Hibernate-JavaConfiguration-Reference-Manual

1) what is Hibernate	**2**
2)Difference b/w JDBC and Hibernate	**2**
3)JDBC Configuration	**3**
4)Hibernate Configuration	**5**
5)Hibernate First Application	**7**
6)Hibernate Persistence Class	**8**
7)Hibernate CRUD Operations	**9**
8)Hibernate Life Cycle	**12**
9)Hibernate Embedded Objects and List	**16**
@Embeddable and @Embedded together.	17
Attribute Overrides	19
ElementCollection	21
Embedded – Id	24
10)Hibernate HQL	**26**
11)Hibernate HCQL	**31**
12)Hibernate Named Queries	**36**
13)Hibernate SQL	**39**
14)Hibernate Transaction Management	**43**
15)Hibernate Inheritance Mapping	**46**
Table Per Hierarchy	46
Table Per Class	49
16)Hibernate Relational Mapping	**51**
17)Hibernate Loading	**51**
18)Hibernate Caching	**51**

By Ratna Srinivasa Rao Karicherla

1) what is Hibernate

Hibernate is a Java framework that simplifies the development of Java application to interact with the database.
It is an open source, lightweight, ORM (Object Relational Mapping) tool.
Hibernate implements the specifications of JPA (Java Persistence API) for data persistence.
Java Persistence API (JPA) is a Java specification that provides certain functionality and standard to ORM tools. The **javax.persistence** package contains the JPA classes and interfaces.

2)Difference b/w JDBC and Hibernate

Hibernate	**JDBC**
As Hibernate is set of Objects, you don't need to learn SQL language. You can treat TABLE as an Object. Only Java knowledge is needed.	In the case of JDBC you need to learn SQL.
Hibernate supports Query cache and It will provide the statistics about your query and database status.	JDBC Not provides any statistics.
Hibernate is database independent, your code will work for all ORACLE,MySQL ,SQLServer etc.	In the case of JDBC, the query must be database specific.

You will get benefit of Cache. Hibernate supports two levels of cache. First level and 2nd level. So, you can store your data into Cache for better performance.	In the case of JDBC you need to implement your java cache.
No need to create any connection pool in case of Hibernate. You can use c3p0.	In case of JDBC you need to write your own connection pool

3)JDBC Configuration

To connect a Java application with Any Database, we need to follow 5 following steps.
In this example we are using MySQL as the database. So, we need to know following information for the MySQL database:

1. **Driver class:** The driver class for the MySQL database is **com.mysql.jdbc.Driver**.
2. **Connection URL:** The connection URL for the MySQL database is **jdbc:mysql://localhost:4444/test**
3. Where **jdbc** is the API, **mysql** is the database, **localhost** is the server name on which mysql is running, we may also use an IP address, **4444** is the port number and **test** is the database name.
4. We may use any database, in such case, we need to replace the **test** with our database name.
5. **Username:** The default username for the mysql database is **root**.
6. **Password:** It is the password given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.

Example to Connect to JDBC using JAVA

```
import java.sql.*;
class MysqlCon{
public static void main (String args[]){
try {

//loading driver class
Class.forName("com.mysql.jdbc.Driver");

//getting Connection Object by passing required Information
Connection con=DriverManager.getConnection(
"jdbc:mysql://localhost:4444/test","root","root");
//here test is database name, root is username and password

//getting statement or prepared statement object to connect database
Statement stmt=con.createStatement();

//querying with executeQuery or executeUpdate and getting resultset
ResultSet rs=stmt.executeQuery("select * from emp");

//processing result
while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));

//closing the connection
con.close();
} catch (Exception e){ System.out.println(e);}
}
}
```

