Hibernate

Topics to be covered

- ORM
- Overview of Hibernate
- Hibernate Architecture
- Hibernate Mapping Types
- Hibernate O/R Mapping
- Hibernate Annotation
- Hibernate Query Language

ORM

- ORM stands for Object-Relational Mapping (ORM)
- It is a programming technique for converting data between relational databases and object oriented programming languages such as Java, C# etc.
- An ORM solution consists of the following four entities:

  Solutions:

  - An API to perform basic CRUD operations on objects of persistent classes.
  - A language or API to specify queries that refer to classes and properties of classes.
  - A configurable facility for specifying mapping metadata.
  - A technique to interact with transactional objects to perform optimization functions.

Java ORM Frameworks:

- Enterprise JavaBeans Entity Beans
- Java Data Objects
- Castor
- TopLink
- Spring DAO
- **Hibernate**
- And many more

Hibernate Introduction

- It was started in 2001 by Gavin King
- The stable release of Hibernate till July 16, 2014, is hibernate 4.3.6
- Hibernate framework simplifies the development of java application to interact with the database.
- Hibernate is an open source, lightweight, ORM (Object Relational Mapping) tool.
- An ORM tool simplifies the data creation, data manipulation and data access.
- It is a programming technique that maps the object to the data stored in the database.
- The ORM tool internally uses the JDBC API to interact with the database.

