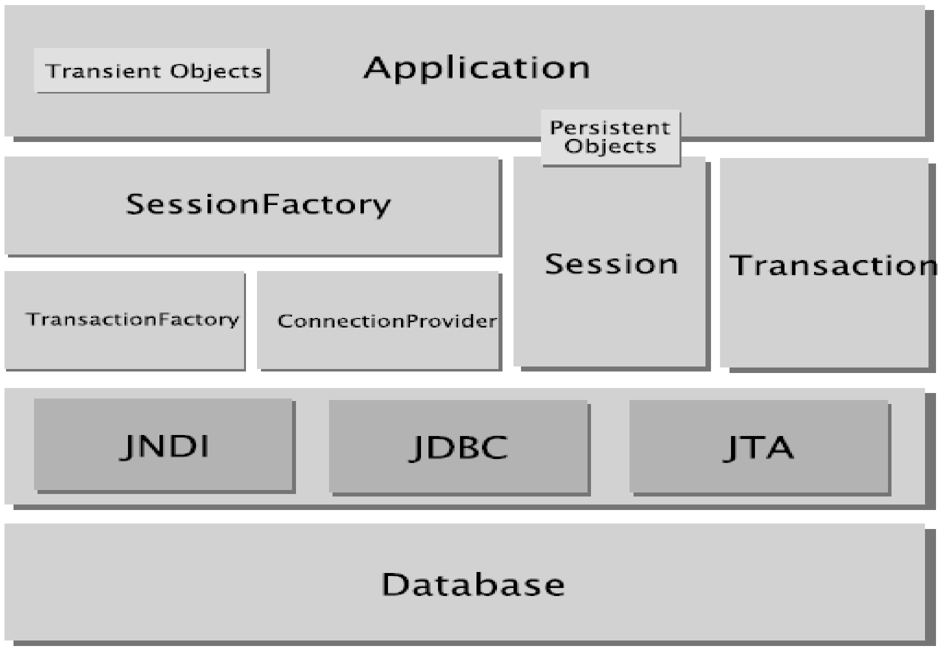
Hibernate Lite Architecture

The "lite" architecture has the application providing its own JDBC connections and manage its own transactions.



Hibernate Full Cream Architecture

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



A simple Hibernate Application

Here are the steps to build a simple Hibernate Application.

- Add required .jar files to your project
- Define the domain model – **Entity** classes
- Setup your Hibernate configuration (**hibernate.cfg.xml**)
- Create the domain object mapping files **<domain_object>.hbm.xml**
- Make Hibernate aware of the mapping files - Update the **hibernate.cfg.xml** with list of mapping files
- Implement **a HibernateUtil** class
- Write our own code

User.java

```
import java.util.Date;
public class User {
  private String uname,pwd,email;
  private Date dj;

    public String getUname() {
        return uname;
    }
    public void setUname(String uname) {
        this.uname = uname;
    }
    public Date getDj() {
        return dj;
    }
    public void setDj(Date dj) {
        this.dj = dj;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getPwd() {
        return pwd;
    }
    public void setPwd(String pwd) {
        this.pwd = pwd;
    }
}
```

hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
   <property name="hibernate.dialect">
           org.hibernate.dialect.Oracle10gDialect</property>
   <property name="hibernate.connection.driver_class">
           oracle.jdbc.OracleDriver</property>
   <property name="hibernate.connection.url">
           jdbc:oracle:thin:@localhost:1521:xe</property>
   <property name="hibernate.connection.username">hibws</property>
   <property name="hibernate.connection.password">hibws</property>
   <property name="hibernate.hbm2ddl.auto">create</property>
   <property name="hibernate.show_sql">true</property>
   <mapping resource="user.hbm.xml"/>
 </session-factory>
</hibernate-configuration>
```

User.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="User" table="USERS">
    <id name="uname">
      <generator class="assigned"/>
    </id>
    <property name="pwd" type="string"/>
    <property name="email" type="string"/>
    <property name="dj" type="timestamp"/>
  </class>
</hibernate-mapping>
```

AddUser.java

```
import java.util.Date;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class AddUser {
    public static void main(String[] args) throws Exception {
        User u = new User();
        u.setUname("Srikanth");
        u.setEmail("srikanthpragada@gmail.com");
        u.setDj( new Date());
```

```
            u.setPwd("password");

            Configuration c = new Configuration().configure();
            SessionFactory sf = c.buildSessionFactory();
            Session session = sf.openSession();
            session.beginTransaction();
            session.save(u);
            session.getTransaction().commit();
            session.close();

            sf.close();
        }
}
```

ListUser.java

```
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class ListUsers {
    public static void main(String[] args) {
        Configuration c = new Configuration().configure();
        SessionFactory sf = c.buildSessionFactory();
        Session session = sf.openSession();
        Query q = session.createQuery("from User");
        List l = q.list();
        for(Object o : l) {
           User u = (User) o;
           System.out.println( u.getUname() + ":" + u.getEmail());
        }
        session.close();
        sf.close();
    }
}
```

Hibernate Framework Objects

- Configuration
- SessionFactory
- Session
- Persistent objects and collections
- ConnectionProvider
- Transaction
- TransactionFactory

Configuration

- ❑ Represents an entire set of mappings of an application's Java types to an SQL database.
- ❑ The Configuration is used to build an (immutable) SessionFactory.
- ❑ Configuration parameters may be passed in any of the following ways:
    - ✓ Pass an instance of java.util.Properties to Configuration.setProperties().
    - ✓ Place hibernate.properties in a root directory of the classpath.
    - ✓ Set System properties using java -Dproperty=value.
    - ✓ Include <property> elements in hibernate.cfg.xml.

Configuration Methods

- ❑ configure(String)
- ❑ configure(URL)
- ❑ configure(File)
- ❑ addFile(String path)
- ❑ addFile(File)
- ❑ addResource(String)
- ❑ addClass(Class)
- ❑ addJar(File)
- ❑ addDirectory(File)

Configuration Using XML file

- ❑ An alternative approach to configuration is to specify a full configuration in a file named hibernate.cfg.xml.
- ❑ This file can be used as a replacement for the hibernate.properties file or, if both are present, to override properties.
- ❑ The XML configuration file is by default expected to be in the root of CLASSPATH.

```
SessionFactory sf = new
                 Configuration().configure().buildSessionFactory();
// You can pick a different XML configuration file using
SessionFactory sf = new Configuration()
                    .configure("catalog.cfg.xml")
                    .buildSessionFactory();
```

Configuration using Properties file

- Properties file **hibernate.properties** can be used to configure hibernate.
- It must be in root of classpath.

hibernate.properties

```
hibernate.dialect=org.hibernate.dialect.Oracle10gDialect
hibernate.connection.driver_class=oracle.jdbc.driver.OracleDriver
hibernate.connection.username=hibws
hibernate.connection.password=hibws
hibernate.connection.url=jdbc:oracle:thin:@localhost:1521:xe
hibernate.connection.pool_size=1
hibernate.show_sql=true
hibernate.hbm2ddl.auto=create
```

```
Configuration c = new Configuration();
c.addResource("Account.hbm.xml");
```

You can specify the mapped class, and let Hibernate find the mapping document for you:

```
Configuration cfg = new Configuration().addClass(catalog.Book.class)
.addClass(catalog.Chapter.class);
```

Hibernate will look for mapping files named **catalog/Book.hbm.xml** and **catalog/Chapter.hbm.xml** in the classpath.

