Unit:- -4

Hibernate

1. Introduction::

Hibernate is an Object-Relational Mapping(ORM) solution for JAVA and it raised as an open source persistent framework created by Gavin King in 2001. It is a powerful, high performance Object-Relational Persistence and Query service for any Java Application.

Hibernate maps Java classes to database tables and from Java data types to SQL data types and relieve the developer from 95% of common data persistence related programming tasks.

Hibernate sits between traditional Java objects and database server to handle all the work in persisting those objects based on the appropriate O/R mechanisms and patterns.

Hibernate is ORM tool

The Hibernate framework is an ORM tool for Java. It is very popular among the developers for creating world class enterprise web and JSE based applications. Hibernate mediates between Java objects and database store. It takes the Java objects from the program and then persists in the database. It takes the instruction from the Java program and translates into a SQL, executes the statements and then returns the result in the Java objects to the Java program.

Java ORM Frameworks:

There are several persistent frameworks and ORM options in Java. A persistent framework is an ORM service that stores and retrieves objects into a relational database.

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

2. Advantages/Limitations

Advantages of Hibernate

- Hibernate takes care of mapping Java classes to database tables using XML files and without writing any line of code.
- Provides simple APIs for storing and retrieving Java objects directly to and from the database.
- If there is change in Database or in any table then the only need to change XML file properties.
- Abstract away the unfamiliar SQL types and provide us to work around familiar Java Objects.
- Hibernate does not require an application server to operate.
- Manipulates Complex associations of objects of your database.
- Minimize database access with smart fetching strategies.
- Provides Simple querying of data.

Disadvantages of Hibernate

- **Lots of API to learn:** A lot of effort is required to learn Hibernate. So, not very easy to learn hibernate easily.
- **Debugging:** Sometimes debugging and performance tuning becomes difficult.
- **Slower than JDBC:** Hibernate is slower than pure JDBC as it is generating lots of SQL statements in runtime.
- **Not suitable for Batch processing:** It advisable to use pure JDBC for batch processing.

3. Hibernate Features

Hibernate is an Open Source persistence technology. Hibernate provide a way of mapping of applications class to the database table.

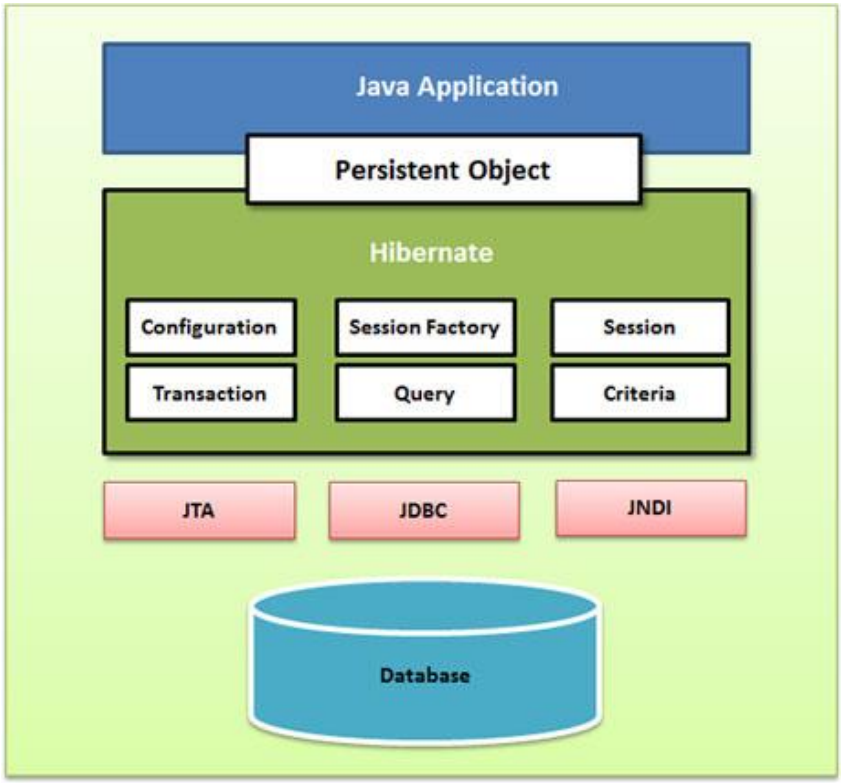
Main Features : It is ORM based and provide persistence management solution or persistent layer. It maps database tables to a class.

Here we are giving some features in points -

- Hibernate provides Object/Relational mappings. Here is different O/R mapping strategies as multiple-objects to single-row mapping,
  Polymorphic associations, bi-directional association, association filtering. It also provides XML mapping documents.
- It provides different object oriented query languages. Minimal object oriented Hibernate query language (HQL), native SQL queries,
  High object oriented concept of criteria.
- It provides transparent persistence without byte code processing.
- It introduces automatic Dirty Checking concept.
- It supports tough concept of composite keys.
- Automatic generation of primary key.
- Hibernate provides Dual-layer Cache Architecture.
- It provides session level cache and optional second-level cache.
- It introduces Lazy initialization.
- Hibernate provide outer join fetching.
- It supports optimistic locking with versioning.
- Optionally provide internal connection pooling and prepared statement caching.
- At system initialization time it generate SQL.
- It provide feature of J2EE integration.
- It supports JMX and JCA.