Advantages of Hibernate

1) Opensource and Lightweight

2) Fast performance

3) Database Independent query

4) Automatic table creation

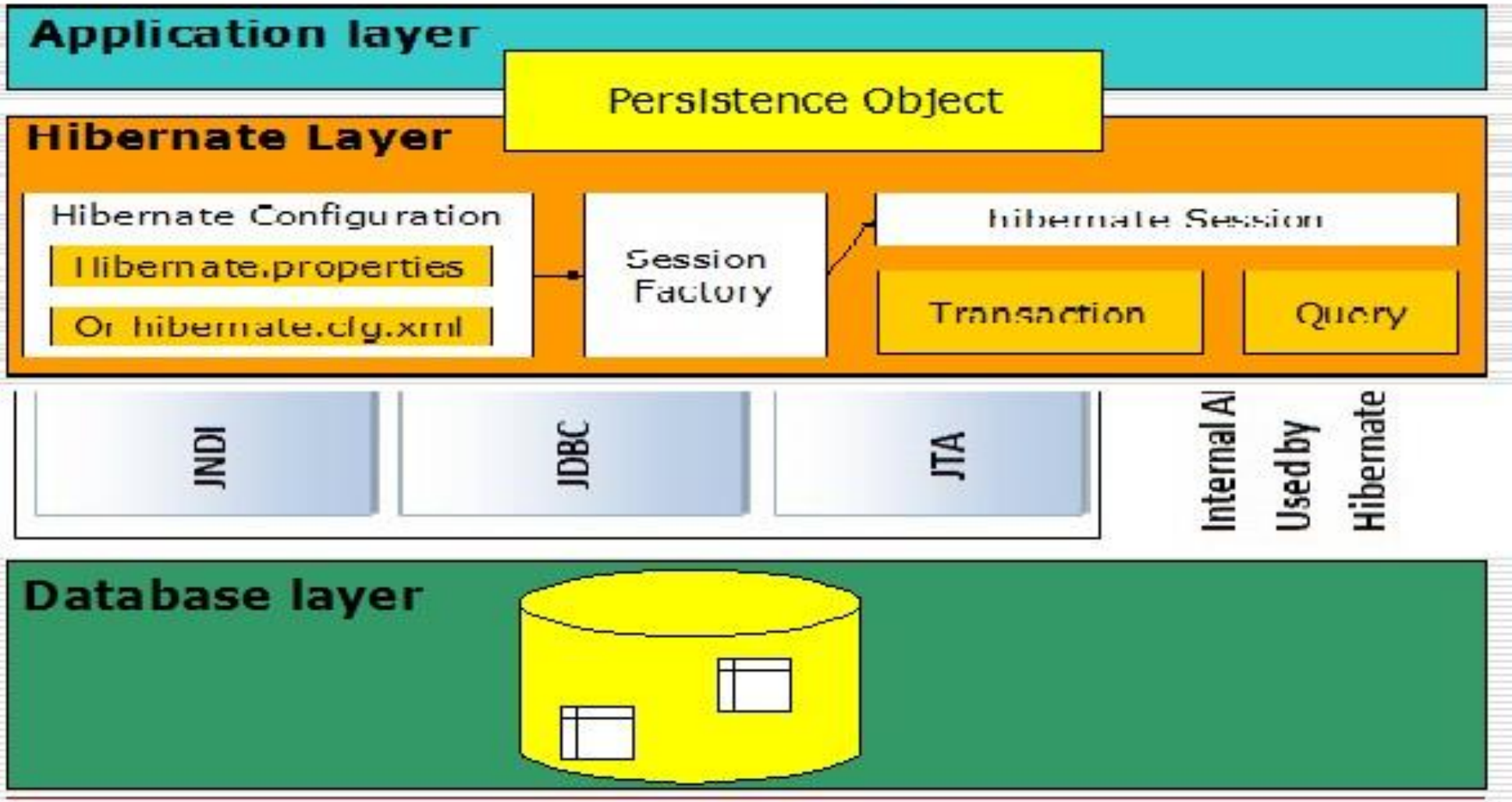
5) Simplifies complex join

6) Provides query statistics and database status

Supported Databases:

- DB2
- MySQL
- PostgreSQL
- Oracle
- Microsoft SQL Server Database, etc….

Hibernate Architecture



Elements of Hibernate Architecture

SessionFactory

- The SessionFactory is a factory of session and client of ConnectionProvider.
- It holds second level cache (optional) of data.
- The org.hibernate.SessionFactory interface provides factory method to get the object of Session.

Session

- The session object provides an interface between the application and data stored in the database.
- It holds a first-level cache (mandatory) of data.
- The org.hibernate.Session interface provides methods to insert, update and delete the object.
- It also provides factory methods for Transaction, Query and Criteria.

Transaction

- The org.hibernate.Transaction interface provides methods for transaction management.
- It is optional

Elements of Hibernate Architecture (contd.)

ConnectionProvider

- It is a factory of JDBC connections.
- It abstracts the application from DriverManager or DataSource.
- It is optional.

TransactionFactory

- It is a factory of Transaction.
- It is optional.

Hibernate – Configuration

- Hibernate requires to know in advance where to find the mapping information that defines how your Java classes relate to the database tables.
- Hibernate also requires a set of configuration settings related to database and other related parameters.
- All such information is usually supplied as a standard Java properties file called **hibernate.properties**, or as an XML file named **hibernate.cfg.xml**.

S.N.	Properties and Description
1	**hibernate.dialect** This property makes Hibernate generate the appropriate SQL for the chosen database.
2	**hibernate.connection.driver_class** The JDBC driver class.
3	**hibernate.connection.url** The JDBC URL to the database instance.

Hibernate Properties (contd.)

4	**hibernate.connection.username** The database username.
5	**hibernate.connection.password** The database password.
6	**hibernate.connection.pool_size** Limits the number of connections waiting in the Hibernate database connection pool.
7	**hibernate.connection.autocommit** Allows autocommit mode to be used for the JDBC connection.

Hibernate - Sessions

- A Session is used to get a physical connection with a database.
- The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database.
- Persistent objects are saved and retrieved through a Session object.
- The main function of the Session is to offer create, read and delete operations for instances of mapped entity classes.
- Instances may exist in one of the following three states at a given point in time:

Transient:

- A new instance of a a persistent class which is not associated with a Session and has no representation in the database and no identifier value is considered transient by Hibernate.

Persistent:

- You can make a transient instance persistent by associating it with a Session. A persistent instance has a representation in the database, an identifier value and is associated with a Session.

Detached:

- Once we close the Hibernate Session, the persistent instance will become a detached instance.

Session Interface Methods

S.N.	Session Methods and Description
1	**Transaction beginTransaction()** Begin a unit of work and return the associated Transaction object.
2	**void cancelQuery()** Cancel the execution of the current query.
3	void clear() Completely clear the session.
4	**Connection close()** End the session by releasing the JDBC connection and cleaning up.
5	Criteria createCriteria(Class persistentClass) Create a new Criteria instance, for the given entity class, or a superclass of an entity class.
6	Criteria createCriteria(String entityName) Create a new Criteria instance, for the given entity name.
7	Serializable getIdentifier(Object object) Return the identifier value of the given entity as associated with this session.
8	**Query createFilter(Object collection, String queryString)** Create a new instance of Query for the given collection and filter string.

Session Interface Methods (contd.)

9	**Query createQuery(String queryString)** Create a new instance of Query for the given HQL query string.
10	SQLQuery createSQLQuery(String queryString) Create a new instance of SQLQuery for the given SQL query string.
11	**void delete(Object object)** Remove a persistent instance from the datastore.
12	void delete(String entityName, Object object) Remove a persistent instance from the datastore.
13	**Session get(String entityName, Serializable id)** Return the persistent instance of the given named entity with the given identifier, or null if there is no such persistent instance.
14	**SessionFactory getSessionFactory()** Get the session factory which created this session.
15	void refresh(Object object) Re-read the state of the given instance from the underlying database.
16	**Transaction getTransaction()** Get the Transaction instance associated with this session.