JDBC connection properties

Usually, you want to have the **SessionFactory** to be created and pool JDBC connections. As soon as you do something that requires access to the database, a JDBC connection will be obtained from the pool. For this to work, we need to pass some JDBC connection properties to Hibernate.

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

DataSource properties

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

Property name	**Purpose**
hibernate.connection.datasource	Datasource JNDI name
hibernate.jndi.url	URL of the JNDI provider (optional)
hibernate.jndi.class	Class of the JNDI InitialContextFactory (optional)
hibernate.connection.username	Database user username (optional)
hibernate.connection.password	Database user password (optional)

SQL Dialects

You should always set the **hibernate.dialect** property to the correct org.hibernate.dialect.Dialect subclass for your database.

RDBMS	**Dialect**
DB2	org.hibernate.dialect.DB2Dialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
MySQL	org.hibernate.dialect.MySQLDialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle 9i	org.hibernate.dialect.Oracle9iDialect
Oracle 10g	org.hibernate.dialect.Oracle10gDialect
Sybase	org.hibernate.dialect.SybaseDialect
Microsoft SQL Server	org.hibernate.dialect.SQLServerDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Informix	org.hibernate.dialect.InformixDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Ingres	org.hibernate.dialect.IngresDialect
Progress	org.hibernate.dialect.ProgressDialect

Configuration Files

- Configuration details may be provided either through .xml (hibernate.cfg.xml) file or through properties file (hibernate.properties).
- Default location for configuration is root of application’s classpath.
- XML file entries override entries in properties file.
- You can provide a different xml file to configure method of Configuration class.

Mapping Files

- Provide mapping file in configuration file
- Alternatively add mapping files/classes/jar/directory to configuration object using addFile(), addJar(), addDirectory(), addResource() or addClass() methods
- Name of the mapping file is classname.hbm.xml

SessionFactory

- A threadsafe (immutable) cache of compiled mappings for a single database.
- A factory for Session and a client of ConnectionProvider.
- Might hold an optional (second-level) cache of data that is reusable between transactions, at a process- or cluster-level.
- Hibernate does allow your application to instantiate more than one SessionFactory. This is useful if you are using more than one database.
- Usually, you want to have the SessionFactory create and pool JDBC connections for you.

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

Session

- A single-threaded, short-lived object representing a conversation between the application and the persistent store.
- Wraps a JDBC connection. Factory for Transaction.
- Holds a mandatory (first-level) cache of persistent objects, used when navigating the object graph or looking up objects by identifier.
- It is the persistence context holding the objects that are in persistent state.

Method	Description
Transaction beginTransaction()	Begins a unit of work and return the associated Transaction object.
Connection close()	Ends the session by releasing the JDBC connection and cleaning up.
Criteria createCriteria (Class persistentClass)	Creates a new Criteria instance, for the given entity class, or a superclass of an entity class.
Query createFilter (Object collection, String queryString)	Creates a new instance of Query for the given collection and filter string.
Query createQuery (String queryString)	Creates a new instance of Query for the given HQL string.
SQLQuery createSQLQuery (String queryString)	Creates a new instance of SQLQuery for the given SQL query string.
void delete(Object object)	Removes a persistent instance from the datastore.
void evict(Object object)	Removes this instance from the session.
void flush()	Forces this session to flush.
Object get(Class clazz, Serializable id)	Returns the persistent instance of the given entity class with the given identifier, or null if there is no such persistent instance.
Query getNamedQuery (String queryName)	Obtains an instance of Query for a named query string defined in the mapping file.
SessionFactory getSessionFactory()	Gets the session factory which created this session.
Transaction getTransaction()	Gets the Transaction instance associated with this session.
boolean isDirty()	Does this session contain any changes which must be synchronized with the database? In other words, would any DML operations be executed if we flushed this session?
Object load(Class theClass, Serializable id)	Returns the persistent instance of the given entity class with the given identifier, assuming that the instance exists.
Object merge(Object object)	Copies the state of the given object onto the persistent object with the same identifier.
void persist(Object object)	Makes a transient instance persistent.

void refresh(Object object)	Re-reads the state of the given instance from the underlying database.
Serializable save(Object object)	Persists the given transient instance, first assigning a generated identifier.
void saveOrUpdate (Object object)	Either saves(Object) or updates(Object) the given instance, depending upon resolution of the unsaved-value checks.
void setFlushMode (FlushMode flushMode)	Sets the flush mode for this session.
void update(Object object)	Updates the persistent instance with the identifier of the given detached instance.

Persistent Objects and Collections

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

Transaction

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

Method	**Meaning**
void begin()	Begin a new transaction.
void commit()	Flush the associated Session and end the unit of work (unless we are in FlushMode.NEVER).
boolean isActive()	Is this transaction still active? Again, this only returns information in relation to the local transaction, not the actual underlying transaction.
void rollback()	Force the underlying transaction to roll back.

boolean wasCommitted()	Check if this transaction was successfully committed.
boolean wasRolledBack()	Was this transaction rolled back or set to rollback only?

ConnectionProvider

- ❑ A factory for (and pool of) JDBC connections.
- ❑ Abstracts application from underlying Datasource or DriverManager.
- ❑ Not exposed to application.

TransactionFactory

- ❑ A factory for Transaction instances.
- ❑ Not exposed to the application.

Persistent Classes

- ❑ Must have no-argument constructor.
- ❑ Optionally contains identifier property – primitive type, wrapper classes, String, Date, or user-defined for composite primary key.
- ❑ Prefers non-final classes.
- ❑ By default, Hibernate persists JavaBeans style properties, and recognizes method names of the form getFoo, isFoo and setFoo.
- ❑ Properties need not be declared public - Hibernate can persist a property with a default, protected or private get / set pair.

Basic value types

The built-in *basic mapping types* may be roughly categorized into the following categories.

Data Type	Description
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.
String	A type mapping from java.lang.String to VARCHAR (or Oracle VARCHAR2).
date, time, timestamp	Type mappings from java.util.Date and its subclasses to SQL types DATE, TIME and TIMESTAMP (or equivalent).

calendar, calendar_date	Type mappings from java.util.Calendar to SQL types TIMESTAMP and DATE (or equivalent).
big_decimal, big_integer	Type mappings from java.math.BigDecimal and java.math.BigInteger to NUMERIC (or Oracle NUMBER).
locale, timezone, currency	Type mappings from java.util.Locale, java.util.TimeZone and java.util.Currency to VARCHAR. 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 SQL CLOB or TEXT type.
serializable	Maps serializable Java types to an appropriate SQL binary type. You may also indicate the Hibernate type serializable with the name of a serializable Java class or interface that does not default to a basic type.
clob, blob	Type mappings for the JDBC classes are java.sql.Clob and java.sql.Blob. These types may be inconvenient for some applications, since the blob or clob object may not be reused outside of a transaction. (Furthermore, driver support is patchy and inconsistent.)
imm_date, imm_time, imm_timestamp, imm_calendar, imm_calendar_date, imm_serializable, imm_binary	Type mappings for what are usually considered mutable Java types, where Hibernate makes certain optimizations appropriate only for immutable Java types, and the application treats the object as immutable. For example, you should not call Date.setTime() for an instance mapped as imm_timestamp. To change the value of the property, and have that change made persistent, the application must assign a new (non-identical) object to the property.

