Chapter 1 – Introduction to Mapping Objects to Relational Databases – ½ hour

What is Persistence

Possible Solutions

Mapping

Common Mapping Techniques

- Primary Keys, Timestamps and version numbers
- Handling Inheritance
  - Lowest Child Inheritance
  - Table per Class Inheritance
  - Table Per Concrete Class
- Working with Relationships
  - One-to-one, One-to-Many, Many-to-Many

What is Persistence ?

Storing an object for use by same or other applications at a later time is known as persistence

Some of the common solutions to this problem are

1. Serialization

   Isn’t fast, very slow, to be avoided as a mechanism for large persistence.

2. XML

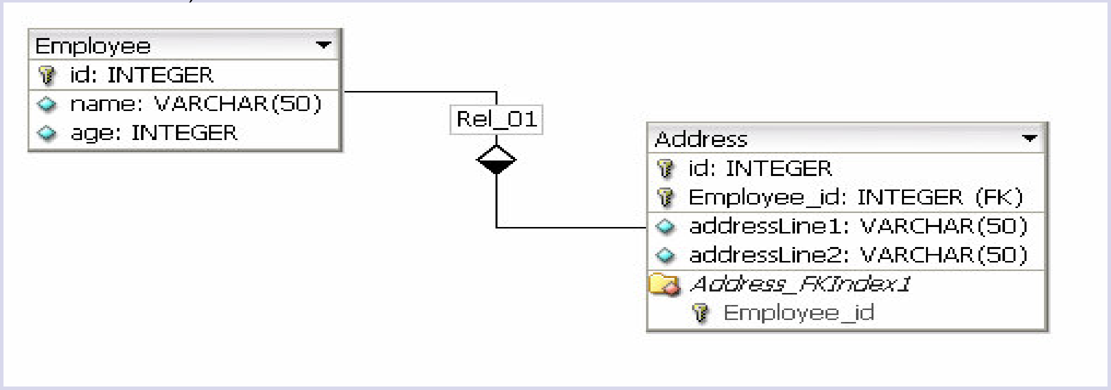
   Additional level of complexity and processing required.

3. Object-oriented database systems mapping

   OO databases haven’t made a large impact in the database market.

Mapping

Mapping is a technique that places an object's attributes in one or more fields of a database.



```
import java.io.Serializable;
import java.util.Set;
public class Employee implements Serializable {

                private String name;
                ….
                private Set addresses;
                public Integer getAge() {
                                return age;
                }
                public void setAge(Integer age) {
                ….
                public Set getAddresses() {
                                return addresses;
                }
                public void setAddresses(Set addresses) {
                                this.addresses = addresses;
                }
```

```
package com.test;

public class Address implements Serializable{

                private Integer id;
                ….
                private Employee empObj;
                public Integer getId() {
                                return id;
                }
                …..
                public void setId(Integer id) {
                                this.id = id;
                }
                public void setEmpObj(Employee empObj) {
                                this.empObj = empObj;
                }

}
```

Haaris Infotech
Driven by Technology

Common Mapping Techniques

Primary keys

The primary key is needed in order for the database server to uniquely distinguish and manage objects stored in the database.

Timestamps

When the persistence layer needs to determine whether an object should be persisted to the database, it can check the timestamp in the table row of the object, if the timestamp is less than the stamp kept in the object, the object should be persisted.

Version Numbers

Another technique is, when the object is pulled from the database, it has an associated version number. If the application changes the object in any way, the persistence layer updates the version number by a single digit.

Common Mapping Techniques

Handling Inheritance

Lowest Child Inheritance mapping

Table-per Class Inheritance mapping

Table Per Concrete Class Inheritance mapping

Relationships

One – to – One

One – to – Many

Many – to – many

Chapter 2- Introduction to Hibernate

Hibernate ?