Session Interface Methods (contd.)

17	**boolean isConnected()** Check if the session is currently connected.
18	boolean isDirty() Does this session contain any changes which must be synchronized with the database?
19	boolean isOpen() Check if the session is still open.
20	Serializable save(Object object) Persist the given transient instance, first assigning a generated identifier.
21	**void saveOrUpdate(Object object)** Either save(Object) or update(Object) the given instance.
22	**void update(Object object)** Update the persistent instance with the identifier of the given detached instance.
23	**void update(String entityName, Object object)** Update the persistent instance with the identifier of the given detached instance

Hibernate - Mapping Types

- The **types** declared and used in the mapping files are not Java data types; they are not SQL database types either.
- These types are called Hibernate mapping types, which can translate from Java to SQL data types and vice versa.

Primitive types:

Mapping type	Java type	ANSI SQL Type
integer	int or java.lang.Integer	INTEGER
long	long or java.lang.Long	BIGINT
short	short or java.lang.Short	SMALLINT
float	float or java.lang.Float	FLOAT
double	double or java.lang.Double	DOUBLE

Primitive types (contd.)

big_decimal	java.math.BigDecimal	NUMERIC
character	java.lang.String	CHAR(1)
string	java.lang.String	VARCHAR
byte	byte or java.lang.Byte	TINYINT
boolean	boolean or java.lang.Boolean	BIT
yes/no	boolean or java.lang.Boolean	CHAR(1) ('Y' or 'N')
true/false	boolean or java.lang.Boolean	CHAR(1) ('T' or 'F')

Date and time types:

Mapping type	Java type	ANSI SQL Type
date	java.util.Date or java.sql.Date	DATE
time	java.util.Date or java.sql.Time	TIME
timestamp	java.util.Date or java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE

Binary and large object types:

Mapping type	Java type	ANSI SQL Type
binary	byte[]	VARBINARY (or BLOB)
text	java.lang.String	CLOB
serializable	any Java class that implements java.io.Serializable	VARBINARY (or BLOB)
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB

JDK-related types:

Mapping type	Java type	ANSI SQL Type
class	java.lang.Class	VARCHAR
locale	java.util.Locale	VARCHAR
timezone	java.util.TimeZone	VARCHAR
currency	java.util.Currency	VARCHAR

Steps to create Hibernate Application

1. Create the Persistent class
2. Create the mapping file for Persistent class
3. Create the Configuration file
4. Create the class that retrieves or stores the persistent object
5. Load the jar file
6. Run the hibernate application

1) Create the Persistent class

A simple Persistent class should follow some rules:

A no-arg constructor:

•It is recommended that you have a default constructor at least package visibility so that hibernate can create the instance of the Persistent class by newInstance() method.

Provide an identifier property (optional):
•It is mapped to the primary key column of the database.

Declare getter and setter methods (optional):

•The Hibernate recognizes the method by getter and setter method names by default.

Prefer non-final class:

•Hibernate uses the concept of proxies, that depends on the persistent class.

•The application programmer will not be able to use proxies for lazy association fetching.

Employee.java

```
public class Employee {
private int id;
private String firstName,lastName;

public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
public String getFirstName() {
    return firstName;
}
public void setFirstName(String firstName) {
    this.firstName = firstName;
}
public String getLastName() {
    return lastName;
}
public void setLastName(String lastName) {
    this.lastName = lastName;
}
}
```

2) Create the mapping file for Persistent class

- The mapping file name conventionally, should be class_name.hbm.xml. There are many elements of the mapping file.

hibernate-mapping
- It is the root element in the mapping file.

class

It is the sub-element of the hibernate-mapping element. It specifies the Persistent class.

id
- It is the subelement of class. It specifies the primary key attribute in the class.

generator

•It is the subelement of id.
•It is used to generate the primary key.

property
•It is the subelement of class that specifies the property name of the Persistent class.

employee.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="com.javatpoint.mypackage.Employee" table="emp1000">
    <id name="id">
     <generator class="assigned"></generator>
    </id>

    <property name="firstName"></property>
    <property name="lastName"></property>

  </class>

 </hibernate-mapping>
```

3) Create the Configuration file

- The configuration file contains informations about the database and mapping file. Conventionally, its name should be hibernate.cfg.xml .

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="hbm2ddl.auto">update</property>
        <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
        <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
        <property name="connection.username">system</property>
        <property name="connection.password">oracle</property>
        <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>

    <mapping resource="employee.hbm.xml"/>
    </session-factory>

</hibernate-configuration>
```

4) Create the class that retrieves or stores the object

- In this class, we are simply storing the employee object to the database.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class StoreData {
public static void main(String[] args) {

    //creating configuration object
    Configuration cfg=new Configuration();
    cfg.configure("hibernate.cfg.xml");//populates the data of the configuration file

    //creating seession factory object
    SessionFactory factory=cfg.buildSessionFactory();

    //creating session object
    Session session=factory.openSession();

    //creating transaction object
    Transaction t=session.beginTransaction();
```

(contd.)