4. Hibernate Architecture:

The Hibernate architecture is layered to keep you isolated from having to know the underlying APIs. Hibernate makes use of the database and configuration data to provide persistence services (and persistent objects) to the application.



Hibernate uses various existing Java APIs, like JDBC, Java Transaction API(JTA), and Java Naming and Directory Interface (JNDI). JDBC provides a rudimentary level of abstraction of functionality common to relational databases, allowing almost any database with a JDBC driver to be supported by Hibernate. JNDI and JTA allow Hibernate to be integrated with J2EE application servers.

Following section gives brief description of each of the class objects involved in Hibernate Application Architecture.

Configuration Object:

The Configuration object is the first Hibernate object you create in any Hibernate application and usually created only once during application initialization. It represents a configuration or properties file required by the Hibernate. The Configuration object provides two keys components:

- **Database Connection:** This is handled through one or more configuration files supported by Hibernate. These files are **hibernate.properties** and **hibernate.cfg.xml**.
- **Class Mapping Setup**
  This component creates the connection between the Java classes and database tables..

SessionFactory Object:

Configuration object is used to create a SessionFactory object which inturn configures Hibernate for the application using the supplied configuration file and allows for a Session object to be instantiated. The SessionFactory is a thread safe object and used by all the threads of an application.

Session Object:

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.

Transaction Object:

A Transaction represents a unit of work with the database and most of the RDBMS supports transaction functionality. Transactions in Hibernate are handled by an underlying transaction manager and transaction (from JDBC or JTA).

This is an optional object and Hibernate applications may choose not to use this interface, instead managing transactions in their own application code.

Query Object:

Query objects use SQL or Hibernate Query Language (HQL) string to retrieve data from the database and create objects. A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

Criteria Object:

Criteria object are used to create and execute object oriented criteria queries to retrieve objects.

5. *Downloading Hibernate:*

It is assumed that you already have latest version of Java is installed on your machine. Following are the simple steps to download and install Hibernate on your machine.

- Make a choice whether you want to install Hibernate on Windows, or Unix and then proceed to the next step to download .zip file for windows and .tz file for Unix.
- Download the latest version of Hibernate from http://www.hibernate.org/downloads.

5.1 **Jars files of Hibernate**

For simplicity, all Java source and configuration files, as well as the required Hibernate and DataDirect Connect *for* JDBC libraries, will be placed in the same location. This will be our examples working directory.

Copy the following required Hibernate libraries from the Hibernate installation to the working directory. Some of the file names in your Hibernate installation will include numbers indicating the version of that file. The version numbers have been omitted from these file names below. These libraries must be on your classpath when you run the example.

lib/antlr.jar
lib/cglib.jar
lib/asm.jar
lib/asm-attrs.jars
lib/commons-collections.jar
lib/commons-logging.jar
lib/jta.jar
lib/dom4j.jar
lib/log4j.jar
hibernate3.jar

Copy the following DataDirect Connect *for* JDBC libraries from the driver installation to the working directory. These libraries must be on your classpath when you run the example.

lib/sqlserver.jar

5.2 **Mapping file in Hibernate**

An Object/relational mappings are usually defined in an XML document. This mapping file instructs Hibernate how to map the defined class or classes to the database tables.

Though many Hibernate users choose to write the XML by hand, a number of tools exist to generate the mapping document. Let us consider our previously defined POJO class whose objects will persist in the table defined in next section.

Employee.java

There would be one table corresponding to each object you are willing to provide persistence. Consider above objects need to be stored and retrieved into the following RDBMS table:

```
create table emp (
  id INT NOT NULL auto_increment,
  firstName VARCHAR(20) default NULL,
  lastName  VARCHAR(20) default NULL,
  PRIMARY KEY (id)
);
```

Based on the two above entities we can define following mapping file which instructs Hibernate how to map the defined class or classes to the database tables.

employee.hbm.xml

You should save the mapping document in a file with the format <classname>.hbm.xml. We saved our mapping document in the file Employee.hbm.xml. Let us see little detail about the mapping elements used in the mapping file:

- The mapping document is an XML document having **<hibernate-mapping>** as the root element which contains all the <class> elements.
- The **<class>** elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database table name is specified using the **table** attribute.
- The **<meta>** element is optional element and can be used to create the class description.
- The **<id>** element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.
- The **<generator>** element within the id element is used to automatically generate the primary key values. Set the **class** attribute of the generator element is set to **native** to let hibernate pick up either **identity, sequence** or **hilo** algorithm to create primary key depending upon the capabilities of the underlying database.
- The **<property>** element is used to map a Java class property to a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

There are other attributes and elements available which will be used in a mapping document and I would try to cover as many as possible while discussing other Hibernate related topics.