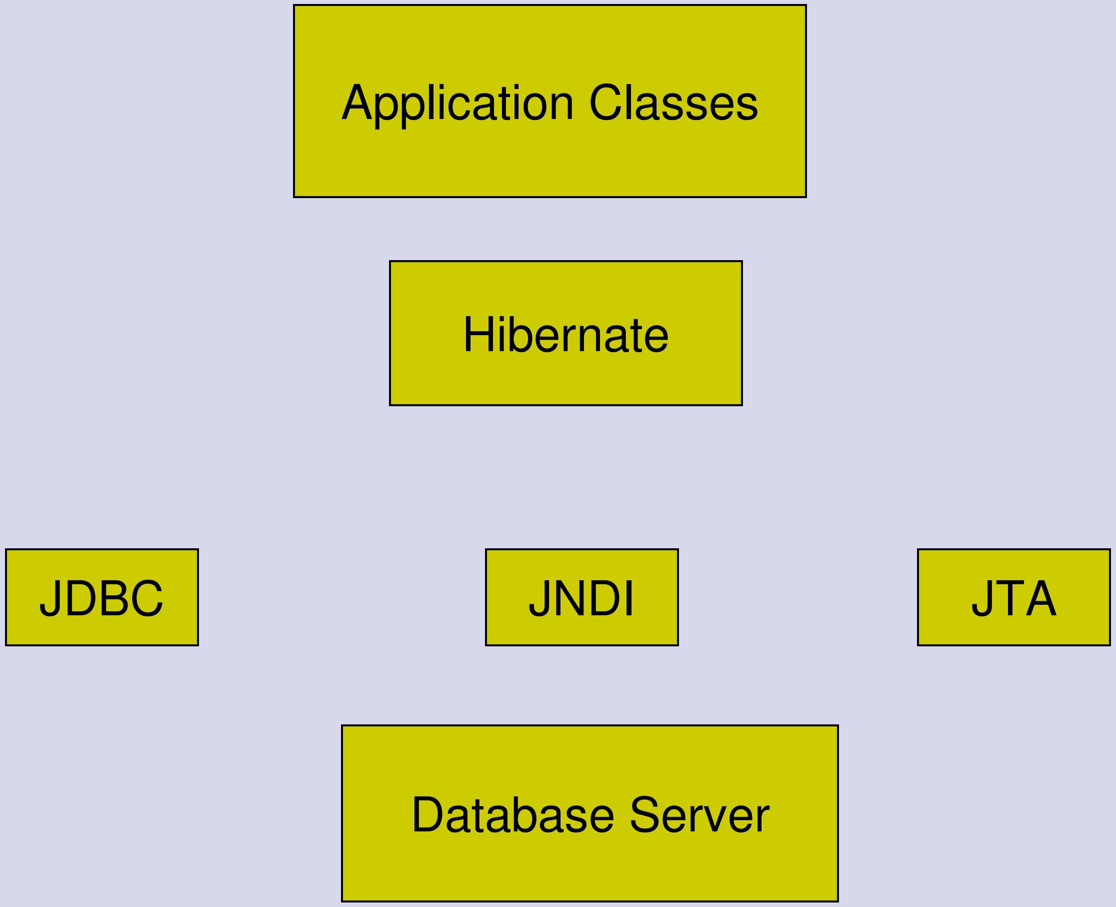
Hibernate is a java-based middleware designed to complete O/R mapping model.

Advantages

1. Writing complex SQL code is not needed.
2. Mapping exercise is straight forward
3. Hibernate can persist even the most complex classes
4. Supports Composition, inheritance, and collections. Etc.