```
    Employee e1=new Employee();
    e1.setId(115);
    e1.setFirstName("sonoo");
    e1.setLastName("jaiswal");

    session.persist(e1);//persisting the object

    t.commit();//transaction is commited
    session.close();

    System.out.println("successfully saved");

}
}
```

5) Load the jar file

- For successfully running the hibernate application, you should have the hibernate4.jar file.

- Some other jar files or packages are required such as

    - ascglib
    - log4j
    - commons
    - SLF4J
    - dom4j
    - xalan
    - xerces

6) Run the hibernate application

To run the hibernate application :

•install the oracle10g for this example.

•load the jar files for hibernate.

•Now Run the StoreData class by **java com.javatpoint.mypackage.StoreData**

Hibernate - Query Language

- Hibernate Query Language (HQL) is an object-oriented query language, similar to SQL, but instead of operating on tables and columns
- HQL works with persistent objects and their properties.
- HQL queries are translated by Hibernate into conventional SQL queries which in turns perform action on database.

FROM Clause

- We will use FROM clause if you want to load a complete persistent objects into memory.
- Following is the simple syntax of using FROM clause:

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
List results = query.list();
```

AS Clause

- The **AS** clause can be used to assign aliases to the classes in your HQL queries, specially when you have long queries.

```
String hql = "FROM Employee AS E";
```

SELECT Clause

```
String hql = "SELECT E.firstName FROM Employee E";
```

WHERE Clause

```
String hql = "FROM Employee E WHERE E.id = 10";
```

ORDER BY Clause

```
String hql = "FROM Employee E WHERE E.id > 10 ORDER BY E.salary DESC";
```

GROUP BY Clause

```
String hql = "SELECT SUM(E.salary), E.firtName FROM Employee E " + "GROUP BY E.firstName";
```

Hibernate Annotation

- We have seen how Hibernate uses XML mapping file for the transformation of data from POJO to database tables and vice versa.
- Hibernate annotations is the newest way to define mappings without a use of XML file.
- You can use annotations in addition to or as a replacement of XML mapping metadata.

```
import javax.persistence.*;

@Entity
@Table(name = "EMPLOYEE")
public class Employee {
   @Id @GeneratedValue
   @Column(name = "id")
   private int id;

   @Column(name = "first_name")
   private String firstName;

@Column(name = "last_name")
   private String lastName;

   @Column(name = "salary")
   private int salary;

   public Employee() {}
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   } …….
```

@Entity Annotation:

- we used the @Entity annotation to the Employee class which marks this class as an entity bean, so it must have a no-argument constructor that is visible with at least protected scope.

@Table Annotation:

- The @Table annotation allows you to specify the details of the table that will be used to persist the entity in the database.

@Id and @GeneratedValue Annotations:

- Each entity bean will have a primary key, which you annotate on the class with the @Id annotation.
- By default, the @Id annotation will automatically determine the most appropriate primary key generation strategy to be used but you can override this by applying the @GeneratedValue annotation which takes two parameters strategy and generator

@Column Annotation:

- The @Column annotation is used to specify the details of the column to which a field or property will be mapped.

- Following are the most commonly used attributes with column annotation:

•**name** attribute permits the name of the column to be explicitly specified.

•**length** attribute permits the size of the column used to map a value particularly for a String value.

•**nullable** attribute permits the column to be marked NOT NULL when the schema is generated.

•**unique** attribute permits the column to be marked as containing only unique values.

Changes in application class

```
public class ManageEmployee {
 private static SessionFactory factory;
 public static void main(String[] args) {
   try{
     factory = new AnnotationConfiguration().
             configure().
            addAnnotatedClass(Employee.class).
             buildSessionFactory();

    }catch (Throwable ex) {
          System.err.println("Failed to create sessionFactory object." + ex);
          throw new ExceptionInInitializerError(ex);
        }
        ManageEmployee ME = new ManageEmployee();

        /* Add few employee records in database */
        Integer empID1 = ME.addEmployee("Zara", "Ali", 1000);…….}
```

Hibernate - O/R Mappings

- Collections Mappings
- Association Mappings
- Component Mappings

Collections Mappings

- If an entity or class has collection of values for a particular variable, then we can map those values using any one of the collection interfaces available in java

Collection type	Mapping and Description
java.util.Set	This is mapped with a <set> element and initialized with java.util.HashSet
java.util.SortedSet	This is mapped with a <set> element and initialized with java.util.TreeSet. The sort attribute can be set to either a comparator or natural ordering.
java.util.List	This is mapped with a <list> element and initialized with java.util.ArrayList
java.util.Collection	This is mapped with a <bag> or <ibag> element and initialized with java.util.ArrayList
java.util.Map	This is mapped with a <map> element and initialized with java.util.HashMap
java.util.SortedMap	This is mapped with a <map> element and initialized with java.util.TreeMap. The **sort** attribute can be set to either a comparator or natural ordering.