4)Hibernate Configuration

In the above step we have connected with database using jdbc by passing driver class,url,username and password similarly in hibernate also we need to connect with database instead of connection we will get SessionFactory and from that we will get Session Object here.

```
public class HibernateConfiguration {

        private static SessionFactory sessionFactory;

        public static SessionFactory getSessionFactory() {
        if (sessionFactory == null) {
        try {

                // creating configuration class
                Configuration configuration = new Configuration ();

                // Hibernate settings equivalent to hibernate.cfg.xml's properties
                Properties settings = new Properties ();
            settings.put(Environment.DRIVER, "com.mysql.cj.jdbc.Driver");
            settings.put(Environment.URL, "jdbc:mysql://localhost:4444/test");
            settings.put(Environment.USER, "root");
            settings.put(Environment.PASS, "root");

settings.put(Environment.DIALECT,"org.hibernate.dialect.MySQL5Dialect");
                // This property makes Hibernate generate the appropriate SQL for
the chosen
                // database.
            settings.put(Environment.SHOW_SQL, "true");
            settings.put(Environment.CURRENT_SESSION_CONTEXT_CLASS,
"thread");
            settings.put(Environment.HBM2DDL_AUTO, "update");
                // setting properties to configuration class
            configuration.setProperties(settings);
                // adding beans to configuration
            configuration.addAnnotatedClass(Student.class);
            configuration.addAnnotatedClass(Laptop.class);
            configuration.addAnnotatedClass(Employee.class);


                // creating service registry by passing configuration
            ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()
                .applySettings(configuration.getProperties()).build();
                // getting sessionFactory object
            sessionFactory = configuration.buildSessionFactory(serviceRegistry);
        } catch (Exception e) {
            e.printStackTrace();
        }
        }
        return sessionFactory;
        }

}
```

5)Hibernate First Application

By Using Above Configuration, we can get our required session object and perform all our crud operations in hibernate before discussing further just a sample hibernate application to interact with the database.

```
public class BulkRecordsInsertUtility {

        public static void main (String [] args) {
      SessionFactory sessionFactory =
HibernateConfiguration.getSessionFactory();

//getting session object
        Session session = sessionFactory.openSession();
//starting transaction
      session.beginTransaction();
        Random r = new Random ();
        for (int i = 1; i <= 15; i++) {
        Employee e = new Employee ();
         e.setAge(r.nextInt(100));
         e.setSalary(r.nextInt(50000));
         e.setName("Emp :" + i);
//saving into database
         session.save(e);
         }
//commit the transaction otherwise no record get inserted
      session.getTransaction().commit();
//close the session
      session.close()
          }

}
```

6)Hibernate Persistence Class

We need a class to be mapped with respective to the table of our database and that class called as entity or persistent class.
Following example shows how to map a normal pojo as an entity.

```
@Entity // annotation marks this class as an entity.
@Table (name = "employee") // annotation specifies the table name where
data of this entity is to be persisted. If you don't use @Table annotation,
hibernate will use the class name as the table name by default.
public class Employee {

          @Id //Marks the field as primary key
          @GeneratedValue //autogenerates the value used only in case of int not
applicable for string
          private Integer empId;

          @Column // annotation specifies the details of the column for this
property or field. If @Column annotation is not specified, property name will be
used as the column name by default.
```

```
        private String name;

        @Column(name=”sal”)
        private Integer salary;

        @Column //no need to specify this attribute if table and class field is
same
        private Integer age;

//getters and setters
}
```

7)Hibernate CRUD Operations

In this we will see some crud operations related with Session Object

```
public class HibernateCrudOperations {

        public static void main (String [] args) {

      SessionFactory sessionFactory =
HibernateConfiguration.getSessionFactory();
        Session session = sessionFactory.openSession();
      session.beginTransaction();
      insertRecordsIntoShop(session, 15, false);
      System.out.println(getRequiredShopByID(session, 142, false));
      deleteRequiredRecordFromShop(session, 142, false);
      System.out.println(updateRequiredShopByID(session, 153, true));
```