5.3 Hibernate Configuration file

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**.
I will consider XML formatted file **hibernate.cfg.xml** to specify required Hibernate properties in my examples. Most of the properties take their default values and it is not required to specify them in the property file unless it is really required. This file is kept in the root directory of your application's classpath.

Hibernate Properties:

Following is the list of important properties you would require to configure for a databases in a standalone situation:

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.
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.

If you are using a database along with an application server and JNDI then you would have to configure the following properties:

S.N.	Properties and Description
1	**hibernate.connection.datasource** The JNDI name defined in the application server context you are using for the application.
2	**hibernate.jndi.class** The InitialContext class for JNDI.
3	**hibernate.jndi.<JNDIpropertyname>** Passes any JNDI property you like to the JNDI *InitialContext*.
4	**hibernate.jndi.url** Provides the URL for JNDI.
5	**hibernate.connection.username** The database username.
6	**hibernate.connection.password** The database password.

Hibernate with MySQL Database:

MySQL is one of the most popular open-source database systems available today. Let us create**hibernate.cfg.xml** configuration file and place it in the root of your application's classpath.

hibernate.cfg.xml

The above configuration file includes **<mapping>** tags which are related to hibernate-mapping file and we will see in next chapter what exactly is a hibernate mapping file and how and why do we use it. Following is the list of various important databases dialect property type:

Database	Dialect Property
Microsoft SQL Server 2000	org.hibernate.dialect.SQLServerDialect
Microsoft SQL Server 2005	org.hibernate.dialect.SQLServer2005Dialect
Microsoft SQL Server 2008	org.hibernate.dialect.SQLServer2008Dialect
MySQL	org.hibernate.dialect.MySQLDialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle 11g	org.hibernate.dialect.Oracle10gDialect
Oracle 10g	org.hibernate.dialect.Oracle10gDialect
Oracle 9i	org.hibernate.dialect.Oracle9iDialect

6. Basic example of Hibernate

Here, we are going to create a simple example of hibernate application using eclipse IDE.
For creating the first hibernate application in Eclipse IDE, we need to follow following steps:

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

Employee.java

```
package www.demo.com;

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;
}


}
```

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="www.demo.com.Employee" table="emp">
   <id name="id">
    <generator class="assigned"></generator>
   </id>

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

  </class>

 </hibernate-mapping>
```

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.MySQLDialect</property>
      <property      name="connection.url">jdbc:mysql://localhost:3306/db1</property>
      <property name="connection.username">root</property>
      <property name="connection.password"></property>
      <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
   <mapping resource="employee.hbm.xml"/>
   </session-factory>

</hibernate-configuration>
```

StoreDeta.java

```
package www.demo.com;
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) {


   Configuration cfg=new Configuration();
   cfg.configure("hibernate.cfg.xml");

   SessionFactory factory=cfg.buildSessionFactory();

   Session session=factory.openSession();

   Transaction t=session.beginTransaction();

   Employee e1=new Employee();
   //e1.setId(1);
   e1.setFirstName("patel");
   e1.setLastName("jiya");

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

   t.commit();
   session.close();

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

}
}
```

** Program for insert, update, delete and select query.

StoreDeta1.java

```
package www.demo.com;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import java.util.*;

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
   SessionFactory factory=cfg.buildSessionFactory();
   Session session=factory.openSession();
   Transaction t;
   Employee e1=new Employee();

      //******** Insert*****

   Try
   {
      t=session.beginTransaction();
           //e1.setId(1);
           e1.setFirstName("sheladiya");
           e1.setLastName("disha");
           session.persist(e1);//persisting the object
           t.commit();//transaction is committed
```

```
            System.out.println("successfully saved");
    }
    catch(Exception e)
    {
            System.out.println(“Error in insert…..”);
    }


    //*****************update********

            Try
      {
      t=session.beginTransaction();
      e1 =(Employee)session.get(Employee.class, 1);
      e1.setFirstName("rrr");
                        session.update(e1);
      t.commit();
      System.out.println("successfully updated");
      }catch(Exception e)
      {
                        System.out.println(“Error in update”);
      }

    //*****************Delete********

      Try
      {
      t=session.beginTransaction();
      e1 =(Employee)session.get(Employee.class, 6);
      session.delete(e1);
      t.commit();
      System.out.println("successfully deleted");
      }catch(Exception e)
      {
                        System.out.println(“Error in delete”);
      }

     //*****************Select********

      Try
      {
      t=session.beginTransaction();
      List employees = session.createQuery("FROM Employee").list();
      Iterator i =employees.iterator();
      while(i.hasNext())
      {
        e1 = (Employee) i.next();
        System.out.print("Id: " + e1.getId());
        System.out.print(" first name " + e1.getFirstName());
        System.out.println("last name " + e1.getLastName());
      }
      t.commit();
            }catch(Exception e)
      {
                        System.out.println(“Error in select”);
      }

    session.close();

}
}
```

7. Hibernate Annotations

So far you 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.