Java.util.Set Example

- A **Set** is a java collection that does not contain any duplicate element.

Step 1. Define RDBMS Tables

```
create table EMPLOYEE (
id INT NOT NULL auto_increment,
first_name VARCHAR(20) default NULL,
last_name VARCHAR(20) default NULL, salary INT default NULL,
PRIMARY KEY (id) );
```

```
create table CERTIFICATE (
id INT NOT NULL auto_increment,
certificate_name VARCHAR(30) default NULL,
employee_id INT default NULL,
PRIMARY KEY (id) );
```

Step 2. Define POJO Classes

```
public class Employee {
   private int id;
   private String firstName;
   private String lastName;
   private int salary;
   private Set certificates;

   public Employee() {}
   public Employee(String fname, String lname, int
salary) {
      this.firstName = fname;
      this.lastName = lname;
      this.salary = salary;
   }
 // getters and setters….
   public Set getCertificates() {
      return certificates;
   }
   public void setCertificates( Set certificates ) {
      this.certificates = certificates; }
}
```

```
public class Certificate {
   private int id;
   private String name;

   public Certificate() {}
   public Certificate(String name) {
      this.name = name;
   }
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getName() {
      return name;
   }
   public void setName( String name ) {
      this.name = name;
   }
```

Step 3. Define Hibernate Mapping File:

```
<hibernate-mapping>
  <class name="Employee" table="EMPLOYEE">
    <meta attribute="class-description">
      This class contains the employee detail.
    </meta>
    <id name="id" type="int" column="id">
      <generator class="native"/>
    </id>
    <set name="certificates" cascade="all">
      <key column="employee_id"/>
      <one-to-many class="Certificate"/>
    </set>
    <property name="firstName" column="first_name" type="string"/>
    <property name="lastName" column="last_name" type="string"/>
    <property name="salary" column="salary" type="int"/>
  </class>

  <class name="Certificate" table="CERTIFICATE">
    <meta attribute="class-description">
      This class contains the certificate records.
    </meta>
    <id name="id" type="int" column="id">
      <generator class="native"/>
    </id>
    <property name="name" column="certificate_name" type="string"/>
  </class>
</hibernate-mapping>
```

Step 4.Create Application Class

```
public class ManageEmployee {
  private static SessionFactory factory;
  public static void main(String[] args) {
    try{
      factory = new Configuration().configure().buildSessionFactory();
    }catch (Throwable ex) {
      System.err.println("Failed to create sessionFactory object." + ex);
      throw new ExceptionInInitializerError(ex);
    }
    ManageEmployee ME = new ManageEmployee();
    /* Let us have a set of certificates for the first employee  */
    HashSet set1 = new HashSet();
    set1.add(new Certificate(“PMP"));
    set1.add(new Certificate("MBA"));
    set1.add(new Certificate(" MCA "));

    /* Add employee records in the database */
    Integer empID1 = ME.addEmployee("Manoj", "Kumar", 4000, set1);

    ME.listEmployees();
```

Contd.

```
public Integer addEmployee(String fname, String lname,
                           int salary, Set cert){
    Session session = factory.openSession();
    Transaction tx = null;
    Integer employeeID = null;
    try{
      tx = session.beginTransaction();
      Employee employee = new Employee(fname, lname, salary);
      employee.setCertificates(cert);
      employeeID = (Integer) session.save(employee);
      tx.commit();
    }catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
    }finally {
      session.close();
    }
    return employeeID;
  }
```

```
public void listEmployees( ){
    Session session = factory.openSession();
    Transaction tx = null;
    try{
      tx = session.beginTransaction();
      List employees = session.createQuery("FROM Employee").list();
      for (Iterator iterator1 =
                  employees.iterator(); iterator1.hasNext();){
        Employee employee = (Employee) iterator1.next();
        System.out.print("First Name: " + employee.getFirstName());
        System.out.print("  Last Name: " + employee.getLastName());
        System.out.println("  Salary: " + employee.getSalary());
        Set certificates = employee.getCertificates();
        for (Iterator iterator2 =
                  certificates.iterator(); iterator2.hasNext();){
          Certificate certName = (Certificate) iterator2.next();
          System.out.println("Certificate: " + certName.getName());
        }
      }
      tx.commit();
    }catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
    }finally {
      session.close();
    }
  }
```