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

        }

        public static void insertRecordsIntoShop(Session session, int count,
boolean insert) {
        if (insert) {
         ShopDAO.insertRequiredNumOfRecords(count, session);
        }
        }

        public static Shop getRequiredShopByID(Session session, int id, boolean
retrieve) {
        Shop shop = null;
        if (retrieve) {
        shop = ShopDAO.getRequiredShopById(id, session);
        }
        return shop;
        }

        public static void deleteRequiredRecordFromShop(Session session, int id,
boolean delete) {
        if (delete) {
         ShopDAO.deleteRequiredShopById(id, session);
        }
        }

        public static Shop updateRequiredShopByID(Session session, int id,
boolean update) {
        Shop shop = null;
        if (update) {
```

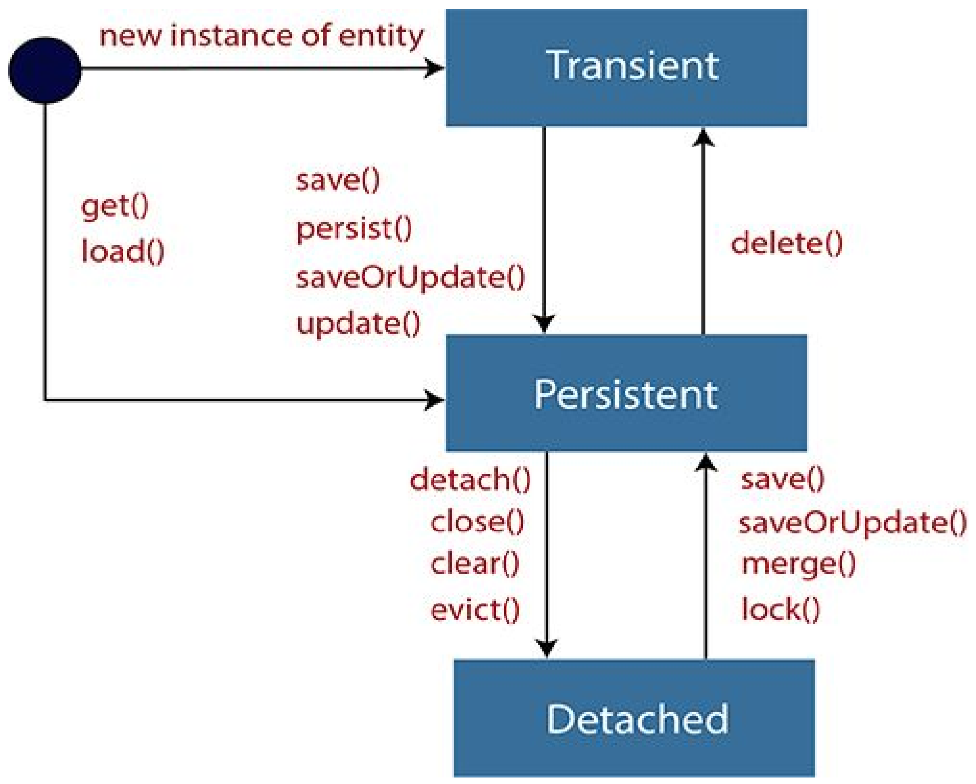
```
        shop = ShopDAO.updateRequiredShopById(id, session);
        }
        return shop;
        }


public class ShopDAO {

        public static void insertRequiredNumOfRecords(int numOfRecords,
Session session) {
        Random r = new Random ();
        for (int i = 1; i <= numOfRecords; i++) {
        Shop s = new Shop();
         s.setCity("shop city " + i);
         s.setName("Shop Name " + i);
         s.setRevenue(r.nextInt(1000000));
         session.save(s);

        }

        }

        public static Shop getRequiredShopById(int shopId, Session session) {
        Shop shop = session.get(Shop.class, shopId);
        return shop;
        }

        public static void deleteRequiredShopById(int shopId, Session session) {
        Shop shop = session.get(Shop.class, shopId);
      session.delete(shop);
        }
        public static Shop updateRequiredShopById(int shopId, Session session) {
        Shop shop = session.get(Shop.class, shopId);
      shop.setName("Ratna General Stores");
      session.update(shop);
        return shop;
        }