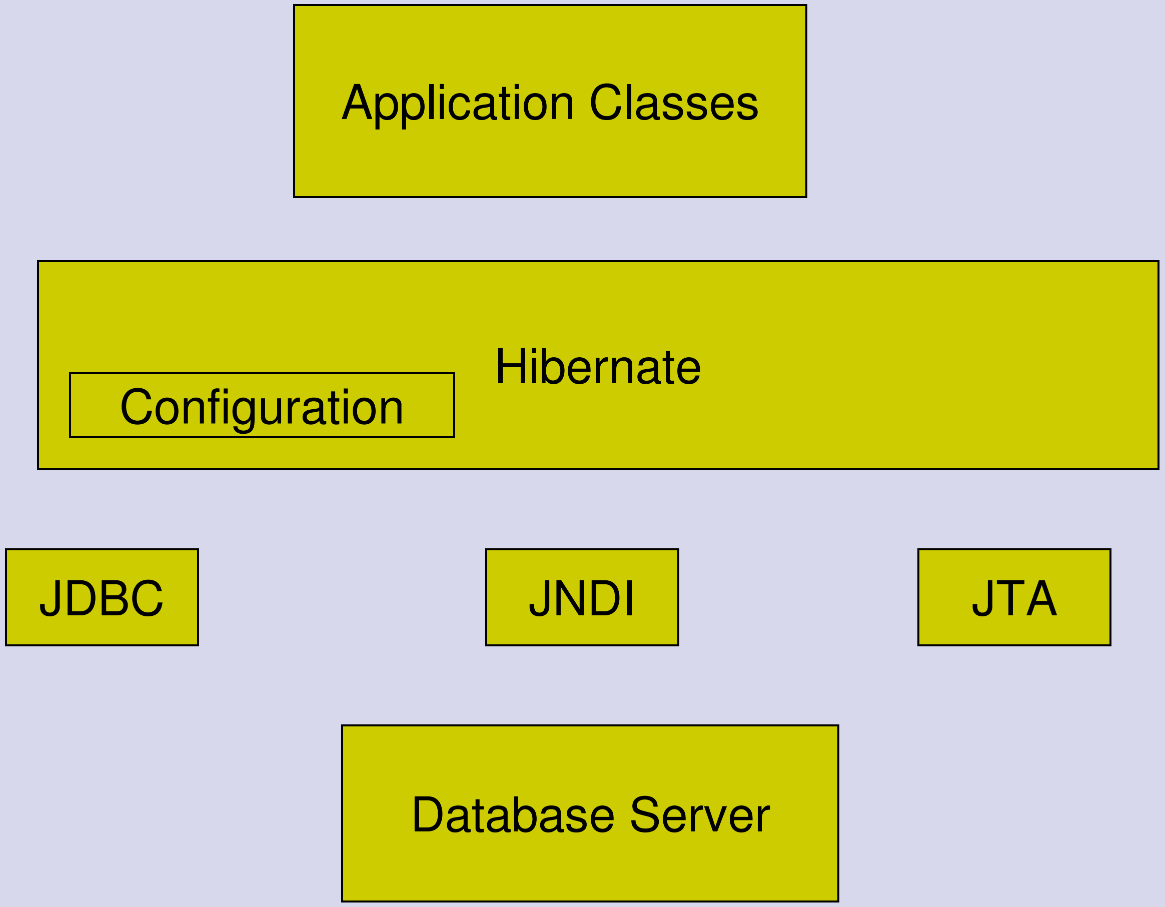
Chapter 2- Introduction to Hibernate

Hibernate Architecture



Chapter 2- Introduction to Hibernate

Hibernate Configuration



Chapter 2- Introduction to Hibernate

Hibernate Configuration

The Configuration class's operation has two key components, the database connection and the class-mapping setup.

The first component is handled through one or more configuration file supported by Hibernate. These files are hibernate.properties and hibernate.cfg.xml file.

Chapter 2- Introduction to Hibernate

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

<hibernate-configuration>

    <session-factory>

        <!-- Database connection settings -->
        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost/hibernate</property>
        <property name="connection.username">root</property>
        <property name="connection.password">mysql</property>

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

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

</hibernate-configuration>
```

Chapter 2- Introduction to Hibernate

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

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

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

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

        <mapping resource="com/test/Employee.hbm.xml"/>
        <mapping resource="com/test/Address.hbm.xml"/>


    </session-factory>
```

Chapter 2- Introduction to Hibernate

Now you can initialize Hibernate using

```
SessionFactory session=new Configuration().configure().buildSessionFactory()
```

Or

```
SessionFactory sessions=new Configuration()
.configure("/com/hiber.cfg.xml").buildSessionFactory();
```

Chapter 2- Introduction to Hibernate

The Core Interfaces

1. The interfaces called by applications to perform basic CRUD and querying operations.
   ex: Session, Transaction and Query

2. Interfaces called by application infrastructure code to configure Hibernate
   ex: Configuration class

3. Callback interfaces that allow the application to react to events occuring inside Hibernate, such as
   ex: Interceptor, lifecycle, and validatable

4. Interfaces that allow extension of Hibernate's powerful mapping functionality
   ex; UserType, CompositeUserType and IdentifierGenerator.

Chapter 2- Introduction to Hibernate

The Session Interface

1. Primary interface used by hibernate applications,
2. Lightweight and inexpensive to create and destroy.
3. Need to be Created and destroyed on every requests.
4. Not Threadsafe, should be designed to be used by only one thread at a time.
5. Can be thought of as a session or as a cache or collections of loaded objects in the unit of work.
6. Persistence related operations such as storing and retrieving is done through session.