Output:

First Name: Manoj Last Name: Kumar Salary: 4000
Certificate: PMP
Certificate: MBA
Certificate: MCA

```
mysql> select * from employee;
+----+------------+-----------+--------+
| id | first_name | last_name | salary |
+----+------------+-----------+--------+
| 1  | Manoj      | Kumar     |   4000 |
+----+------------+-----------+--------+
1 row in set (0.00 sec)

mysql> select * from certificate;
+----+-----------------+------------+
| id | certificate_name | employee_id |
+----+-----------------+------------+
| 1  | MBA             |          1  |
| 2  | PMP             |          1  |
| 3  | MCA             |          1  |
+----+-----------------+------------+
3 rows in set (0.00 sec)
```

Java.util.SortedSet Example

- A **SortedSet** is a java collection that does not contain any duplicate element and elements are ordered using their natural ordering or by a comparator provided.

- **Changes in mapping file:**

```
<set name="certificates" cascade="all" sort="natural">
        <key column="employee_id"/>
        <one-to-many class="Certificate"/>
      </set>
```

- **Changes in application class**

```
TreeSet set1 = new TreeSet();
set1.add(new Certificate("PMP"));
…..
```

Output

First Name: Manoj Last Name: Kumar Salary: 4000
Certificate: MBA
Certificate: MCA
Certificate: PMP

Java.util.List Example

- A **List** is a java collection that stores elements in sequence and allow duplicate elements.

- **Changes in mapping file:**

```
<list name="certificates" cascade="all">
……
```

- **Changes in application class**

```
ArrayList set1 = new ArrayList();
set1.add(new Certificate("MCA"));
set1.add(new Certificate("MBA"));
set1.add(new Certificate("PMP"));
set1.add(new Certificate("MCA"));
```

Output

First Name: Manoj Last Name: Kumar Salary: 4000
Certificate: MBA
Certificate: MCA
Certificate: MCA
Certificate: PMP

Java.util.Collection Example

- A **Bag** is a java collection that stores elements without caring about the sequencing but allow duplicate elements in the list.
- A bag is a random grouping of the objects in the list.
- **Changes in mapping file:**

```
<bag name="certificates" cascade="all">
```

- **Changes in application class**

```
ArrayList set1 = new ArrayList();
set1.add(new Certificate("MCA"));
set1.add(new Certificate("MBA"));
set1.add(new Certificate("PMP"));
set1.add(new Certificate("MBA"));
```

Output

```
First Name: Manoj  Last Name: Kumar  Salary: 4000
Certificate: MBA
Certificate: PMP
Certificate: MCA
Certificate: MBA
```

Java.util.Map Example

- A **Map** is a java collection that stores elements in key-value pairs and does not allow duplicate elements in the list.

- **Changes in mapping file:**

  <map name="certificates" cascade="all"

- **Changes in application class**

```
HashMap set = new HashMap();
set.put("ComputerScience", new Certificate("MCA"));
……
public void listEmployees( ){
…….
Map ec = employee.getCertificates();
        System.out.println("Certificate: " +
          (((Certificate)ec.get("ComputerScience")).getName()));
……
}
```

Output:

First Name: Manoj Last Name: Kumar Salary: 4000
Certificate: MCA
…..

Java.util.SortedMap Example

- A **SortedMap** is a similar java collection as **Map** that stores elements in key-value pairs and provides a total ordering on its keys.
- Duplicate elements are not allowed in the map.

- **Changes in mapping file:**

```
<map name="certificates" cascade="all" sort="natural">
```

- **Changes in application class**

```
TreeMap set1 = new TreeMap();
     set1.put("ComputerScience", new Certificate("MBA"))
  ……
public void listEmployees( ){
…….
SortedMap<String, Certificate> map =
                                    employee.getCertificates();
         for(Map.Entry<String,Certificate> entry : map.entrySet()){
           System.out.print("\tCertificate Type: " +  entry.getKey());
           System.out.println(",  Name: " +
                          (entry.getValue()).getName());…..}
```

Output:

```
First Name: Manoj  Last Name: Kumar  Salary: 4000
Certificate Type: BusinessManagement,  Name: MBA
Certificate Type: ComputerScience,  Name: MCA
Certificate Type: ProjectManagement,  Name: PMP
```

Association Mappings

Mapping type	Description
Many-to-One	Mapping many-to-one relationship using Hibernate
One-to-One	Mapping one-to-one relationship using Hibernate
One-to-Many	Mapping one-to-many relationship using Hibernate
Many-to-Many	Mapping many-to-many relationship using Hibernate

One to Many Example

- Check java.util.Set example where one employee is associated with many certificates