Hibernate Annotations is the powerful way to provide the metadata for the Object and Relational Table mapping. All the metadata is clubbed into the POJO java file along with the code this helps the user to understand the table structure and POJO simultaneously during the development.

If you going to make your application portable to other EJB 3 compliant ORM applications, you must use annotations to represent the mapping information but still if you want greater flexibility then you should go with XML-based mappings.

Environment Setup for Hibernate Annotation

First of all you would have to make sure that you are using JDK 5.0 otherwise you need to upgrade your JDK to JDK 5.0 to take advantage of the native support for annotations.

Second, you will need to install the Hibernate 3.x annotations distribution package, available from the sourceforge: (Download Hibernate Annotation) and copy **hibernate-annotations.jar, lib/hibernate-comons-annotations.jar** and **lib/ejb3-persistence.jar** from the Hibernate Annotations distribution to your CLASSPATH

Annotated Class Example:

As I mentioned above while working with Hibernate Annotation all the metadata is clubbed into the POJO java file along with the code this helps the user to understand the table structure and POJO simultaneously during the development.

Consider we are going to use following EMPLOYEE table to store our objects:

```
create table employee (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   e_city  VARCHAR(20) default NULL,
   PRIMARY KEY (id)
);
```

Following is the mapping of Employee class with annotations to map objects with the defined EMPLOYEE table:

Annotation_Example

Employee.java

```
package www.patel.sankul;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name = "employee")
public class Employee implements Serializable
{
   @Id
   @GeneratedValue
   @Column(name="e_id")
   private int id;

   @Column(name="e_name")
   private String empName;

   @Column(name="e_city")
   private String empCity;

   public int getId() {
         return id;
   }

   public void setId(int id) {
         this.id = id;
   }

   public String getEmpName() {
         return empName;
   }

   public void setEmpName(String empName) {
         this.empName = empName;
   }

   public String getEmpCity() {
         return empCity;
   }

   public void setEmpCity(String empCity) {
         this.empCity = empCity;
   }
}
```

Hibernate.cfg.xml

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

```
<hibernate-configuration>
<session-factory>
<property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/db1</property>
<property name="hibernate.connection.username">root</property>
<property name="hibernate.connection.password"></property>
<property name="hibernate.connection.pool_size">10</property>
<property name="dialect">org.hibernate.dialect.MySQLDialect</property>
<mapping class="www.patel.sankul.Employee" />
</session-factory>
</hibernate-configuration>
```

ShowData.java

```
package www.patel.sankul;

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

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

   Configuration cfg=new Configuration();
   cfg.configure("hibernate.cfg.xml");

   SessionFactory factory=cfg.buildSessionFactory();

   Session session=factory.openSession();

   Transaction t=session.beginTransaction();

   Employee e1=new Employee();
   //e1.setId(1);
   e1.setEmpName("jiya");
   e1.setEmpCity("amreli");

   session.persist(e1);

   t.commit();
   session.close();

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

} }
```

Hibernate detects that the @Id annotation is on a field and assumes that it should access properties on an object directly through fields at runtime. If you placed the @Id annotation on the getId() method, you would enable access to properties through getter and setter methods by default. Hence, all other annotations are also placed on either fields or getter methods, following the selected strategy. Following section will explain the annotations used in the above class.

@Entity Annotation:

The EJB 3 standard annotations are contained in the **javax.persistence** package, so we import this package as the first step. Second 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.

The @Table annotation provides four attributes, allowing you to override the name of the table, its catalogue, and its schema, and enforce unique constraints on columns in the table. For now we are using just table name which is EMPLOYEE.

@Id and @GeneratedValue Annotations:

Each entity bean will have a primary key, which you annotate on the class with the **@Id** annotation. The primary key can be a single field or a combination of multiple fields depending on your table structure.

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** which I'm not going to discuss here, so let us use only default the default key generation strategy. Letting Hibernate determine which generator type to use makes your code portable between different databases.

@Column Annotation:

The @Column annotation is used to specify the details of the column to which a field or property will be mapped. You can use column annotation with the following most commonly used attributes:

- **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.

8. Hibernet Inheritance

Java is an object oriented language. It is possible to implement Inheritance in Java. Inheritance is one of the most visible facets of Object-relational mismatch. Hibernate can help you map such Objects with relational tables. But you need to choose certain mapping strategy based on your needs.