}

@Entity
@Table(name = "shop")
public class Shop {
        @Id
        @GeneratedValue
        private int shopId;
        private String city;
        private String name;
        private float revenue;
//getters & Setters
        }
```

8)Hibernate Life Cycle

In Hibernate, either we create an object of an entity and save it into the database, or we fetch the data of an entity from the database.
Here, each entity is associated with the lifecycle.

The entity object passes through the different stages of the life cycle.
The Hibernate lifecycle contains the following states: -

- Transient state
- Persistent state
- Detached state



Transient state

- The transient state is the initial state of an object.
- Once we create an instance of POJO class, then the object enters in the transient state.
- Here, an object is not associated with the Session. So, the transient state is not related to any database.
- Hence, modifications in the data don't affect any changes in the database.
- The transient objects exist in the heap memory. They are independent of Hibernate.

Persistent state

- As soon as the object associated with the Session, it entered in the persistent state.
- Hence, we can say that an object is in the persistence state when we save or persist it.
- Here, each object represents the row of the database table.
- So, modifications in the data make changes in the database.

Detached State

- Once we either close the session or clear its cache, then the object enters the detached state.
- As an object is no more associated with the Session, modifications in the data don't affect any changes in the database.
- However, the detached object still has a representation in the database.
- If we want to persist the changes made to a detached object, it is required to reattach the application to a valid Hibernate session.
- To associate the detached object with the new hibernate session, use any of these methods - load(), merge(), refresh(), update() or save() on a new session with the reference of the detached object.

//Lifecycle example

```
public class HibernateLifeCycle {

        public static void main (String [] args) {

      SessionFactory sessionFactory =
HibernateConfiguration.getSessionFactory();
        Session session = sessionFactory.openSession();
      session.beginTransaction();

        // transient state
        Laptop l = new Laptop ();
      l.setBrand("Sony");
      l.setPrice(434567d);

        // persistent state
      session.save(l);
      l.setPrice(45000d);
      l.setPrice(95000d);// last one will update

      session.getTransaction().commit();
        // detach state
      session.detach(l);
      l.setPrice(30000d);
      session.close();
      l.setPrice(50000d);

        }

}
```

// Moving from Detached state to Transient State

```
public class DetachedToTransientState {

        public static void main (String[] args) {

      SessionFactory sessionFactory =
HibernateConfiguration.getSessionFactory();
        Session session = sessionFactory.openSession();
```

```
    session.beginTransaction();

      //persistent state
      Laptop laptop = session.get(Laptop.class, 158);
    System.out.println(laptop);
    session.getTransaction().commit();
      //detached state
    session.close();


    laptop.setBrand("Apple");
      session = sessionFactory.openSession();
    session.beginTransaction();
      //persistent state
    session.update(laptop);

    session.getTransaction().commit();
      //detached
    session.close();

      }

}
```

9)Hibernate Embedded Objects and List

@Embeddable and @Embedded together.

You mark your class as @Embeddable, which indicates that this class will not exist in the DB as a separate table.
Now when you use @Embedded on the field itself.

```
@Entity
@Table (name = "user")
public class User {
        @Id
        @GeneratedValue
        private Integer id;
        private String name;
        private Long mobile;
        @Temporal(TemporalType.DATE)
        private Date dob;
        @Embedded
private Address address;
//getters setters
}

@Embeddable
public class Address {

        @Column(name="state_name")
        private String state;
        @Column(name="city_name")
        private String city;
        @Column(name="street_name")
        private String street;
        @Column(name="pin_code")
        private Integer pin;

//getters & Setters

}

public class HibernateValueTypeEmbeddedObjects {

        @SuppressWarnings("deprecation")
        public static void main(String[] args) {

      SessionFactory sessionFactory =
HibernateConfiguration.getSessionFactory();
        Session session = sessionFactory.openSession();
      session.beginTransaction();
        User u = new User();
      u.setName("Ratna");
      u.setDob(new Date(1992, 9, 1));
      u.setMobile(8499899121l);

        Address a = new Address();
      a.setCity("Hyderabad");
      a.setState("Telangana");
      a.setPin(500081);
      a.setStreet("Ayyappa Society Sai Nagar");
      u.setAddress(a);
      session.save(u);
session.getTransaction().commit();
  } }
```

Attribute Overrides

Suppose If you want to save two instances of the same object with home address and permanent address then we use @AttributeOverrides and change the fields of one instance.