Instance States

The following are the different states in which an instance could be.

Transient

- The instance is not, and has never been associated with any session (persistence context)
- It has no persistent identity (primary key value)
- It has no corresponding row in the database

Persistent

- The instance is currently associated with a session (persistence context)
- It has a persistent identity (primary key value) and likely to have a corresponding row in the database
- Changes to objects in this state are automatically saved to the database

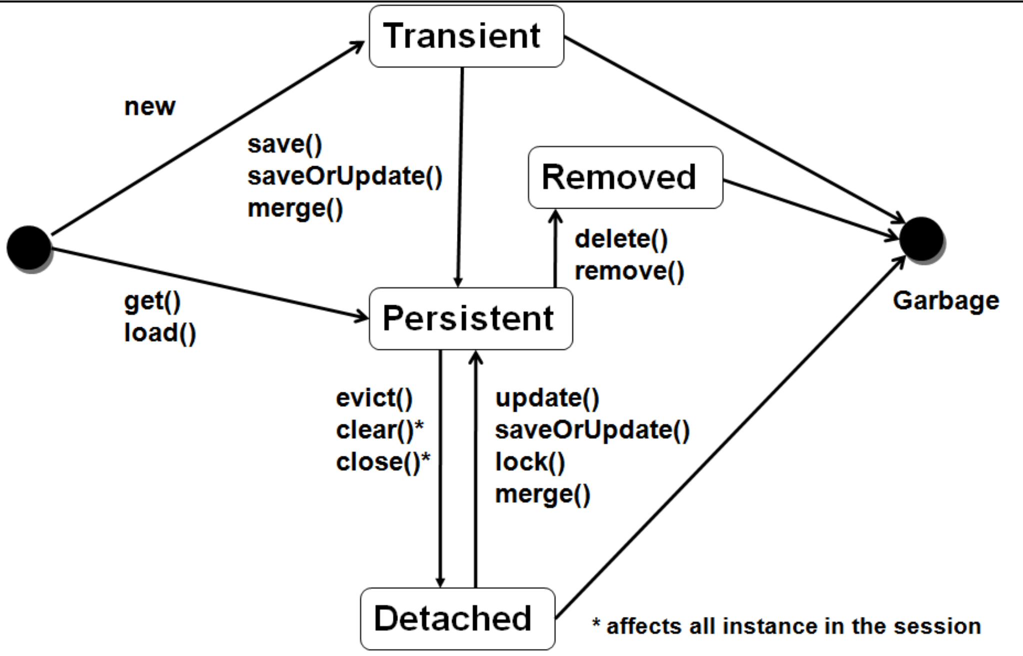
Detached

- The instance was once associated with a persistence context, but that context was closed, or the instance was serialized to another process
- It has a persistent identity and, perhaps, a corresponding row in the database
- No longer managed by Hibernate
- Can be reattached using method like merge(), lock() and update()

Removed

- A previously persistent object that is deleted from the database using session.delete() method.
- Java instance may still exist, but it is ignored by Hibernate.

```
Publisher p = new Publisher(); // Object p is Transient
p.setName("Manning");
p.setEmail("info@manning.com");
p.setAddress("Ny, USA");
SessionFactory sf = HibernateUtil.getSessionFactory();
Session session = sf.openSession();
session.beginTransaction();
session.save(p);    // Object p becomes Persistent
session.getTransaction().commit();  // SQL INSERT
session.beginTransaction();
p.setName("Manning publisher"); // modify object
session.getTransaction().commit();  // SQL UPDATE
session.close();
// object p becomes Detached as session is closed
sf.close();
```

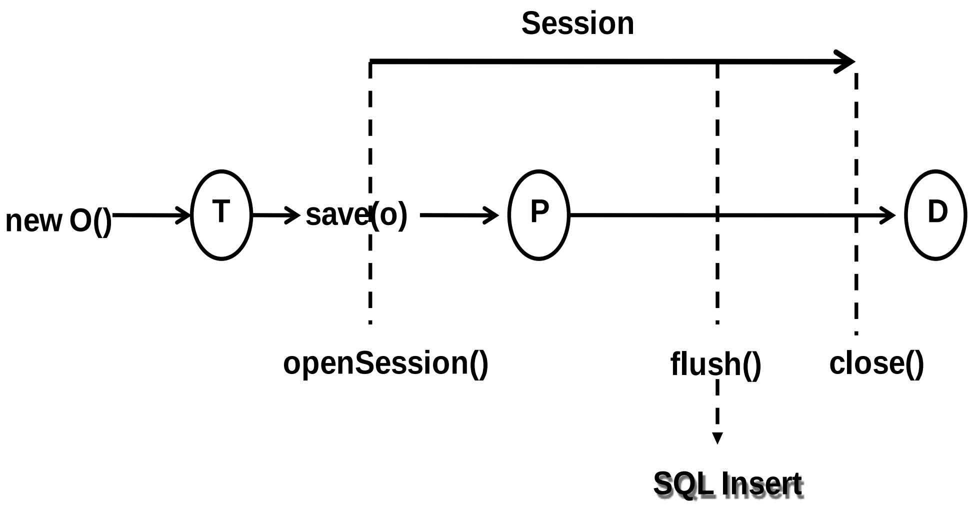


State Transition

- Transient instances may be made persistent by calling save(), persist() or saveOrUpdate()
- Persistent instances may be made removed by calling delete()
- Any instance returned by a get() or load() method is persistent
- Detached instances may be made persistent by calling update(), saveOrUpdate(), lock(), merge()

Difference between load() and get()

- Method load() throws exception if the given id is not found in the database but get() returns null
- Method load() tries to return a proxy where only id is initialized. The rest of the object is initialized when you refer to non-id property.
- Method get() always hits the database and returns an initialized object



Reattaching a detached object

A detached instance can be attached (made persistent) using **update**() method as follows.

```
Session session = sf.openSession();
// Querying object results in SELECT
Publisher p = (Publisher)session.get(Publisher.class, 3);
// p is persistent
session.close();

// p becomes detached
p.setName("Manning Publisher"); // modify detached object
session = sf.openSession();  // create a new session
session.beginTransaction();
session.update(p);//p is shifted to persistent state from detached
session.getTransaction().commit();  // results in UPDATE
session.close();
```

Removing an Object

The following code shows how to delete an object thereby causing corresponding row to be deleted from table.

```
Session session = sf.openSession();
session.beginTransaction();
/*  You can delete even a transient instance because delete()
implicitly makes it persistent before deleting it     */

 Publisher p = (Publisher) session.get(Publisher.class, 1);
 if ( p != null)
     session.delete(p);
 else
     System.out.println("Publisher not found!");

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

Bulk Updates

- ❑ Hibernate provides methods for bulk SQL-style DML statement execution which are performed through the Hibernate Query Language.
- ❑ It is more optimal to run directly in the database (not in memory) as it avoids loading potentially thousands of records into memory to perform the exact same action.
- ❑ Changes made to database records are NOT reflected in any in-memory objects.