8.1. Inheritance Annotation

Hibernate supports the three basic inheritance mapping strategies:

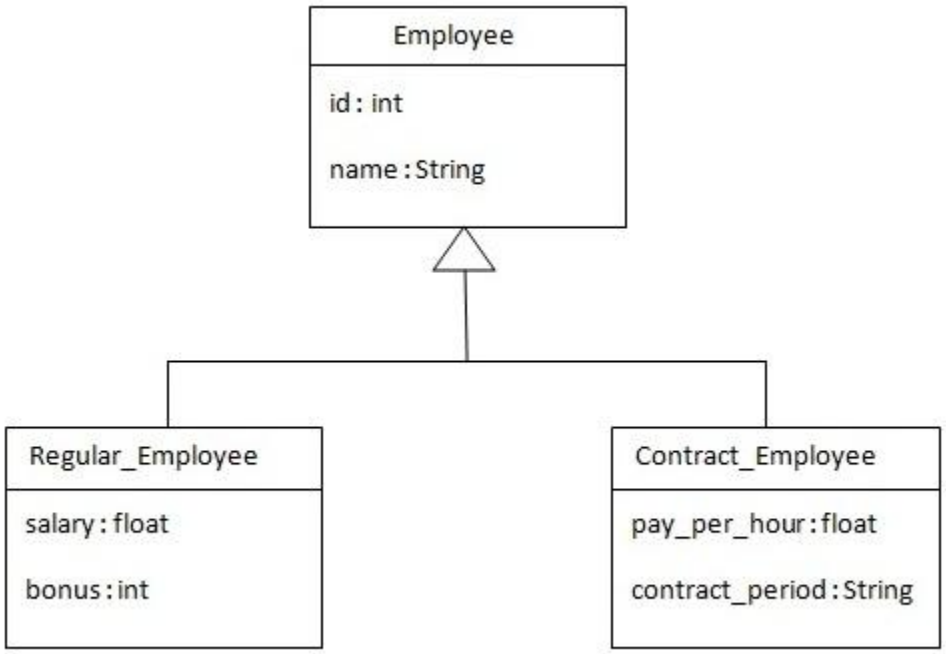
1. Table per class hierarchy
2. Table per sub class
3. Table per concrete class

1) Table Per Class Hierarchy using Annotation

In the previous page, we have mapped the inheritance hierarchy with one table only using xml file. Here, we are going to perform this task using annotation. You need to use @Inheritance(strategy=InheritanceType.SINGLE_TABLE), @DiscriminatorColumn and @DiscriminatorValue annotations for mapping table per hierarchy strategy.

In case of table per hierarchy, only one table is required to map the inheritance hierarchy. Here, an extra column (also known as **discriminator column**) is created in the table to identify the class.

Let's see the inheritance hierarchy:



There are three classes in this hierarchy. Employee is the super class for Regular_Employee and Contract_Employee classes.

The table structure for this hierarchy is as shown below:

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(10,0)	No	-	1
TYPE	VARCHAR2(255)	No	-	-
NAME	VARCHAR2(255)	Yes	-	-
SALARY	FLOAT	Yes	-	-
BONUS	NUMBER(10,0)	Yes	-	-
PAY_PER_HOUR	FLOAT	Yes	-	-
CONTRACT_DURATION	VARCHAR2(255)	Yes	-	-
				1 - 7

Example of Hibernate Table Per Hierarchy using Annotation

1) Employee.java

```
package www.annotation.com;

import javax.persistence.*;

@Entity
@Table(name = "employee")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="type",discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue(value="employee")

public class Employee
{
	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)

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

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

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

2) Regular_Employee.java

```
package www.annotation.com;
import javax.persistence.*;

@Entity
@DiscriminatorValue("regularemployee")
public class Regular_Employee extends Employee
{

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

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

	//setters and getters
	public void setSalary(float salary)
	{
		this.salary=salary;
	}
	public float getSalary()
	{
		return salary;
	}
	public void setBonus(int bonus)
	{
		this.bonus=bonus;
	}
	public int getBonus()
	{
		return bonus;
	}

}
```

3) Contract_Employee.java

```
package www.annotation.com;

import javax.persistence.*;

@Entity
@DiscriminatorValue("contractemployee")
public class Contract_Employee extends Employee{

   @Column(name="pay_per_hour")
   private float pay_per_hour;

   @Column(name="contract_duration")
   private String contract_duration;


   //setters and getters
   public void setPayPerHour(float pay_per_hour)
	{
		this.pay_per_hour=pay_per_hour;
```

```
        }
        public float getsetPayPerHou()
        {
                return pay_per_hour;
        }
        public void setContactDuration(String cd)
        {
                this.contract_duration=cd;
        }
        public String getContactDuration()
        {
                return contract_duration;
        }
}
```

4) 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">

<!-- Generated by MyEclipse Hibernate Tools.                   -->
<hibernate-configuration>

   <session-factory>
      <property name="hbm2ddl.auto">update</property>
      <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
      <property name="connection.url">jdbc:mysql://localhost:3306/db1</property>
      <property name="connection.username">root</property>
      <property name="connection.password"></property>
      <property name="connection.driver_class">com.mysql.jdbc.Driver</property>

      <mapping class="www.annotation.com.Employee"/>
      <mapping class="www.annotation.com.Contract_Employee"/>
      <mapping class="www.annotation.com.Regular_Employee"/>
   </session-factory>

</hibernate-configuration>
```