```
@Entity
@Table(name = "person")
public class Person {

	@Id
	@GeneratedValue
	private Integer id;
	private String name;
	private Long mobile;
	@Temporal(TemporalType.DATE)
	private Date dob;
	@Embedded
	@AttributeOverrides({ @AttributeOverride(name = "state", column =
@Column(name = "home_state_name")),
	@AttributeOverride(name = "street", column = @Column(name =
"home_street_name")),
	@AttributeOverride(name = "city", column = @Column(name =
"home_city_name")),
	@AttributeOverride(name = "pin", column = @Column(name =
"home_pin_code")) })
	private Address homeAddress;
	@Embedded // not mandatory to put annotation if class is marked as
//embeddble
	private Address workAddress;
```

```
//getters setters
}

@Embeddable
public class Address {

        @Column(name="state_name")
        private String state;
        @Column(name="city_name")
        private String city;
        @Column(name="street_name")
        private String street;
        @Column(name="pin_code")
        private Integer pin;}

public class HibernateAttributeOverridesEmbeddedObjects {

        @SuppressWarnings("deprecation")
        public static void main(String[] args) {

      SessionFactory sessionFactory =
HibernateConfiguration.getSessionFactory();
        Session session = sessionFactory.openSession();
      session.beginTransaction();
        Person u = new Person();
      u.setName("Srinivas");
      u.setDob(new Date(1992, 9, 1));
      u.setMobile(8499899121l);

        Address a = new Address();
      a.setCity("Hyderabad");
      a.setState("Telangana");
```

```
        a.setPin(500081);
        a.setStreet("Ayyappa Society Sai Nagar");
        u.setHomeAddress(a);

           Address a1 = new Address();
           a1.setCity("Hyderabad");
           a1.setState("Telangana");
           a1.setPin(500081);
           a1.setStreet("DLF Building");
        u.setWorkAddress(a1);

        session.save(u);

        session.getTransaction().commit();

           }

}
```

ElementCollection

In Hibernate Annotation, @ElementCollection is the feature which gets the column values from another table without mapping two tables.
We have taken two entity client and address.
In client entity, we will fetch address without mapping client and address entity.

@CollectionTable will join the two tables for the given primary and foreign key.

```
@Entity
@Table(name = "client")
public class Client {
	@Id
	@GeneratedValue
	private Integer id;
	private String name;
	private Long mobile;
	@Temporal(TemporalType.DATE)
	private Date dob;
	@ElementCollection
	@JoinTable(name = "client_address", joinColumns = @JoinColumn(name
= "client_id"))
	@GenericGenerator(name = "increment", strategy = "increment")
	@CollectionId(columns = @Column(name = "address_id"), generator =
"increment", type = @Type(type = "long"))
	private Collection<Address> address = new ArrayList<>();
//getters & setters
}

@Embeddable
public class Address {

	@Column(name="state_name")
	private String state;
	@Column(name="city_name")
	private String city;
	@Column(name="street_name")
	private String street;
	@Column(name="pin_code")
	private Integer pin;
```

```
//getters setters
}

public class HibernateSavingCollections {

	public static void main(String[] args) {

       SessionFactory sessionFactory =
HibernateConfiguration.getSessionFactory();
	Session session = sessionFactory.openSession();
       session.beginTransaction();
	Client u = new Client();
       u.setName("Ratna");
       u.setDob(new Date());
       u.setMobile(8499899121l);

	Address a = new Address();
       a.setCity("Hyderabad");
       a.setState("Telangana");
       a.setPin(500081);
       a.setStreet("Ayyappa Society Sai Nagar");

	Address a1 = new Address();
	a1.setCity("Vijyawada");
	a1.setState("Andhra");
	a1.setPin(521157);
	a1.setStreet("Annapurna Nilayam");


       u.getAddress().addAll(Arrays.asList(a, a1));

       session.save(u);
	session.getTransaction().commit();

	}

}
```

Embedded – Id

It is like a composite primary key

```

@Entity
@Table(name = "account")
public class Account {
	@EmbeddedId
	private AccountSecret loginId;
	private String branch;
	private String ifscCode;
	private double accountNumber;
//getters setters
}

@Embeddable
public class AccountSecret implements Serializable {
	/**
	*
	*/
	private static final long serialVersionUID = 1L;
```