```
Transaction tx = session.beginTransaction();
Query q = session.createQuery
       ("update Title set price = price-50 where price > 500");
int cnt = q.executeUpdate();
System.out.println("No. of rows  updated :  " + cnt);
tx.commit();
```

Flushing Context

- ❑ Submits the stored SQL statements to the database
- ❑ Flushing occurs :
    - ✓ When **transaction.commit**() is called
    - ✓ When **session.flush**() is called explicitly
    - ✓ Before a query is executed, if stored statements would affect the results of the query

- You can determine whether changes are to be committed using **isDirty**() method
- Methods **setFlushMode**() and **getFlushMode**() set and get flush mode of the current session.

Flushing Mode options

The following are valid flush modes.

ALWAYS	Every query flushes the session before the query is executed.
AUTO (**default**)	Hibernate manages the query flushing to guarantee that the data returned by the query is up to date.
COMMIT	Hibernate flushes the session on transaction commits.
NEVER	Application needs to manage the session flushing with flush() methods. Hibernate never flushes the session itself.

Generating Id

Hibernate generates unique ID for primary key using any of the following options.

increment	Generates key by adding 1 to current highest key value.
identity	Supports identity columns in DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL.
sequence	Uses a sequence in DB2, and Oracle.
hilo	Uses a hi/lo algorithm to efficiently generate identifiers of type long, short or int, given a table and column (by default hibernate_unique_key and next_hi respectively) as a source of hi values. The hi/lo algorithm generates identifiers that are unique only for a particular database.
seqhilo	Uses a hi/lo algorithm to efficiently generate identifiers of type long, short or int, given a named database sequence.
uuid	Uses a 128-bit UUID algorithm to generate identifiers of type string, unique within a network (the IP address is used). The UUID is encoded as a string of hexadecimal digits of length 32.
guid	Uses a database-generated GUID string on MS SQL Server and MySQL.

native	Picks identity, sequence or hilo depending upon the capabilities of the underlying database.
assigned	Lets the application assign an identifier to the object before save() is called. This is the default strategy if no <generator> element is specified

```
<id name="acno" column="ACNO" type="int">
    <generator class="hilo">
        <param name="table">acno_table</param>
        <param name="column">acno</param>
        <param name="max_lo">1</param>
    </generator>
</id>
```

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

Composite ID Class

- A class represents composite primary key (id).
- Class must override equals() and hashCode() to implement composite identifier equality.
- It must also implement Serializable interface.

```
<class name="Book" table="books">
 <id name="id" column="id" type="int">
       <generator class="assigned" />
 </id>
 <property  name="title"  length="50"  />
 <property  name="price"/>
 <many-to-one column="publisher" class="Publisher"
                                      name="publisher" />
</class>
```

```
<class name="Chapter" table="chapters">
  <composite-id class="ChapterId" name="id">
        <key-property name="bookId"/>
        <key-property name="chapterNo"/>
  </composite-id>
  <property  name="title"  length="50"  />
</class>
```

```
public class ChapterId  implements Serializable {
   private int bookId, chapterNo;
   public ChapterId(int bookId, int chapterNo) {
        this.bookId = bookId;
        this.chapterNo = chapterNo;
   }
   public boolean equals(Object obj) {  // code }
   public int hashCode() {   // code }
}
```

```
// get publisher
Publisher p = (Publisher) session.get(Publisher.class,1);

Book b  = new Book();
b.setId(101);
b.setTitle("Hibernate");
b.setPrice(1000);
b.setPublisher(p);
// create chapters
Chapter c1 = new Chapter();
c1.setId( new ChapterId(101,1));  // construct composite id
c1.setTitle("Getting Started");

Chapter c2 = new Chapter();
c2.setId( new ChapterId(101,2));
c2.setTitle("Working with objects");

// object becomes Persistent
session.save(b);
session.save(c1);
session.save(c2);
```

```
SQL> desc chapters
 Name                               Null?    Type
 --------------------------------- -------- -------------------------
 BOOKID                             NOT NULL NUMBER(10)
 CHAPTERNO                          NOT NULL NUMBER(10)
 TITLE                                       VARCHAR2(50)
SQL> select * from chapters;
   BOOKID  CHAPTERNO TITLE
---------- ---------- --------------------------------------------------
       101          1 Getting Started
       101          2 Working with objects
```

Composition using Component Class

- A component is a contained object that is persisted as a value type, not an entity reference.
- The term "component” refers to the object-oriented notion of composition.

```
public class Address {
   private String dno,street,city;
   // getter and setter methods
}
```

```
public class Customer {
   private int cid;
   private String name;
   private Address address;
   // getter and setter methods
}
```

```
<class name="Customer" table="Customers">
    <id name="cid">
      <generator class="native"/>
    </id>
    <property name="name" type="string"/>
    <component class="Address" name="address">
      <property name="dno"/>
      <property name="street"/>
      <property name="city"/>
    </component>
</class>
```

```
SQL> desc Customers
Name                                               Null?    Type
------------------------------------------------- -------- ------------
 CID                                             NOT NULL NUMBER(10)
 NAME                                                      VARCHAR2(20)
 DNO                                                       VARCHAR2(20)
 STREET                                                    VARCHAR2(20)
 CITY                                                      VARCHAR2(20)
```

```
SQL> select *from customers;
       CID NAME             DNO            STREET               CITY
---------- ---------------- -------------  -------------------- ---------------
         1 Srikanth         50-116-6/3/10  Seethammadhara N.E   Vizag
SQL>
```

Collection Mapping

- Hibernate requires that persistent collection-valued fields be declared as an interface type
- The actual interface might be Set, Collection, List, Map, SortedSet, SortedMap
- Use <set>,<list>, <map>, <bag>, <array> and <primitive-array> mapping elements to represent collection

```
public class Customer {
    private int custid;
    private String name;
    private Set phones = new HashSet();
    // getter and setter methods for custid, name and phones
}
```

```
<class name="Customer" table="Customers">
   <id name="custid">
      <generator class="native"/>
   </id>
   <property name="name"  length="20"  />
   <set name="phones" table="CustomerPhones">
     <key column="custid"/>
     <element column="phonenumber" not-null="true" type="string"/>
   </set>
</class>
```

```
Customer cust = new Customer();
cust.setName("Srikanth Pragada");
cust.getPhones().add("23223232");
cust.getPhones().add("13232322");
session.save(cust);
```

```
SQL> select * from customers;
    CUSTID NAME
---------- --------------------
         1 Srikanth
SQL> select * from customerphones;
    CUSTID PHONENUMBER
---------- --------------------
         1 13232322
         1 23223232
```

Association Mapping

- ❑ Relationship between entity classes can be represented in Hibernate.
- ❑ We must decide the cardinality and direction of the relationship.
- ❑ Relationship between Entities is represented in mapping file.
- ❑ Entities also need to have attributes to represent relationships between entities.

Cardinality or Multiplicity

Multiplicity deals with how many instances could be present on each side of the relationship.

- ❑ one-to-one
- ❑ many-to-one
- ❑ one-to-many
- ❑ many-to-many

Directionality of Relationship

Unidirectional

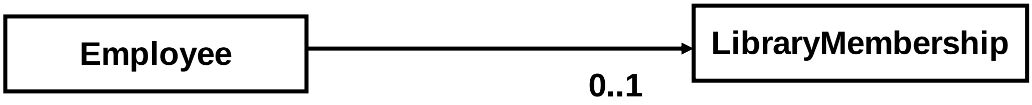
- ❑ Can only traverse objects from one side of the relationship.
- ❑ Given an Account object, we can obtain related transaction objects.
- ❑ Given a Transaction object, we cannot obtain related Account object.

Bidirectional

- ❑ Can traverse objects from both sides of the relationship.
- ❑ Given an Account object, we can obtain related Transaction objects.
- ❑ Given a Transaction object, we can obtain related Account object.

One to One Relationship

- ❑ Expresses a relationship between two entities where each instance of the first entity is related to a single instance of the second or vice versa
- ❑ Can be expressed in the database in two ways
    - ✓ Giving each of the respective tables the same primary key values
    - ✓ Using foreign key constraint from one table onto a unique identifier column of the other