Chapter 2- Introduction to Hibernate

The SessionFactory Interface

1. Session is obtained from a SessionFactory
2. HeavyWeight and is shared among many application threads.
3. Typically one for the whole application.
4. If your application accessed multiple databases using Hibernate, you'll need one instance for each database.

5. The sessionFactory caches generated SQL statements and other mapping metadata that Hibernate uses at runtime. It also holds cached data that has been read in one unit of work and may be reused in a future unit of work – only if class and collection mappings specify that this second-level cache is desirable.

Chapter 2- Introduction to Hibernate

The Configuration Interface

The Configuration object is used to configure Hibernate.

The application uses a configuration instance to specify the location of mapping documents and Hibernate-specific properties and then create the SessionFactory.

The Transaction interface

The transaction interface is an optional API

Can also use JDBC transaction or JTA etc

Makes hibernate portable between different kinds of execution environments and containers.

Chapter 2- Introduction to Hibernate

Query and Criteria Interfaces

The Query interface allows you to perform queries against the database and control how the query is executed.

Queries written in HQL or in the native SQL dialect of your database

A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

The criteria interface is very similar, it allows you to create and execute object-oriented criteria queries.

A Query instance is lightweight and can't be used outside the Session that created it.

Types

A hibernate type object maps a java type to a database column type

The type may span multiple columns

There is a rich range of build in types, even for many jdk classes. (viz., Serializable)

Hibernate supports user-defined custom types. Through UserType and /compositeUserType interfaces.

Chapter 3- Creating Persistent Classes

Creating Persistent Java Classes.

Hibernate will only persist attributes specified in the mapping document, therefore you can include temporary attributes if needed. There values will be lost after use.

All attributes that will be persisted should be declared private and have setter/getter methods.

They can be private or protected or public.

Hibernate can handle mapping form just about any data type

If an attribute type is collection u need to do some plumbing in mapping

Mapped classes can use the concept of composition

All classes should contain an ID

All classes should have a default constructor

Interfaces, final classes and some inner classes can be mapped to permanent storage

Chapter 3- Creating Persistent Classes

Mapping a Basic Java Class

```
Public class Book
{
Private int id,pages,copyright;
private String title,author,isbn;
Private float cost;
//Generate setter and getter methods for the above fields
………
}
```

Chapter 3- Creating Persistent Classes

```
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="com.test.Book" table="BOOKS">
        <id name="id" type="int" unsaved-value="0">
            <generator class="hilo"/>
        </id>
        <property name="title" />
        <property name="author" />
        <property name="isbn" not-null="true"/>
        <property name="pages" type="integer" column="pagecount"/>
        <property name="copyright"/>
        <property name="cost">
                <column name="cost" sql-type="NUMERIC(12,2)"/>
      </property>       </class></hibernate-mapping>
```

Chapter 3- Creating Persistent Classes

1. unsaved-value – Spefies whether hibernate should persist an object.

The identifier attribute of our Book class is called Id, it holds a value of null for a new object and integer value for an object pulled from storage. Hibernate obtains the value of the identifier and compares it to the value specified in the unsaved-value attribute to determine whether hibernate should persist the object. If the default value of the identifier used in your application isn't null, you need to place the default value in the unsaved-value attribute.

<generator> element

When Hibernate needs to add a new row to the db for a java object that has been instantiated, it must fill the Id column with a unique value in order to uniquely identify this persisted object.

Chapter 3- Creating Persistent Classes

1. Built-in generators

Increment – it performs a select on the db, determines the current largest ID value, and increment to the next value, (if you are in a multithreaded environment this generator isn't safe, and column type should be int,short, and long.

Identity – if the db has an identity column associated with it (db2,mysql,ms-sql,sybas and column type should be int,short, and long)

Sequence – if the db has a sequence column (db2, postgre, oracle, SAP and column type should be int,short, and long)

Chapter 3- Creating Persistent Classes

1. Built-in generators

Hilo –generates unique IDs for a table, they need not be sequential. This generator must have access to a secondary table to determine the seed value.

The default table is hibernate-unique-key and the required column is next-value. You need to insert one row in the table.

Create table hibernate-unique-key next-value int

Insert into hibernate-unique-key values(100)

For custom type

```
<generator class="hilo">
    <param name="table">hilo</param>
    <param name="column">next</param>
    <param name="max to">500</param>
</generator>
```

Chapter 3- Creating Persistent Classes

1. Built-in generators

Seqhilo

Hibernate combines the sequence and hilo generators.(db2,postgre,oracle,sap)

Uuid.hex

Creates a unique string based on appending the following values, the machines IP address, the startuptime of the current JVm, the current time, and the counter value.