5) StoreTest.java

```
package www.annotation.com;
import org.hibernate.*;
import org.hibernate.cfg.*;

public class StoreTest
{
public static void main(String[] args)
{
   AnnotationConfiguration cfg=new AnnotationConfiguration();
   Session session=cfg.configure("hibernate.cfg.xml").buildSessionFactory().openSession();

   Transaction t=session.beginTransaction();

   Employee e1=new Employee();

   e1.setName("sonoo");
```

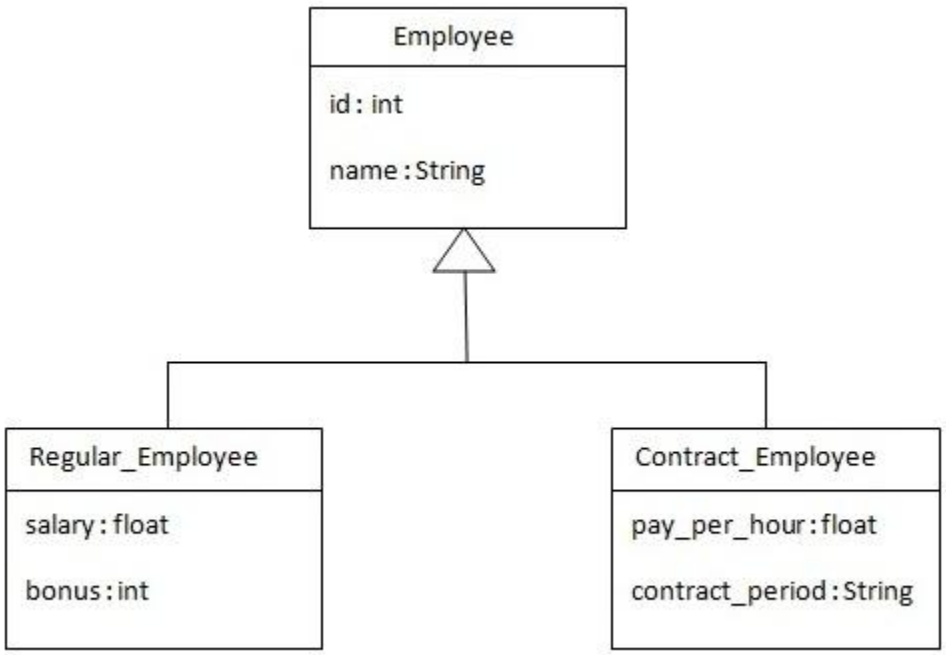
```
    Regular_Employee e2=new Regular_Employee();
    e2.setName("Vivek Kumar");
    e2.setSalary(50000);
    e2.setBonus(5);

    Contract_Employee e3=new Contract_Employee();
    e3.setName("Arjun Kumar");
    e3.setPayPerHour(1000);
    e3.setContactDuration("15 hours");

    session.persist(e1);
    session.persist(e2);
    session.persist(e3);

    t.commit();
    session.close();
    System.out.println("successfully inserted…..");
}
}
```

2) Table Per Subclass using Annotation

As we have specified earlier, in case of table per subclass strategy, tables are created as per persistent classes but they are reated using primary and foreign key. So there will not be duplicate columns in the relation.

We need to specify **@Inheritance(strategy=InheritanceType.JOINED)** in the parent class and **@PrimaryKeyJoinColumn** annotation in the subclasses.

Let's see the hierarchy of classes that we are going to map.



The table structure for each table will be as follows:

Table structure for Employee class

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(10,0)	No	-	1
NAME	VARCHAR2(255)	Yes	-	-
				1 - 2

Table structure for Regular_Employee class

Column Name	Data Type	Nullable	Default	Primary Key
EID	NUMBER(10,0)	No	-	1
SALARY	FLOAT	Yes	-	-
BONUS	NUMBER(10,0)	Yes	-	-
				1 - 3

Table structure for Contract_Employee class

Column Name	Data Type	Nullable	Default	Primary Key
EID	NUMBER(10,0)	No	-	1
PAY_PER_HOUR	FLOAT	Yes	-	-
CONTRACT_DURATION	VARCHAR2(255)	Yes	-	-
				1 - 3

Example of Table per subclass class using Annotation

In this example we are creating the three classes and provide mapping of these classes in the employee.hbm.xml file.