```
public class Employee {
   private int id;
   private String name;
   private LibraryMembership membership;
   // getter and setter methods
}
```

```
public class LibraryMembership {
    private int id;
    private String type;

   // getter and setter methods
}
```

```
<hibernate-mapping>
  <class name="Employee" table="Employees">
    <id name="id">
        <generator class="native"/>
    </id>
    <property length="20" name="name"/>
    <many-to-one cascade="all"
       class="LibraryMembership" name="membership" unique="true"/>
  </class>
  <class name="LibraryMembership" table="LibraryMemberships">
    <id name="id">
      <generator class="native"/>
    </id>
    <property length="20" name="type"/>
  </class>
</hibernate-mapping>
```

```
Employee e = new Employee();
e.setName("Srikanth Pragada");
LibraryMembership lm = new LibraryMembership();
lm.setType("Normal");
e.setMembership(lm);
session.beginTransaction();
session.save(e);
```

```
SQL> select * from employees;
        ID NAME                 MEMBERSHIP
---------- -------------------- ----------
         1 Srikanth                      2
SQL> select * from librarymemberships;
        ID TYPE
---------- --------------------
         2 Normal
```

Bidirectional One-to-One

The following code and mapping show how to represent bidirectional one-to-one association.

```
public class LibraryMembership {
    private int id;
    private String type;
    private Employee employee;
    // other code
}
```

```
public class Employee {
    private int id;
    private String name;
    private LibraryMembership membership;
    // other code
}
```

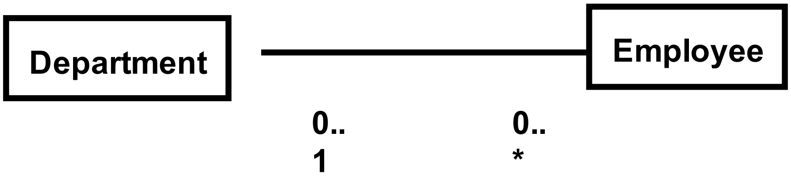
```
<hibernate-mapping>
  <class name="Employee" table="Employees">
    <id name="id">
      <generator class="native"/>
    </id>
    <property name="name"/>
    <many-to-one cascade="all" class="LibraryMembership"
                    name="membership" unique="true"/>
  </class>
  <class name="LibraryMembership" table="LibraryMemberships">
    <id name="id">
      <generator class="native"/>
    </id>
    <property name="type"/>
    <one-to-one name="employee" property-ref="membership"/>
  </class>
</hibernate-mapping>
```

```
Employee e = new Employee();
e.setName("Srikanth");

LibraryMembership lm = new LibraryMembership();
lm.setType("Normal");
e.setMembership(lm);
lm.setEmployee(e);
```

Many To One/One To Many Relationship

- An entity refers to another entity through a reference
- Relationship is defined in many side
- Foreign key is used in many-side table to represent relationship in database
- A collection is used on the one side class to contain references to many objects
- Inverse="true" is given on one side to indicate it doesn't store relationship



```
public class Department {
   private int id;
   private String name;
   private Set<Employee> employees = new HashSet<Employee>();
   // getter and setter methods
}
```

```
public class Employee {
   private int id;
   private String name;
   private Department department;
   // getter and setter methods
}
```

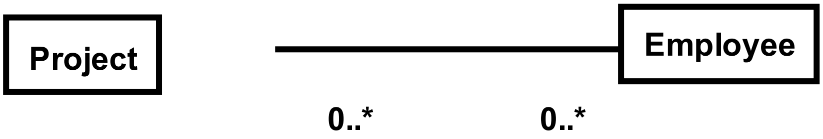
```
<hibernate-mapping>
  <class name="Employee" table="Employees">
    <id name="id">
      <generator class="native"/>
    </id>
    <property length="20" name="name"/>
 <many-to-one cascade="all" class="Department" name="department"/>
  </class>
  <class name="Department"  table="Departments">
    <id name="id">
      <generator class="native"/>
    </id>
    <property length="20" name="name"/>
    <set inverse="true" name="employees">
      <key column="department"/>
      <one-to-many class="Employee"/>
    </set>
  </class>
</hibernate-mapping>
```

```
Employee e = new Employee();
e.setName(“Sergy Brin");
Employee e2 = new Employee();
e2.setName(“Larry Page");
Department d = new Department();
d.setName(“Java");
d.getEmployees().add(e);
d.getEmployees().add(e2);
// maintain relationship on both sides
e.setDepartment(d);
e2.setDepartment(d);
session.save(e);
session.save(e2);
```

```
SQL> select * from department;
        ID NAME
---------- --------------------
         2 Java
SQL> select * from employees;
        ID NAME                 DEPARTMENT
---------- -------------------- ----------
         1 Larry Page                    2
         3 Sergy Brin                    2
```

Many To Many Relationship

- A single employee deals with many projects and a single project has many employees
- On both sides, map the Collection objects
- Either side can be made inverse



```
public class Employee {
   private int id;
   private String name;
   private Set<Project> projects = new HashSet<Project>();
   // getter and setter methods
}
```

```
public class Project {
    private int id;
    private String title;
    private Set<Employee> employees = new HashSet<Employee>();

    // getter and setter methods
}
```

```
<class name="manytomany.Employee" table="Employees">
  <id name="id">
    <generator class="native"/>
  </id>
  <property length="20" name="name"/>
  <set cascade="all" name="projects" table="EmployeeProjects">
    <key column="employeeid"/>
    <many-to-many class="manytomany.Project" column="projectid"/>
  </set>
</class>
<class name="manytomany.Project"  table="Projects">
  <id name="id">
    <generator class="native"/>
  </id>
  <property length="20" name="title"/>
  <set inverse="true" name="employees" table="EmployeeProjects">
```

```
    <key column="projectid"/>
   <many-to-many class="manytomany.Employee" column="employeeid"/>
  </set>
</class>
```

```
Employee e = new Employee();
e.setName("Gavin King");
Employee e2 = new Employee();
e2.setName("Christian Bauer");
Project p = new Project();
p.setTitle("Hibernate");
e.getProjects().add(p);
e2.getProjects().add(p);
p.getEmployees().add(e);
p.getEmployees().add(e2);
session.save(e);
session.save(e2);
```