```
        @Temporal(TemporalType.DATE)
        private Date accountStartDate;
        private Long mobileNumber;
//getters setters
}

public class HibernateEmbeddedId {

        public static void main(String[] args) {

      SessionFactory sessionFactory =
HibernateConfiguration.getSessionFactory();
        Session session = sessionFactory.openSession();
      session.beginTransaction();

        Account a = new Account();
      a.setAccountNumber(123456789);
      a.setBranch("MLA Colony");
      a.setIfscCode("IFSC12345");
      AccountSecret accountSecret = new AccountSecret();
      accountSecret.setAccountStartDate(new Date());
      accountSecret.setMobileNumber(8499899121l);
      a.setLoginId(accountSecret);
      session.save(a);
      session.getTransaction().commit();

        }

}
```

10)Hibernate HQL

Hibernate Query Language (HQL) is the same as SQL (Structured Query Language) but it doesn't depend on the table of the database.

Instead of table names, we use class names in HQL. So it is a database independent query language.

```
@SuppressWarnings("deprecation")
@Entity
@Table(name = "laptop")
@org.hibernate.annotations.Entity(selectBeforeUpdate = true)
public class Laptop {
	@Id
	@GeneratedValue
	private Integer id;

	@Column
	private String brand;

	@Column
	private Double price;

//setters getters
}

@SuppressWarnings("deprecation")
public class LaptopDAO {

	@SuppressWarnings("unchecked")
	public static List<Laptop> getAllLaptops(Session session, String query) {
		Query<Laptop> reqList = session.createQuery(query);
		return reqList.list();

	}

	@SuppressWarnings("unchecked")
	public static Laptop getRequiredLaptop(Session session, String query) {
		Query<Laptop> reqLaptop = session.createQuery(query);
		return reqLaptop.uniqueResult();
	}

	@SuppressWarnings("unchecked")
	public static Object[] getRequiredData(Session session, String query) {
		Query<Object[]> reqData = session.createQuery(query);
		return reqData.uniqueResult();
	}

	@SuppressWarnings("unchecked")
	public static List<Object[]> getRequiredDataList(Session session, String
query) {
		Query<Object[]> reqList = session.createQuery(query);
		return reqList.list();

	}

	@SuppressWarnings("unchecked")
	public static Double getTotals(Session session, String query) {
		Query<Double> reqData = session.createQuery(query);
		return reqData.uniqueResult();
	}

	@SuppressWarnings("unchecked")
	public static Double getTotalsUsingPositionalParameters(Session session,
String query, double b) {
		Query<Double> reqData = session.createQuery(query);
		reqData.setParameter("b", b);
		return reqData.uniqueResult();
	}
	@SuppressWarnings("unchecked")
	public static List<Laptop> getAllLaptopsPagination(Session session, String
query) {
		Query<Laptop> reqList = session.createQuery(query);
		reqList.setFirstResult(1);
		reqList.setMaxResults(5);
		return reqList.list();

	}

	@SuppressWarnings("unchecked")
	public static int updateLaptop(Session session, String query, double price,
String brand) {
		Query<Double> reqData = session.createQuery(query);
		reqData.setParameter("price", price);
		reqData.setParameter("brand", brand);
		return reqData.executeUpdate();
	}

	@SuppressWarnings("unchecked")
	public static int deleteLaptop(Session session, String query, String brand) {
		Query<Double> reqData = session.createQuery(query);
		reqData.setParameter("brand", brand);
		return reqData.executeUpdate();
	}

	@SuppressWarnings({ "unchecked" })
	public static List<Laptop> getListAsNativeSQL(Session session, String
query) {
		SQLQuery<Laptop> reqList = session.createSQLQuery(query);
		reqList.addEntity(Laptop.class);
		return reqList.list();

	}

	@SuppressWarnings({  "rawtypes" })
	public static List getRequiredFieldListAsNativeSQL(Session session, String
query) {
		SQLQuery reqList = session.createSQLQuery(query);
		reqList.setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);
		return reqList.list();

	}

}

public class HibernateHQL {

	public static void main(String[] args) {
		double b = 20000;
		SessionFactory sessionFactory =
HibernateConfiguration.getSessionFactory();
		Session session = sessionFactory.openSession();
		session.beginTransaction();
		// getAllLaptops
		List<Laptop> allLaptops = LaptopDAO.getAllLaptops(session, "from
Laptop");
		allLaptops.forEach(System.out::println);