1) Employee.java

```
package www.annotation.com;

import javax.persistence.*;

@Entity
@Table(name = "employee")
@Inheritance(strategy=InheritanceType.JOINED)

public class Employee {
@Id
@GeneratedValue(strategy=GenerationType.AUTO)

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

@Column(name = "name")
private String name;
```

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

}
```

2) Regular_Employee.java

```
package www.annotation.com;

import javax.persistence.*;

@Entity
@Table(name="regular_employee")
@PrimaryKeyJoinColumn(name="id")

public class Regular_Employee extends Employee
{

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

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

	//setters and getters
	public void setSalary(float salary)
	{
		this.salary=salary;
	}
	public float getSalary()
	{
		return salary;
	}
	public void setBonus(int bonus)
	{
		this.bonus=bonus;
	}
	public int getBonus()
	{
		return bonus;
	}

}
```

3) Contract_Employee.java

```
package www.annotation.com;
import javax.persistence.*;

@Entity
@Table(name="contract_employee")
@PrimaryKeyJoinColumn(name="id")

public class Contract_Employee extends Employee{

   @Column(name="pay_per_hour")
   private float pay_per_hour;

   @Column(name="contract_duration")
   private String contract_duration;

   //setters and getters
   public void setPayPerHour(float pay_per_hour)
         {
               this.pay_per_hour=pay_per_hour;
         }
         public float getsetPayPerHou()
         {
               return pay_per_hour;
         }
         public void setContactDuration(String cd)
         {
               this.contract_duration=cd;
         }
         public String getContactDuration()
         {
               return contract_duration;
         }
}
```

4) 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">

<!-- Generated by MyEclipse Hibernate Tools.                   -->
<hibernate-configuration>

   <session-factory>
      <property name="hbm2ddl.auto">update</property>
      <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
      <property name="connection.url">jdbc:mysql://localhost:3306/db1</property>
      <property name="connection.username">root</property>
      <property name="connection.password"></property>
      <property name="connection.driver_class">com.mysql.jdbc.Driver</property>

      <mapping class="www.annotation.com.Employee"/>
      <mapping class="www.annotation.com.Contract_Employee"/>
      <mapping class="www.annotation.com.Regular_Employee"/>
   </session-factory>
```

```
</hibernate-configuration>
```

5) StoreData.java

```
package www.annotation.com;

import org.hibernate.*;
import org.hibernate.cfg.*;

public class StoreData
{
public static void main(String[] args)
{
   AnnotationConfiguration cfg=new AnnotationConfiguration();
   Session session=cfg.configure("hibernate.cgf.xml").buildSessionFactory().openSession();

   Transaction t=session.beginTransaction();

   Employee e1=new Employee();

   e1.setName("sonoo");

   Regular_Employee e2=new Regular_Employee();
   e2.setName("Vivek Kumar");
   e2.setSalary(50000);
   e2.setBonus(5);

   Contract_Employee e3=new Contract_Employee();
   e3.setName("Arjun Kumar");
   e3.setPayPerHour(1000);
   e3.setContactDuration("15 hours");

   session.persist(e1);
   session.persist(e2);
   session.persist(e3);

   t.commit();
   session.close();
   System.out.println("success");
}
}
```

3) Table Per Concrete class using Annotation

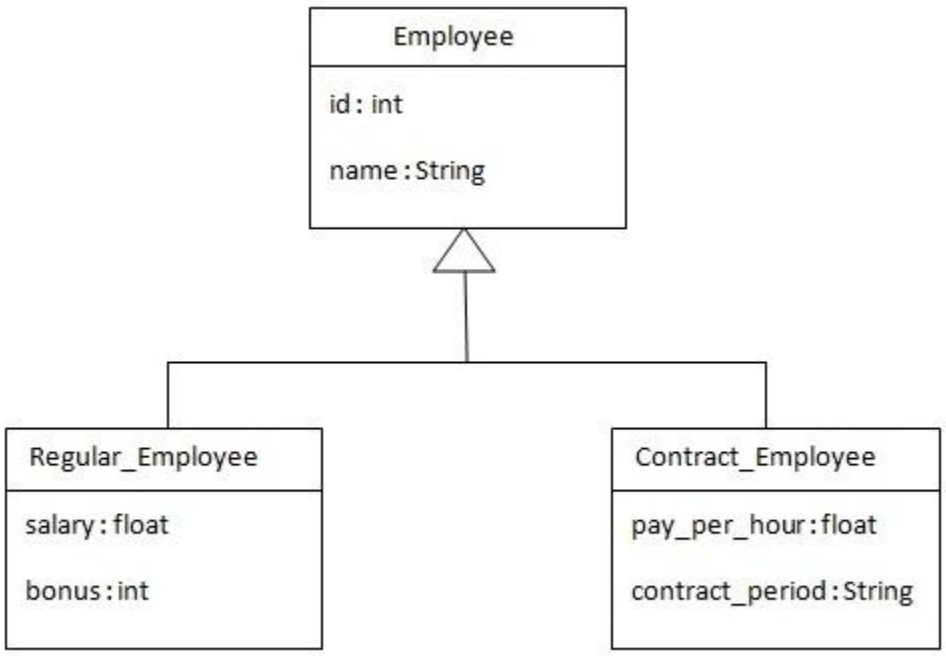
In case of Table Per Concrete class, tables are created per class. So there are no nullable values in the table. Disadvantage of this approach is that duplicate columns are created in the subclass tables.

Here, we need to use @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS) annotation in the parent class and @AttributeOverrides annotation in the subclasses.

@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS) specifies that we are using table per concrete class strategy. It should be specified in the parent class only.