```
SQL> select * from projects;
        ID TITLE
---------- --------------------
         2 Hibernate
SQL> select * from employees;
        ID NAME
---------- --------------------
         1 Gavin King
         3 Christian Bauer
SQL> select * from employeeprojects;
EMPLOYEEID  PROJECTID
---------- ----------
         1          2
         3          2
```

Cascading

Propagate the persistence action not only to the object submitted, but also to any objects associated with that object.

none

- Default behavior. No cascading is done.

save-update

- Saves or updates associated objects
- Associated objects can be transient or detached

delete

- Deletes associated persistent instances

delete-orphan

- Enables deletion of associated objects when they're removed from a collection
- Enabling this tells Hibernate that the associated class is NOT SHARED, and can therefore be deleted when removed from its associated collection

Implementing Inheritance

Hibernate supports the three basic inheritance mapping strategies:

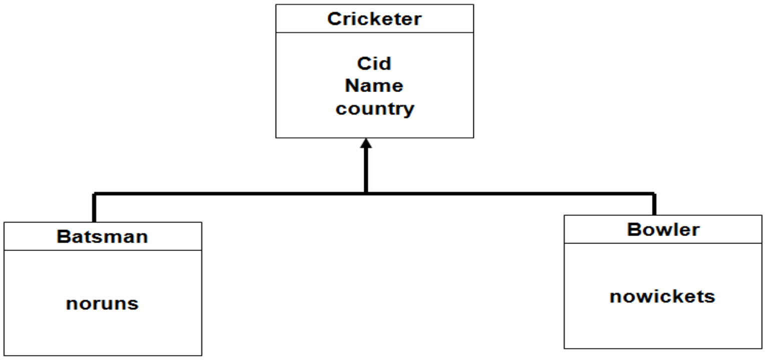
- table per class hierarchy
- table per subclass
- table per concrete class

The following classes are used to demonstrate implementation of inheritance in Hibernate.

```
public abstract  class Cricketer {
   private int cid;
   private String name,country;
   // getter and setter methods
}

public class Batsman extends Cricketer {
   private int noruns;
   // getter and setter methods
}

public class Bowler extends Cricketer {
   private int nowickets;
   // getter and setter methods
}
```



```
Batsman bat = new Batsman();
bat.setCid(1);
bat.setName("Sachin Tendulkar");
bat.setCountry("India");
bat.setNoruns(11222);
session.save(bat);
Bowler bow = new Bowler();
bow.setCid(2);
bow.setName("Anil Kumble");
bow.setCountry("India");
bow.setNowickets(450);
session.save(bow);
```

Table per Class Hierarchy

A single table for the whole class hierarchy. Discriminator column contains key to identify the base type.

Advantages
Offers best performance even for deep hierarchy since single select may suffice.

Disadvantages
Changes to members of the hierarchy require column to be altered, added or removed from the table.

```
<hibernate-mapping>
  <class name="inh.Cricketer" table="ALLCRICKETERS">
    <id column="CID" name="cid" type="int">
      <generator class="native"/>
    </id>
    <discriminator column="PTYPE" length="10" type="string"/>
    <property length="20" name="name"/>
    <property length="20" name="country"/>
    <subclass discriminator-value="batsman" name="inh.Batsman">
      <property name="noruns"/>
    </subclass>
    <subclass discriminator-value="bowler" name="inh.Bowler">
      <property name="nowickets"/>
    </subclass>
  </class>
</hibernate-mapping>
```

```
SQL> select * from allcricketers;
CID  PTYPE     NAME                 COUNTRY     NORUNS       NOWICKETS
---- --------- -------------------- ----------- ------------ ----------
   1 batsman    Sachin Tendulkar     India        11222
   2 bowler     Anil Kumble          India                        450
SQL>
```

Table per subclass

Foreign key relationship exists between common table and subclass tables.

Advantages

- Does not require complex changes to the schema when a single parent class is modified
- Works well with shallow hierarchy

Disadvantages

- Can result in poor performance – as hierarchy grows, the number of joins required to construct a leaf class also grows

```
<hibernate-mapping>
  <class name="inh.Cricketer" table="CRICKETERS">
    <id column="CID" name="cid" type="int">
      <generator class="native"/>
    </id>
    <property length="20" name="name"/>
    <property length="20" name="country"/>
    <joined-subclass name="inh.Bowler" table="Bowlers">
      <key column="cid"/>
      <property name="nowickets"/>
    </joined-subclass>
    <joined-subclass name="inh.Batsman" table="Batsmen">
      <key column="cid"/>
      <property name="noruns"/>
    </joined-subclass>
  </class>
</hibernate-mapping>
```

```
SQL> select * from cricketers;
       CID NAME                 COUNTRY
---------- -------------------- --------------------
         1 Sachin Tendulkar     India
         2 Anil Kumble          India
SQL> select * from bowlers;
       CID  NOWICKETS
---------- ----------
         2        450
SQL> select * from batsmen;
       CID     NORUNS
---------- ----------
         1      11222
```

Table per concrete subclass

Map each of the concrete classes as normal persistent class.

Advantages

- Easiest to implement

Disadvantages

- Data belonging to a parent class is scattered across a number of different tables, which represent concrete classes
- A query of parent class is likely to cause a large number of SELECT operations
- Changes to a parent class can touch large number of tables

```
<hibernate-mapping>
  <class abstract="true" name="inh.Cricketer”>
    <id column="CID" name="cid" type="int">
      <generator class="native"/>
    </id>
    <property length="20" name="name"/>
    <property length="20" name="country"/>
    <union-subclass name="inh.Batsman" table="ONLYBATSMEN">
      <property column="noruns" name="noruns"/>
    </union-subclass>
    <union-subclass name="inh.Bowler" table="ONLYBOWLERS">
      <property column="nowickets" name="nowickets"/>
    </union-subclass>
  </class>
</hibernate-mapping>
```

```
SQL> select * from onlybatsman;
       CID NAME                 COUNTRY                  NORUNS
---------- -------------------- -------------------- ----------
         1 Sachin Tendulkar     India                     11222

SQL> select * from onlybowler;
       CID NAME                 COUNTRY               NOWICKETS
---------- -------------------- -------------------- ----------
         2 Anil Kumble          India                       450
```

Querying In Hibernate

The following are the various options provided by Hibernate for querying the data from data source.

- HQL – Hibernate Query Language
- Named Queries
- Query By Example (QBE)
- Criteria
- Native SQL
- Filters

HQL (Hibernate Query Language)

- Looks very much like SQL.
- HQL is fully object-oriented, understanding notions like inheritance, polymorphism and association.
- Hibernate engine may turn one HQL statement into several SQL statements
- Queries are case-insensitive, except for names of Java classes and properties.
- Bypasses any object caches, such as the persistence context or 2nd Level Cache

HQL Supports the following expressions:

- mathematical operators +, -, *, /
- binary comparison operators =, >=, <=, <>, !=, like
- logical operations and, or, not
- Parentheses ( ), indicating grouping
- in, not in, between, is null, is not null, is empty, is not empty, member of and not member of
- "Simple" case, case ... when ... then ... else ... end, and "searched" case, case when ... then ... else ... end
- string concatenation ...||... or concat(...,...)
- current_date(), current_time(), current_timestamp()
- second(...), minute(...), hour(...), day(...), month(...), year(...),
- Any function or operator defined by EJB-QL 3.0: substring(), trim(), lower(), upper(), length(),
- locate(), abs(), sqrt(), bit_length(), mod()
- coalesce() and nullif()
- str() for converting numeric or temporal values to a readable string
- cast(... as ...), where the second argument is the name of a Hibernate type, and extract(... from ...) if ANSI cast() and extract() is supported by the underlying database the HQL index() function, that applies to aliases of a joined indexed collection
- HQL functions take collection-valued path expressions: size(), minelement(), maxelement(), minindex(), maxindex(), along with the special elements() and indices() functions, which may be quantified using some, all, exists, any, in
- Any database-supported SQL scalar function like sign(), trunc(), rtrim(), sin()
- JDBC-style positional parameters?
- named parameters :name, :start_date, :x1
- SQL literals 'foo', 69, 6.66E+2, '1970-01-01 10:00:01.0'
- Java public static final constants eg.Color.TABBY

Query Interface

- An object-oriented representation of a Hibernate query.
- A particular page of the result set may be selected by calling setMaxResults(), setFirstResult().
- Supports named query parameters (:name) and JDBC style parameters (?).
- You may not mix and match JDBC-style parameters and named parameters in the same query.

- Queries are executed by calling list(), scroll() or iterate().
- A query may be re-executed by subsequent invocations. Its lifespan is, however, bounded by the lifespan of the Session that created it.

Method	**Meaning**
int executeUpdate()	Executes the update or delete statement.
String[]getNamedParameters()	Returns the names of all named parameters of the query.
List list()	Returns the query results as a `List`.
set<Type>(int position, Type value) set<Type>(String name,Type value)	Assigns value to parameter of type <Type>
setComment(String comment)	Adds a comment to the generated SQL.
setFetchSize(int fetchSize)	Sets a fetch size for the underlying JDBC query.
setFirstResult(int firstResult)	Sets the first row to retrieve.
setMaxResults(int maxResults)	Sets the maximum number of rows to retrieve.
setParameter(int position, Object val)	Binds a value to a JDBC-style query parameter.
Object uniqueResult()	Returns a single instance that matches the query, or null if the query returns no results.