Many to One Example

- **Changes in mapping file:**

```
<many-to-one name="address" column="address"
class="Address" not-null="true"/>
```

- **Changes in application class**

```
Address address = ME.addAddress(“Crossroad","Hyderabad","AP","532");

/* Add employee records in the database */
 Integer empID1 = ME.addEmployee("Manoj", "Kumar", 4000, address);

/* Add another employee record in the database */
Integer empID2 = ME.addEmployee("Dilip", "Kumar", 3000, address);
```

Output:

```
First Name: Manoj  Last Name: Kumar  Salary: 4000
Address
Street: Crossroad
City: Hyderabad
State: AP
Zipcode: 532
First Name: Dilip  Last Name: Kumar  Salary: 3000
Address
Street: Crossroad
City: Hyderabad
State: AP
Zipcode: 532
```

One to One Example

- **Changes in mapping file:**

```
<one-to-one name="address" column="address" class="Address"
not-null="true"/>
```

or

```
<many-to-one name="address" column="address" unique="true"
      class="Address" not-null="true"/>
```

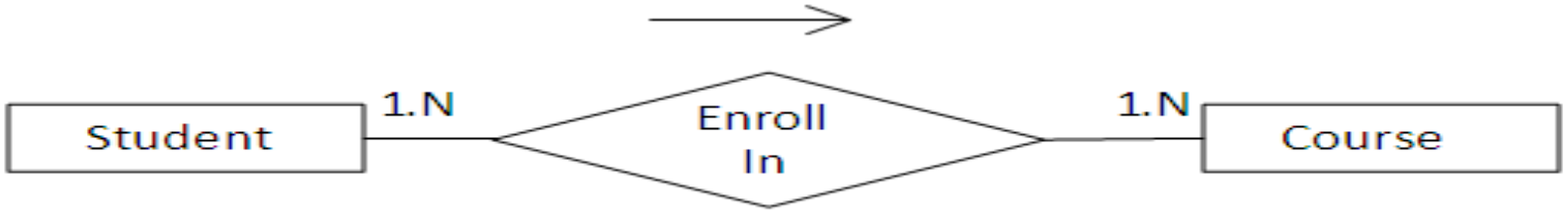
- **Changes in application class**

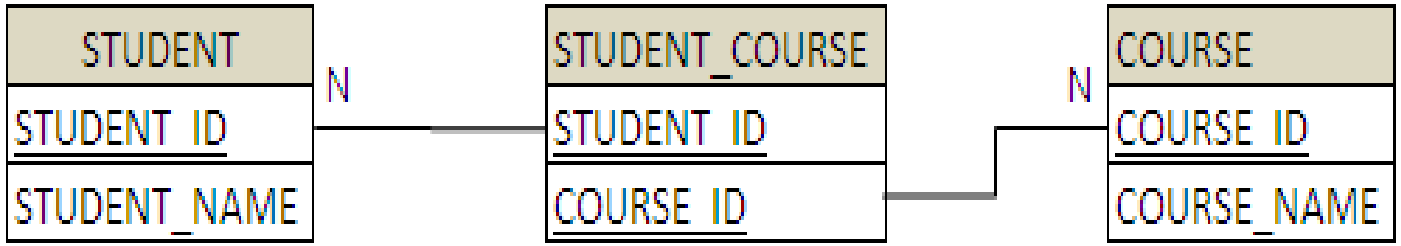
```
 Address address1 =
ME.addAddress(“Crossroad","Hyderabad","AP","532");

Integer empID1 = ME.addEmployee("Manoj", "Kumar", 4000, address1);

Address address2 =
ME.addAddress("Saharanpur","Ambehta","UP","111");
Integer empID2 = ME.addEmployee("Dilip", "Kumar", 3000, address2);
```

Output:

```
First Name: Manoj  Last Name: Kumar  Salary: 4000
 Address
 Street: Crossroad
  City: Hyderabad
  State: AP
  Zipcode: 532
First Name: Dilip  Last Name: Kumar  Salary: 3000
Address
Street: Saharanpur
City: Ambehta
  State: UP
  Zipcode: 111
```

Many to Many Example





- *Student.hbm.xml* is used to create the *STUDENT* and *STUDENT_COURSE* table.

```
<hibernate-mapping>
   <class name="com.abc.student.Student" table="STUDENT">
      <meta attribute="class-description">This class contains student details.</meta>
      <id name="studentId" type="long" column="STUDENT_ID">
         <generator class="native" />
      </id>
      <property name="studentName" type="string" length="100" not-null="true"
column="STUDENT_NAME" />
      <set name="courses" table="STUDENT_COURSE" cascade="all">
         <key column="STUDENT_ID" />
         <many-to-many column="COURSE_ID"  class="com.abc.student.Course" />
      </set>
   </class>
</hibernate-mapping>
```