@AttributeOverrides defines that parent class attributes will be overriden in this class. In table structure, parent class table columns will be added in the subclass table.

The class hierarchy is given below:



The table structure for each table will be as follows:

Table structure for Employee class

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(10,0)	No	-	1
NAME	VARCHAR2(255)	Yes	-	-
				1 - 2

Table structure for Regular_Employee class

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(10,0)	No	-	1
NAME	VARCHAR2(255)	Yes	-	-
SALARY	FLOAT	Yes	-	-
BONUS	NUMBER(10,0)	Yes	-	-
				1 - 4

Table structure for Contract_Employee class

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(10,0)	No	-	1
NAME	VARCHAR2(255)	Yes	-	-
PAY_PER_HOUR	FLOAT	Yes	-	-
CONTRACT_DURATION	VARCHAR2(255)	Yes	-	-
				1 - 4

Example of Table per concrete class

1) Employee.java

```
package www.annotation.com;

import javax.persistence.*;

@Entity
@Table(name = "employee")
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)

public class Employee {
@Id
@GeneratedValue(strategy=GenerationType.AUTO)

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

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

//setters and getters
public void setId(int id)
{
	this.id=id;
}
public int getId()
{
	return id;
}
public void setName(String name)
{
	this.name=name;
}
public String getName()
{
	return name;
}

}
```

2) Regular_Employee.java

```
package www.annotation.com;

import javax.persistence.*;

@Entity
@Table(name="regular_employee")
@AttributeOverrides({
   @AttributeOverride(name="id", column=@Column(name="id")),
   @AttributeOverride(name="name", column=@Column(name="name"))
})


public class Regular_Employee extends Employee
{

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

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

        //setters and getters
        public void setSalary(float salary)
        {
                this.salary=salary;
        }
        public float getSalary()
        {
                return salary;
        }
        public void setBonus(int bonus)
        {
                this.bonus=bonus;
        }
        public int getBonus()
        {
                return bonus;
        }

}
```

3) Contract_Employee.java

```
package www.annotation.com;

import javax.persistence.*;

@Entity
@Table(name="contract_employee")
@AttributeOverrides({
   @AttributeOverride(name="id", column=@Column(name="id")),
   @AttributeOverride(name="name", column=@Column(name="name"))
})


public class Contract_Employee extends Employee{

   @Column(name="pay_per_hour")
```

```
    private float pay_per_hour;

    @Column(name="contract_duration")
    private String contract_duration;


    //setters and getters
    public void setPayPerHour(float pay_per_hour)
            {
                    this.pay_per_hour=pay_per_hour;
            }
            public float getsetPayPerHou()
            {
                    return pay_per_hour;
            }
            public void setContactDuration(String cd)
            {
                    this.contract_duration=cd;
            }
            public String getContactDuration()
            {
                    return contract_duration;
            }
}
```

4) 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">

<!-- Generated by MyEclipse Hibernate Tools.                   -->
<hibernate-configuration>

    <session-factory>
        <property name="hbm2ddl.auto">update</property>
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="connection.url">jdbc:mysql://localhost:3306/db2</property>
        <property name="connection.username">root</property>
        <property name="connection.password"></property>
        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>

        <mapping class="www.annotation.com.Employee"/>
        <mapping class="www.annotation.com.Contract_Employee"/>
        <mapping class="www.annotation.com.Regular_Employee"/>
    </session-factory>

</hibernate-configuration>
```

5) StoreData.java

```
package www.annotation.com;

import org.hibernate.*;
import org.hibernate.cfg.*;

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

```
AnnotationConfiguration cfg=new AnnotationConfiguration();
Session session=cfg.configure("hibernate.cgf.xml").buildSessionFactory().openSession();
Transaction t=session.beginTransaction();

Employee e1=new Employee();

e1.setName("sonoo");

Regular_Employee e2=new Regular_Employee();
e2.setName("Vivek Kumar");
e2.setSalary(50000);
e2.setBonus(5);

Contract_Employee e3=new Contract_Employee();
e3.setName("Arjun Kumar");
e3.setPayPerHour(1000);
e3.setContactDuration("15 hours");

session.persist(e1);
session.persist(e2);
session.persist(e3);

t.commit();
session.close();
System.out.println("success");
}
}
```

9. 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 session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed them as needed. 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.

A Session instance is serializable if its persistent classes are serializable. A typical transaction should use the following idiom:

```
Session session = factory.openSession();
Transaction tx = null;
try {
  tx = session.beginTransaction();
  // do some work
```

```
   ...
   tx.commit();
}
catch (Exception e) {
   if (tx!=null) tx.rollback();
   e.printStackTrace();
}finally {
   session.close();
}
```

If the Session throws an exception, the transaction must be rolled back and the session must be discarded.

Session Interface Methods:

There are number of methods provided by the **Session** interface but I'm going to list down few important methods only, which we will use in this tutorial. You can check Hibernate documentation for a complete list of methods associated with **Session** and **SessionFactory**.

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.
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.
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.