```
01: <class name="Title" table="Titles">
02:     <id name="titleid">
03:       <generator class="native"/>
04:     </id>
05:     <property length="50" name="title"/>
06:     <property name="price" type="integer"/>
07:     <many-to-one cascade="all" class="Subject"
08:                           column="subcode" name="subject"/>
09:     <joined-subclass name="ETitle" table="ETITLES">
10:       <key column="titleid"/>
11:       <property name="weburl"/>
12:     </joined-subclass>
13: </class>
14: <class name="Subject" table="Subjects">
15:     <id name="subcode">
16:       <generator class="assigned"/>
```

```
    </id>

    <property name="subname"/>
    <set inverse="true" name="titles">
      <key column="subcode"/>
      <one-to-many class="Title"/>
    </set>
</class>
```

HQL Examples

The following are the example of HQL queries.

- **from** Title
- from Title t where t.price> 500
- **select** title, subject.subname from Title **order by** price desc
- select t.title, s.subname from Title as t **inner join** t.subject as s
- select s.subname, t.title from Title t right join t.subject s
- from Title t where t.subject.subname like 'A%'
- select **max(price)** from Title
- select t.subject.subname, max(t.price) from Title t **group by** t.subject.subname
- select new List(t.title, t.price) from Title t
- select upper(t.title) || ' - ' || upper(t.subject.subname) from Title t
- from Title t inner join t.subject s where t.price > 500 or s.subname like 'A%'
- select c.name from Course c where size(c.subjects) > 2
- select c.name from Course c where 'SQL' in elements (c.subjects)
- select title, price from Title where price > (select avg(price) from Title)f
- from Title t where t.price > :price
- from Title t where t.price > ?

Processing results of a Query - Objects

When you do not use projection, Hibernate returns a collection of objects.

```
List result = session.createQuery("from ETitle").list();
for( Object obj : result) {
    ETitle et = (ETitle) obj;
    System.out.printf("%s : %s :  %d\n",
          et.getTitle(), et.getWeburl(), et.getPrice());
}
```

Processing results of a Query – Fields

When you use projection with select then Hibernate returns an array of objects for each row.

```java
List result = session.createQuery
              ("select title, price from Title").list();
for(Object obj : result) {
    Object [] row = (Object[]) obj;
    System.out.printf("%s : %d\n",
               row[0].toString(), (int)row[1]);
}
```

HQLDemo.java

```java
//Program to take HQL query from user and display the results
package query;
import java.util.List;
import java.util.Scanner;
import org.hibernate.cfg.Configuration;
import org.hibernate.classic.Session;
public class HQLDemo {
    public static void main(String[] args) {
      Configuration c = new Configuration();
      c.configure("hibernate.cfg.xml");
      c.addResource("catalog.hbm.xml");
      Session session = c.buildSessionFactory().openSession();
      Scanner s = new Scanner(System.in);
      String query;
      while ( true){
       System.out.println("Enter Query : ");
       query = s.nextLine();
       if ( query.length() == 0 ) break;
       try {
        List result = session.createQuery(query).list();
        System.out.println("Query Result");
        System.out.println(
         "=====================================================");
        for( Object obj : result) {
          if ( obj instanceof Object[]) {
              for ( Object o : (Object[]) obj) {
                   System.out.print(o.toString() + "  ");
              }
              System.out.println();
          }
          else
           System.out.println( obj.toString());
        }
       }
```

```
        catch(Exception ex){
            System.out.println("Error-->" + ex.getMessage());
        }
     }
   }
}
```

Named Queries

- Define queries in object mapping files
- Query can be 'global' or included inside class definition
- It enables you to modify the query without changing source code of your application
- If inside class definition, need to prefix with fully qualified class name when calling

```
<query name="costlytitles">
    <![CDATA[
                from Title t where t.price > :price
    ]]>
</query>
```

```
List result = session.getNamedQuery("costlytitles")
              .setParameter("price",500)
              .list();
```

Query By Example

- QBE allow you to obtain the result by providing sample data regarding the data you want.
- You populate an instance with the required data and then hibernate creates query with the data provided by you in the instance.
- Example class, which implements Criteria interface, contains the QBE functionality.
- Certain fields ignored by default
  - ✓ Object Identifiers
  - ✓ Version property
  - ✓ Associations
  - ✓ Any null valued properties
- Important methods are :
  - ✓ enableLike(MatchMode mode)
  - ✓ ignoreCase()

- ✓ excludeZeroes() : Excludes zero-valued properties
- ✓ excludeProperty(String name) : Exclude a particular named property
- ✓ excludeNone() : Do not exclude null or zero-valued properties

```
Subject s = new Subject();
s.setSubname("java");
Example subjectex =
    Example.create(s).ignoreCase().enableLike(MatchMode.ANYWHERE);

List result =
    session.createCriteria(Subject.class).add(subjectex).list();
// print result
```

```
Title t= new Title();
t.setTitle("comp");

Example titleex = Example.create(t)
                   .ignoreCase()
                   .enableLike(MatchMode.ANYWHERE)
                   .excludeProperty("price");
result = session.createCriteria(Title.class).add(titleex).list();
```

Query By Criteria

- ❑ Build a query by defining multiple 'Criterion'
- ❑ Allows you to specify constraints without direct string manipulations
- ❑ Criteria object is created from session and contains all the restriction/projection/aggregation/order information for a single query
- ❑ DetachedCriteria is same as Criteria, but created without the presence of a session but later attached to a session and executed
- ❑ Criterion represents a single restriction for a particular query
- ❑ Restrictions is utility class used to create Criterion objects

Methods

- ❑ add(Criterion criterion)
- ❑ addOrder(Order order)
- ❑ List list()
- ❑ Criteria createCriteria (String associationPath)
- ❑ setFetchSize (int fetchSize)
- ❑ setFirstResult (int firstResult)
- ❑ setMaxResults (int maxResults)
- ❑ Object uniqueResult()

```
Criteria ct = session.createCriteria(Title.class)
                .add( Restrictions.between("price",300,500))
                .addOrder(Order.desc("price"));
List lst1 = ct.list();

ct = session.createCriteria(Title.class)
          .add( Restrictions.gt("price",400))
          .createCriteria("subject")
          .add( Restrictions.eq("subcode","java"));
List lst2 = ct.list();
```

Native Query

- You may also express queries in the native SQL dialect of your database.
- Hibernate allows you to specify handwritten SQL (including stored procedures) for all create, update, delete, and load operations.
- Used for very complicated queries or taking advantage of some database features, like hints.