Uuid.string

The generator is like Uuid.hex, but the result is a string 16 characters long.

Native

Picks identity, sequence or hilo, depending on the database.

Chapter 3- Creating Persistent Classes

To put data into the database, let us use a sample application

```
public class HibernateUtil {
        private static final SessionFactory sessionFactory;

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

Chapter 3- Creating Persistent Classes

To put data into the database, let us use a sample application

```
public static final ThreadLocal session=new ThreadLocal();
public static Session currentSession() throws HibernateException
{
      Session s=(Session)session.get();
      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();
      }
}
}
```

Chapter 3- Creating Persistent Classes

```
public class Tester {

        /**
         * @param args
         */
        public static void main(String[] args) {
                // TODO Auto-generated method stub

                Session session = HibernateUtil.currentSession();
                Book book=new Book();
                book.setTitle(….
                …
                session.save(book);
                session.flush();
                session.close();
}
```

Chapter 3- Creating Persistent Classes

Mapping a class with Binary Data

```
Class.. {
Private Blob data;
…
}
```

Mapping

```
<property…
    <column name="data" sql-type="blob"/>
```

Chapter 3- Creating Persistent Classes

Mapping a Serializable Class

In most cases, you persist a java object by specifying attributes in the class that will be stored in the database. An alternative mechanism takes advantage of the Serialization interface available to java classes.

```
Class Counter{
Private Integer ivalue;
…
}
```

Mapping

```
<property name="ivalue" type="serializable">
    <column name="value" sql-type="blob"/>
</property>
```

Chapter 3- Creating Persistent Classes

Usage

```
Counter cou=(Counter)session.load(Counter.class,1)

Int v=cou.getIvalue().intValue();
```

Chapter 3- Creating Persistent Classes

Mapping with Date/Calendar Attributes

```
Class Account{
..
Private Data setupdate;
}
```

Mapping

```
<property…
    <column name="setup" sql-type="Date"/>
…
```

Usage

```
Account a=(Account)session.load(Account.class,1);
a.getSetupdate();
```

Chapter 3- Creating Persistent Classes

Mapping a Read-Only Class

<class name=“a”…. Mutable=“false”>

….

</class>

Using session.save(a) – the row dosent change, since hibernate knows the object couldn’t be updated.

Chapter 3- Creating Persistent Classes

Mapping classes using Versioning/Timestamps

The version or timestamp is used as a replacement for the identifier that would typically be found in the class and corresponding database table. Since the timestamp and version change when an object is updated in the database, you don't need the identifier for uniqueness.

```
class VModule
{
Int id;
String name;
Int version;
…
}
```

Chapter 3- Creating Persistent Classes

Mapping

```
…
<version name="version" column="version' type='integer"
      unsaved-value="undefined"/>
..
```

Sql

```
Create table vmodule {
Id int not null auto-increment primary key,version int,name txt);
```

Usage

```
..
VModule mod=new VModule();
Mod.setName("aa");session.save(mod);session.flush();
Sysout(mod.getVersion());
```

Chapter 3- Creating Persistent Classes

```
Timestamp
Import java.sql.*;
Class Module {
    …
Timestamp timestamp;
}
Mapping
<timestamp name="timestamp" column="stamp" unsaved-value="undefined"/>
Sql
Create table module( .. Stamp timestamp…);
```

Chapter 3- Creating Persistent Classes

Mapping Inheritance with Java Classes.

Table-Per-Class Hierarchy mapping

```
<hibernate-mapping package="com.test">

   <class name="CD" table="CD" discriminator-value="cd">
      <id name="id" column="ID" unsaved-value="0">
         <generator class="hilo"/>
      </id>
      <descriminator column="cd_type" type="string"/>
      <property name="title" column="TITLE"/>
    ...
    <subclass name="SpecialEditionCD" discriminator-value="SpecialEditionCD">
    <property name="newfeatures" type="string"/>
    </subclass>
    <subclass name="InternationalCD" discriminator-value="InternationalCD">
    <property name="country" type="string"/>
    </subclass>
   </class>