- *Course.hbm.xml* is used to create the *COURSE* table

```
<hibernate-mapping>
<class name="com.abc.student.Course" table="COURSE">
        <meta attribute="class-description">
                This class contains course details.
        </meta>
        <id name="courseId" type="long" column="COURSE_ID">
                <generator class="native"/>
        </id>
        <property name="courseName" type="string" column="COURSE_NAME"/>
</class>
</hibernate-mapping>
```

Hibernate configuration File

```
<hibernate-configuration>
   <session-factory>
      <property name="hibernate.connection.driver_class"> org.hsqldb.jdbcDriver</property>
      <property name="hibernate.connection.url"> jdbc:hsqldb:hsql://localhost</property>
      <property name="hibernate.connection.username">sa</property>
      <property name="connection.password"></property>
      <property name="connection.pool_size">1</property>
      <property name="hibernate.dialect"> org.hibernate.dialect.HSQLDialect</property>
      <property name="show_sql">true</property>
      <property name="hbm2ddl.auto">create-drop</property>
      <mapping resource="com/vaannila/student/Student.hbm.xml"/>
      <mapping resource="com/vaannila/student/Course.hbm.xml"/>
   </session-factory>
</hibernate-configuration>
```

Create POJO Classes

```
public class Student implements java.io.Serializable {
        private long studentId;
        private String studentName;
        private Set<Course> courses = new HashSet<Course>(0);

        public Student() {
        }
        public Student(String studentName) {
                this.studentName = studentName;
        }
        public Student(String studentName, Set<Course> courses) {
                this.studentName = studentName;
                this.courses = courses;
        }   //getters and setters
        public Set<Course> getCourses() {
                return this.courses;
        }
        public void setCourses(Set<Course> courses) {
                this.courses = courses;
        }
}
```

Create POJO Classes (contd.)

```
public class Course implements java.io.Serializable {

        private long courseId;
        private String courseName;

        public Course() {
        }

        public Course(String courseName) {
                this.courseName = courseName;
        }

        //getters and setters
        public String getCourseName() {
                return this.courseName;
        }

        public void setCourseName(String courseName) {
                this.courseName = courseName;
        }
}
```

Create Application Main Class

```
public class Main {

        public static void main(String[] args) {

                Session session = HibernateUtil.getSessionFactory().openSession();
                Transaction transaction = null;
                try {
                        transaction = session.beginTransaction();

                        Set<Course> courses = new HashSet<Course>();
                        courses.add(new Course("Maths"));
                        courses.add(new Course("Computer Science"));

                        Student student1 = new Student(“Eswar", courses);
                        Student student2 = new Student("Joe", courses);
                        session.save(student1);
                        session.save(student2);

                        transaction.commit();
                        …….}
```

Output:

The *STUDENT* table has two records.

STUDENT_ID	STUDENT_NAME
1	Eswar
2	Joe

The *COURSE* table has two records.

COURSE_ID	COURSE_NAME
1	Computer Science
2	Maths

The *STUDENT_COURSE* table has four records to link the student and courses.

STUDENT_ID	COURSE_ID
1	1
1	2
2	1
2	2

- Each student has enrolled in the same two courses, this illustrates the many-to-many mapping.

Component Mapping Example

- An component is an object that is stored as an value rather than entity reference.
- This is mainly used if the dependent object doen't have primary key.
- It is used in case of composition (HAS-A relation), that is why it is termed as component.
- A single table would be used to keep the attributes contained inside the class variable

```
create table EMPLOYEE (
   id INT NOT  NULL auto_increment,
   first_name  VARCHAR(20) default NULL,
   last_name   VARCHAR(20) default NULL,
   salary      INT  default NULL,
   street_name VARCHAR(40) default NULL,
   city_name   VARCHAR(40) default NULL,
   state_name  VARCHAR(40) default NULL,
   zipcode     VARCHAR(10) default NULL,
   PRIMARY KEY (id)
);
```

Mapping File

```
<component name="address" class="Address">
        <property name="street" column="street_name" type="string"/>
        <property name="city" column="city_name" type="string"/>
        <property name="state" column="state_name" type="string"/>
        <property name="zipcode" column="zipcode" type="string"/>
 </component>
```

Output

```
mysql>
select id, first_name,salary, street_name, state_name from EMPLOYEE;
+----+------------+--------+-------------+------------+
| id | first_name | salary | street_name | state_name |
+----+------------+--------+-------------+------------+
|  1 | Manoj      |   5000 | Crossroad   | AP         |
|  2 | Dilip      |   3000 | Saharanpur  | UP         |
+----+------------+--------+-------------+------------+
```

Thank You