```
List result = session.createSQLQuery
                              ("select * from subjects").list();
List result2 = session.createSQLQuery
                        ("select * from subjects")
                              .addEntity(Subject.class).list();
List result2 = session.createSQLQuery
                 ("select titleid,title from titles").
                    addScalar("TITLEID", Hibernate.INTEGER).
                    addScalar("TITLE", Hibernate.STRING).list();
Query qry = session.createSQLQuery
                  ("select * from titles where title like ?");
qry.setString(0,"Hi%");
result = qry.list();
```

Filters

- A Hibernate filter is a global, named, parameterized filter that may be enabled or disabled for a particular Hibernate session.
- You can attach a filter criteria to a class.
- Application can then make the decision at runtime whether given filters should be enabled and what their parameter values should be.

```
<filter-def name="costlybooks">
```

```
      <filter-param name="cutoffPrice" type="integer" />
</filter-def>
```

Then associate the filter with a class as follows:

```
<class name="query.Title" table="Titles">
   . . .
   <filter name="costlybooks" condition="price > :cutoffPrice" />
</class>
```

In Java code, enable the filter and pass a value for the parameter defined in the filter.

```
Filter costlybooks = session.enableFilter("costlybooks");
costlybooks.setParameter("cutoffPrice", new Integer(500));
List result = session.createQuery("from Title").list();
```

Using Annotations

- ❑ Starting from Hibernate 3, you can use annotations for metadata instead of XML files.
- ❑ The annotations are same as JPA annotations.
- ❑ Hibernate provides additional annotations for its extensions.

```
@Entity
@Table( name="Accounts")
public class Account implements Serializable {
    @Id
    private int acno;
    @Column(name="cname", length=30)
    private String cname;
    private double balance;
    public Account() {
    }
    // provide getter and setter methods for properties
}
```

```
Configuration c = new Configuration();
c.addAnnotatedClass(Account.class);
SessionFactory sf = c.configure().buildSessionFactory();
```

Caching

- Hibernate provides Level-one cache called as L1 cache. This is client side database cache.
- First-level cache is associated with the *Session* object.
- Second-level cache is associated with the *SessionFactory* object.
- All request must pass through L1 cache.
- L1 ensures a request for a given object from database will return the object instance from cache, if exists, thus preventing multiple loads.
- Object from L1 cache can be individually discarded by invoking evict() method of session object.
- Optional level-two cache can be any cache that implements org.hibernate.cache.CacheProvider interface.
- Use hibernate.cache.provider_class property to specify level-two cache (L2).

Hibernate supports these open-source cache implementations out of the box:

- EHCache (org.hibernate.cache.EhCacheProvider)
- OSCache (org.hibernate.cache.OSCacheProvider)
- SwarmCache (org.hibernate.cache.SwarmCacheProvider)
- JBoss TreeCache(org.hibernate.cache.TreeCacheProvider)

```
<hibernate-configuration>
<session-factory>
  ...
  <property name="hibernate.cache.use_query_cache">true</property>
  <property name="hibernate.cache.use_second_level_cache">true
                                                          </property>
  <property name="hibernate.cache.provider_class">
                org.hibernate.cache.EhCacheProvider
  </property>
  ...
</session-factory>
</hibernate-configuration>
```

Fetching

- A fetching strategy is the strategy Hibernate will use for retrieving associated objects if the application needs to navigate the association.
- Fetching strategy will have performance impact.
- Default fetch mode is vulnerable to N+1 selects problem where "N" is number of SELECTs used to retrieve the associated entity or collection.
- Unless you explicitly disable lazy fetching by specifying *lazy="false"*, the subsequent select will only be executed when you actually access the association.
- Fetch strategies may be declared in the mapping files, or over-ridden by a particular HQL or Criteria query.

```
String hql = "from Title t join fetch t.subject as s";
Query query = session.createQuery(hql);
List results = query.list();
```

Concurrency – What is the problem?

- User A retrieves data of product 101 at 10:10:10
- User B retrieve data of product 101 at 10:10:11
- User A and B both modify the product information in the memory
- User B writes changes back at 10:10:13 and commits changes
- User A tries to write changes back at 10:10:14 and commit changes
- At this point there are two options:
  - ✓ Option 1: Allows user A to proceed overwriting changes made by user B
  - ✓ Option 2: Stop user A from making changes as data has changed since he retrieved the data

Optimistic Concurrency

- By default Hibernate doesn't provide any concurrency control – so last commit wins.
- You can use version based optimistic concurrency control in which a number or timestamp is used to detect changes to row in the table.
- Or you can compare the content of the row with content of the object to detect changes in the row since you read it into memory.

Version based concurrency

- Each entity instance has a version, which can be a number or timestamp.
- Hibernate increments the version number whenever it makes changes to object.
- It uses version to detect changes and throw exception

```
public class User {
    private String uname,email;
    private int version;
    public int getVersion() {
        return version;
    }
    // other code
}
```

```
<hibernate-mapping>
  <class dynamic-update="true" name="User"
             optimistic-lock="version" table="USERS">
    <id name="uname">
      <generator class="assigned"/>
    </id>
    <version name="version"/>
    <property name="email" type="string"/>
  </class>
</hibernate-mapping>
```

```
update USERS set version=?, email=? where uname=? and version=?
```

Content based concurrency

In content based optimistic lock, Hibernate checks whether the data in the table is same as the data that was retrieved earlier.

```
public class User {
    private String uname,email;
    // code
}
```

```
<hibernate-mapping>
  <class dynamic-update="true" name="User"
                optimistic-lock="all" table="USERS">
    <id name="uname">
      <generator class="assigned"/>
    </id>
   <property name="email" type="string"/>
  </class>
</hibernate-mapping>
```

```
update USERS set email=? where uname=? and email=?
```

Interceptor

- 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.
- Create an interceptor (class) which either implements Interceptor directly or (better) extend EmptyInterceptor.
- Register the interceptor with session using openSession() method or using setInterceptor() method of Configuration object.

```
package interceptors;
import java.io.Serializable;
import org.hibernate.EmptyInterceptor;
import org.hibernate.type.Type;
public class SaveInterceptor extends EmptyInterceptor {
   @Override
   public boolean onSave(Object entity, Serializable id,
        Object[] state, String[] propertyNames, Type[] types) {
        // display state and propertyNames
        for (Object o : state) {
            System.out.println("State : " + o);
        }
        if (entity instanceof User) {
            User u = (User) entity;
            System.out.println(u.getUname());
            state[0] = "New Password";
            return true;  // apply changes
        }
        return false;  // ignore changes
    }
}
```

Registering an Interceptor

```
SessionFactory sf = c.buildSessionFactory();
Session session = sf.openSession( new SaveInterceptor());
```