</hibernate-mapping>
```

Haaris Infotech
Driven by Technology

Chapter 3- Creating Persistent Classes

Mapping Inheritance with Java Classes.

The first addition in the mapping is the <descriminator> element.

When the user instantiates and saves a CD object, it's saved to a table called CD with the appropriate attributes from the CD class mapped to the database columns.

When the user instantiates and saves a SpecialEditionCD object, it is saved to the same CD table as the CD object. Additional attributes are mapped from the SpecialEditionCD.

id	Title	newfeatures	country	Cd_type
4343	Rock	null	null	Cd
344	Rock	blast	null	SpecialEditionCD
4555	Rock	null	india	InternationalCD

Chapter 3- Creating Persistent Classes

Usage

```
CD cd=…
SpecialEditionCD scd=…

…

Session.save(cd);
Session.save(scd);
Session.save(icd);

Session.flush();
Session.close();
```

Chapter 3- Creating Persistent Classes

Table-Per-Subclass hierarchy Mapping

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


<hibernate-mapping package="com.test">

    <class name="CD" table="CD" discriminator-value="cd">
        <id name="id" column="ID" unsaved-value="0">
            <generator class="hilo"/>
        </id>
        <property name="title" column="TITLE"/>
      ...
      <joined-subclass name="SpecialEditionCD" table="secd">
      <key column="id"/>
      <property name="newfeatures" type="string"/>
      </subclass>
      <joined-subclass name="InternationalCD" table="icd">
      <key column="id"/>
      <property name="country" type="string"/>
      </subclass>
    </class>

</hibernate-mapping>
```

Chapter 3- Creating Persistent Classes

Table per concrete class mapping

Need to create the tables with all the properties needed.

Need to have <class..> element for every class/table.

Chapter 3- Creating Persistent Classes

Working with Column Formulas

```
<class..

    <property name="fullname" formula="concat(firstname,' ',lastname)"/>
</class>
```

Chapter 3- Working with Collections

Working with Collections

Map - HashMap, SortedMap - TreeMap,
Set - HashSet, SortedSet - TreeSet, List - ArrayList,
any array of basic types or other persisted classes.

Associations

Ex:

```
import java.io.Serializable;
import java.util.Set;
public class Employee {

            private String name;
            ….
            private List addresses;
            public Integer getAge() {
                        return age;
            }
            public void setAge(Integer age) {
            ….
            public List getAddresses() {
                        return addresses;
            }
            public void setAddresses(List addresses) {
                        this.addresses = addresses;
            }
```

Chapter 3- Working with Collections

Index Elements

If the collection is based on an index, List, array, or Map, you must have a column in the table to hold the index position of each element in the collection.

Ex: <index column="column" type="type" length="length"/>

In case of Map, you'll most commonly use another class as the index to the values in the map.

<index-many-to-many column="column" class="class"/>

If you need to use a composite index for your collection, you can specify it using the <composite-index>

```
<composite-index class="class">
        <key-property name="name" type="type" column="column"/>
</composite-index>
```

The key-property element to specify the individual pieces of the composite index.

Chapter 3- Working with Collections

Element Elements

All the values of a collection can be basic types or classes except another collection.
If your collection only contains values, you use the <element>

<element column="column" type="type"/>

The column and type are created in the database table.

If your collection contains objects, you use the <many-to-many> element to specify those objects.

<many-to-many column="column" class="class" outer-join="true |false |auto"/>

You can save an entire component in the java collection using the

```
<composite-element class="class">
    <property name=""/>
</composite-element>
```

Chapter 3- Working with Collections

Bi-directional Associations

Lazy Initialization

There can be a issue of performance when considering the loading of a persisted collection. If the collection is very large.

```
<class name="Group" proxy="Group".
          <set lazy="true'></set>
</class>
```

Typically the class is the same as the class being mapped.

Haaris Infotech
Driven by Technology

Chapter 3- Working with Collections

Mapping Maps / SortedMaps

```
Class SupportProperty
{
    private int id;private String name;
    private Map properties;
}
Mapping
<class name="supportProperty" table="supportproperty">
    <id name="id">
        <generator class="native"/>
    </id>
    <map name="properties">
        <key column="part_id'/>
        <index column="property_name" type="string"/>
        <element column="property_value" type="string"/>
    </map>
```

Haaris Infotech
Driven by Technology

Chapter 3- Working with Collections

SQL

```
Create table supportproperty(…

Create table properties (
Id int, property_name text, property_value text);
```

When hibernate persists a SupportProperty object, it stores the name of the object and allows MySQL to automatically create a primary key for the supportproperty table. Hibernate obtains the created primary key and inserts it along with the key/values of the properties map into the properties table.

