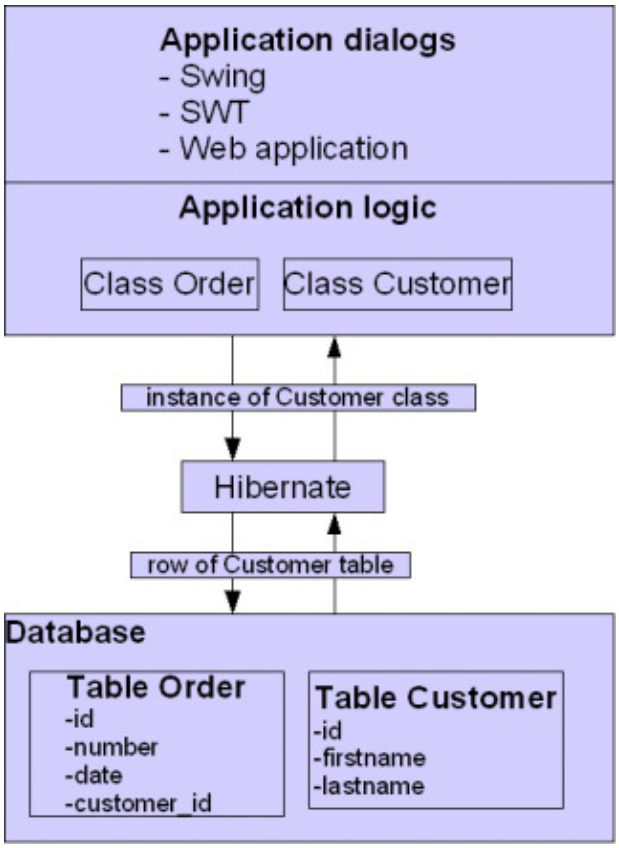
Hibernate Tutorial

Short Introduction – (refresh of memories from the lectures…)

Hibernate is a solution for object relational mapping and a persistence management solution or persistent layer. This is probably not understandable for anybody learning Hibernate.

What you can imagine is probably that you have your application with some functions (business logic) and you want to save data in a database. When you use Java, all the business logic normally works with objects of different class types. Your database tables are not at all objects.

Hibernate provides a solution to map database tables to a class. It copies one row of the database data to a class. In the other direction it supports to save objects to the database. In this process the object is transformed to one or more tables.



Part A – Configuring the Data Storage

1. Create a new JavaDB database instance in the name of **demo**.
   ~~host=localhost, port=1527~~, username=app, and password=app

2. Create the following table in the APP schema
   Person (personid, name, age)

Use the database connection provided for the database demo, when it was created.

Note: Watch the demo at the practical session!

Part B – Create a Java Console Application for a Simple POJO

1. Create a new java console application. Name it as **hibernatedemo**.

2. Import the Hibernate Libraries to the NetBeans project.

3. Create a new POJO Class for **Person** in **hibernatedemo** package the as below

```
public class Person {
    private int personid ;
    private String name;
    private int age;

    // Generate Getters and Setters for the above
    // Properties

    @Override
    public String toString() {
        return "Person: "+getPersonid()+
               " Name: "+getName()+
               " Age: "+getAge();
    }
}
```

Note the toString() method is the Person Class has be overridden to print the content of the Person Class.

4. Create the Hibernate configuration

   Use the Hibernate Configuration file with the help of the Hibernate Configuration Wizard.

   Add the following *properties* to the generated configuration file

   Optional configuration property
   `hibernate.show_sql=true`

   Miscellaneous configuration property
   `hibernate.current_session_context_class=thread`

5. Create the Hibernate Mapping File.

   You can use a XML file or annotations in the source code to define, the mapping of the class/ class attributes to a database table/table column.

   Use the Hibernate Mapping Wizard to generate the mapping file in the **hibernatedemo** package. File name can be person.hbm.

   Add the following property mappings to the generated xml file

```
<id column="personid" name="personid">
      <generator class="increment"/>
</id>
<property column="name" name="name"/>
<property column="age" name="age"/>
```

6. Create a SessionFactory.

   A session factory is important for the Hibernate Framework to make transactions from the client to the database. The Session Factory implements the Singleton design pattern, which ensures, that only one instance of the session is used per thread.
   Note: You should only get your Hibernate session from this factory.

   Create a class named **SessionFactoryUtil** in the package **hibernatedemo**.
   This **SessionFactoryUtil** class can be added from the available HibernateUtil.Java template file

   Add the methods openSession(), getCurrentSession(), close() to the generated class.

```
package hibernatedemo;

import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.SessionFactory;
import org.hibernate.Session;
```

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

    static {
        try {
            // Create the SessionFactory from standard (hibernate.cfg.xml)
            // config file.
            sessionFactory = new AnnotationConfiguration().configure()
                  .buildSessionFactory();
        } catch (Throwable ex) {

            // Log the exception.
            System.err.println("Initial SessionFactory creation failed."
            + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    /**
    * Opens a session and will not bind it to a session context
    * @return the session
    */
    public static Session openSession() {
        return sessionFactory.openSession();
    }

    /**
    * Returns a session from the session context.
    * If there is no session in the context it opens a session,
    * stores it in the context and returns it.
    * This factory is intended to be used with a hibernate.cfg.xml
    * including the following property <property
    * name="current_session_context_class">thread</property>
    * This would return
    * the current open session or if this does not exist, will create a new
    * session
    *
    * @return the session
    */
    public static Session getCurrentSession() {
        return sessionFactory.getCurrentSession();
    }

    /**
    * closes the session factory
    */
    public static void close(){
        if (sessionFactory != null)
            sessionFactory.close();
    }
}
```

7. Create a Test Client

Now a Test Client needs to be written and this can be done in the Main.Java class itself.

We need to write three methods to manipulate the lifecycle operation of the persistent object

- **listPerson()** – List all persons in the Console
- **createPerson(Person p)** – Creates new person p in the Person Table
- **deletePerson(Person p)** – Removes the person p in the Person Table
- **updatePerson(Person p)** – Updates a person p in the Person Table

```
 private static void listPerson() {
    Transaction tx = null;
    Session session = SessionFactoryUtil.getCurrentSession();
    try {
        tx = session.beginTransaction();
        List persons = session.createQuery(
                        "select p from Person as p").list();
        System.out.println("*** Content of the Person Table ***");
        System.out.println("*** Start ***");
        for (Iterator iter = persons.iterator(); iter.hasNext();) {
            Person element = (Person) iter.next();
            System.out.println(element);
        }
        System.out.println("*** End ***");
        tx.commit();
    } catch (RuntimeException e) {
        if (tx != null && tx.isActive()) {
            try {
                // Second try catch as the rollback could fail as well
                tx.rollback();
            } catch (HibernateException e1) {
                System.out.println("Error rolling back transaction");
            }
            throw e;
        }
    }
}

private static void deletePerson(Person person) {
    Transaction tx = null;
    Session session = SessionFactoryUtil.getCurrentSession();
    try {
        tx = session.beginTransaction();
        session.delete(person);
        tx.commit();
    } catch (RuntimeException e) {
        if (tx != null && tx.isActive()) {
            try {
                // Second try catch as the rollback could fail as well
                tx.rollback();
            } catch (HibernateException e1) {
                System.out.println("Error rolling back transaction");
```

```
                }
                // throw again the first exception
                throw e;
            }
        }
    }

    private static void createPerson(Person person) {
        Transaction tx = null;
        Session session = SessionFactoryUtil.getCurrentSession();
        try {
            tx = session.beginTransaction();
            session.save(person);
            tx.commit();
        } catch (RuntimeException e) {
            if (tx != null && tx.isActive()) {
                try {
                    // Second try catch as the rollback could fail as well
                    tx.rollback();
                } catch (HibernateException e1) {
                    System.out.println("Error rolling back transaction");
                }
                // throw again the first exception
                throw e;
            }
        }
    }

    private static void updatePerson(Person person) {
        Transaction tx = null;
        Session session = SessionFactoryUtil.getCurrentSession();
        try {
            tx = session.beginTransaction();
            session.update(person);
            tx.commit();
        } catch (RuntimeException e) {
            if (tx != null && tx.isActive()) {
                try {
                    // Second try catch as the rollback could fail as well
                    tx.rollback();
                } catch (HibernateException e1) {
                    System.out.println("Error rolling back transaction");
                }
                // throw again the first exception
                throw e;
            }
        }
    }
```

8. Try to understand the above methods.
   How a session is obtained, how a transaction is executed and committed, and how finally handle exceptions and rollback any problematic transactions.

9. Use the above methods in the Main method and observe the console output for the results

```
public static void main(String[] args) {

        Person p1 = new Person();
        p1.setName("Saman");
        p1.setAge(22);
        createPerson(p1);