Chapter 3- Working with Collections

Usage

```
SupportProperty sp=new SupportProperty();

Sp.setName(“aaaaa”);
HashMap p=new HashMap();
p.put(“color”,”blue”);
p.put(“inf”,”mac”);
p.setProperties(p);

…..

Map p2=sp2.getProperties();
Sysout(p2.get(“color”));
```

Chapter 3- Working with Collections

Mapping an Object Map :<many-to-many> element

Chapter 4- Using Persistent Objects

Saving an object into the session

1. session.save(book)

Loading Data into an Object

Public object load(Class class, Serializable id,LockMode mode)
Public object load(Class class, Serializable id)
Public void load(Class class, Serializable id)

Book book=(Book)session.load(User.class,0)
Or session.get(user.class,0); -- will not throw an exception if record dosent exist.

Flusing Objects

During the execution of the flush, all queued SQL statements are executed. Hibernate executes flush automatically when a commit() method is called as well as during the find() and iterate() methods.

Chapter 4- Using Persistent Objects

Deleting objects

Session.delte(book)

Refreshing Objects

Session.refresh(book);

Updating Objects

saveOrUpdate(book) – this method determines whether the provided object should be either saved for the first time or updated.

Update(book) – hibernate determines whether the object truly needs to be updated and then issues the appropriate SQL statement to make the changes in the underlying db.

Update(book,0) -

Flush() – the flush method can determine whether a change has been made to the object, if so it updates the database.

Chapter 4- Using Persistent Objects

Finding Object

Public List find(String query)

Public List find(String query,Object value, Type type)

Public List find(String query, Object[] values, Type[] types)

List books=session.find("from Book");

Session.find("from User where username "+username +"'");

Session.find("from User where username =?",username,Hibernate.String)

Session.find("from User where username =? Or upper(username)",new
    Object[]{username,username},new Type[]{Hibernate.STRING,Hibernate.STRING}

Chapter 4- Using Persistent Objects

Finding Large Resultsets

```
Public List iterate(String query)
Public List iterate(String query,Object value, Type type)
Public List iterate(String query, Object[] values, Type[] types)

Iterator iter=session.iterate("from User user order by user");
While(iter.hasNext())
{
User user=(User)iter.next();
…..
}
```

Chapter 4- Using Persistent Objects

Filtering Collections.

Book book=session.load(Book.class,0)

Collection authors=session.filter(book.getAuthors(),"where this.authorname like 'a%' ");

Collection authors=session.filter(book.getAuthors(),"select this.authorname where this.city like 'c%' ")

Here we're returning only the authornames instead of the entire object.

Chapter 4- Using Persistent Objects

Scalar Queries

So far we have retrieved a object or list of objects, but hibernate also lets you to retrieve individual attributes or groups of attributes.

```
      List list=session.find(“select book.name,size(book.authors) from book);
      for(int i=0;i<list.size();i++)
      {
                Object [] row=list.get(i);
                String name=(String)row[0];
….
      }
```

We can also pull individual objects too

```
      List list=session.find(“select book, book.name,size(book.authors) from book);
..
      Book book=(Book)row[0];
```

Queries and Named Queries

Using the Query object

Allows more control over the results

```
      Query query=session.createQuery("from Book book where book.price >100")
If(query.uniqueResult())
{
      …..
}
```

Named Queries

```
<query name="findALL">
    <![CDATA[ from Book book ]]>
</query>
```

Make this entry in the mapping file

```
Query query=session.getNamedQuery("findALL");
Query.setMaxResults(25);
Query.setFirstResult(5);
List books=query.list()
..
```

Chapter 4- Using Persistent Objects

Named Queries with Named Parameters

```
<query name="findALL">
    <![CDATA[ from Book book where book.price >  :value ]]>
</query>
```

Make this entry in the mapping file

```
Query query=session.getNamedQuery("findALL");
Query.setMaxResults(25);
Query.setFirstResult(5);
Query.setInt("value",100)
List books=query.list()
..
```

Chapter 4- Using Persistent Objects

Criteria Query

```
Criteria criteria=session.createCriteria(Book.class)

Criteria.add(Expression.eq("name","ramu");
Criteria.setMaxResults(5);
List book=criteria.list();
```