        Person p2 = new Person();
        p2.setName("Peter");
        p2.setAge(31);
        createPerson(p2);

        listPerson();

        p1.setAge(44);
        updatePerson(p1);

        p2.setName("Peter John");
        updatePerson(p2);

        listPerson();

    }
```

10. Now, insert some Persons with different ages and then perform a query to list all persons who are above the age 25. You may use HQL to achieve this.
    Hint:

```
Tx = session.beginTransaction();
Query q = session.createQuery("select p from Person as p where
                                                    p.age>:age");
Person fooPerson = new Person();
fooPerson.setAge(age);
q.setProperties(fooPerson);
List persons = q.list();
```

Further Reference

1. HQL (Hibernate Query Language)
   http://docs.jboss.org/hibernate/orm/3.3/reference/en/html/queryhql.html

2. Working with Objects in Hibernate
   http://docs.jboss.org/hibernate/orm/3.3/reference/en-US/html/objectstate.html

Part C – Create a Java Console Application for POJO having a One-Many Relationship

Now, let's look how a one-to-many relationship can be associated with the Person Class.

1. Assume that the Person has many Hats. That is the person called "saman" can have hat1, hat 2, ... hatN

2. Create the Hat table in the demo database as Hat (hatid, color, size, personid), where personid will be the foreign key

3. Create the Hat class.

```
public class Hat {
    private int hatid;
    private String color;
    private String size;
    private int personid;

   // Getters and Setters

    public String toString() {
        return "Hat: "+getHatid()+
               " Color: "+getColor()+
               " Size: "+getSize();
    }
}
```

4. Modify the Person class such that now it will have a collection attribute called **hats.**
   Also have methods to add and remove hats for a person also (e.g addHat() and removeHat()).

```
public class Person {
    private int personid ;
    private String name;
    private int age;
    private Set hats;

    public Person(){
        hats = new HashSet();
    }

    // Getters and Setters

    public void addHat(Hat hat){
        this.hats.add(hat);
    }

    public void removeHat(Hat hat){
        this.hats.remove(hat);
    }

    public String toString() {
```

```
        String personString = "Person: "+getPersonid()+
                              " Name: "+getName()+
                              " Age: "+getAge();
        String hatString = "";
        for (Iterator iter = hats.iterator(); iter.hasNext();) {
            Hat hat = (Hat) iter.next();
            hatString = hatString + "\t\t"+ hat.toString()+"\n";
        }
        return personString + "\n" + hatString;
    }
}
```

5. Create the Hibernate mapping file for the Hat class

```
<?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="hibernatedemo.Hat" table="HAT">
    <id column="hatid" name="hatid">
      <generator class="increment"/>
    </id>
    <property column="personid" name="personid"/>
    <property column="color" name="color"/>
    <property column="size" name="size"/>
  </class>
</hibernate-mapping>
```

6. Modify the Person Hibernate mapping file to have the Person-to-Hat One-to-Many relationship

```
<?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="hibernatedemo.Person" table="PERSON">
    <id column="personid" name="personid">
      <generator class="increment"/>
    </id>
    <property column="name" name="name"/>
    <property column="age" name="age"/>
    <set cascade="all" name="hats" table="HAT">
      <key column="personid"/>
      <one-to-many class="hibernatedemo.Hat"/>
    </set>
  </class>
</hibernate-mapping>
```

7. Run some sample test code in the Main.Java as below and observe the console output

```
    public static void main(String[] args) {
        Person p1 = new Person();
        p1.setName("Saman With Hats");
        p1.setAge(30);

        Hat h1 = new Hat();
        h1.setColor("Black");
        h1.setSize("Small");

        Hat h2 = new Hat();
        h2.setColor("White");
        h2.setSize("Large");

        p1.addHat(h1);
        p1.addHat(h2);

        createPerson(p1);
        listPerson();
    }
```

Here you may observe that when you create the person, then the relevant person p1 was inserted to the Person Table and additionally the person's hats h1 and h2 were also inserted to the Hat Table.

Homework

Try to implement the above exercise using the Hibernate Annotations.

http://docs.jboss.org/hibernate/annotations/3.5/reference/en/html_single/

Requirements

- Make sure you have JDK 5.0 or above installed.
- Download and unpack the Hibernate Core distribution from the Hibernate website. Hibernate 3.5 and onward contains Hibernate Annotations.

END OF DOCUMENT