		// getAllLaptops whose price greater than 30000
		List<Laptop> filteredList = LaptopDAO.getAllLaptops(session, "from
Laptop where price>30000");
		filteredList.forEach(System.out::println);

		// getRequiredLaptop whose brand is Laptop : 14
		Laptop requiredLaptop = LaptopDAO.getRequiredLaptop(session,
"from Laptop where brand='Laptop : 14'");
		System.out.println(requiredLaptop);

		// getRequiredData whose brand is Laptop : 14
		Object[] requiredData = LaptopDAO.getRequiredData(session,
				"select brand,price from Laptop where brand='Laptop :
14'");
		Arrays.asList(requiredData).forEach(System.out::println);

		// getAllDataList
		List<Object[]> requiredDataList =
LaptopDAO.getRequiredDataList(session, "select brand,price from Laptop ");
		requiredDataList.forEach(obj -> System.out.println(obj[0] + " :" +
obj[1]));

		// getRequiredDataList
		List<Object[]> requiredList =
LaptopDAO.getRequiredDataList(session,
				"select brand,price from Laptop l where l.price>40000
");
		requiredList.forEach(obj -> System.out.println(obj[0] + " :" +
obj[1]));

		// getTotals
		Double getTotals = LaptopDAO.getTotals(session, "select
sum(price) from Laptop where price >20000 ");
		System.out.println(getTotals);

		// getTotals with positional params
		Double getTotalsPos =
LaptopDAO.getTotalsUsingPositionalParameters(session,
				"select sum(price) from Laptop where price >:b ", b);
		System.out.println(getTotalsPos);

		// allLaptopsPagination
		List<Laptop> allLaptopsPagination =
LaptopDAO.getAllLaptopsPagination(session, "from Laptop");
		allLaptopsPagination.forEach(System.out::println);

		// updateLaptop
		int updateLaptop = LaptopDAO.updateLaptop(session, "update
Laptop set price=:price where brand=:brand", 60000d,
				"Laptop : 1");
		System.out.println(updateLaptop);

		// deleteLaptop
		int deleteLaptop = LaptopDAO.deleteLaptop(session, "delete from
Laptop where brand=:brand", "Laptop : 1");
```

```
                System.out.println(deleteLaptop);

                session.getTransaction().commit();
        }

}
```

11)Hibernate HCQL

The Hibernate Criteria Query Language (HCQL) is used to fetch the records based on the specific criteria.

The Criteria interface provides methods to apply criteria such as retrieving all the records of a table whose salary is greater than 50000 etc.

The HCQL provides methods to add criteria, so it is **easy** for the java programmer to add criteria.

The java programmer is able to add many criteria on a query.

```
@Entity
@Table(name = "student")
public class Student {
	@Id
	@GeneratedValue
	private int id;

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

	@Column(name = "rollnumber")
	private Integer rollNumber;

	@Column(name = "gender")
	private Boolean gender;

//getters & setters
}

public class StudentDAO {

	@SuppressWarnings({ "unchecked", "deprecation" })
	public static List<Student> getAllStudents(Session session) {
		Criteria createCriteria = session.createCriteria(Student.class);
		return createCriteria.list();

	}

	@SuppressWarnings({ "unchecked", "deprecation" })
	public static List<Student> getAllStudentsPagination(Session session, int
pos1, int pos2) {
		Criteria createCriteria = session.createCriteria(Student.class);
		createCriteria.setFirstResult(pos1);
		createCriteria.setMaxResults(pos2);
		return createCriteria.list();
	}

	// get the records whose rollNumber is greater than 500
	@SuppressWarnings({ "unchecked", "deprecation" })
	public static List<Student> getFilteredStudents(Session session) {
		Criteria createCriteria = session.createCriteria(Student.class);
		createCriteria.add(Restrictions.ge("rollNumber", 500))
				.add(Restrictions.in("id", Arrays.asList(98, 99, 100,
101))).addOrder(Order.desc("rollNumber"));
		return createCriteria.list();

	}

	// HCQL with Projection
	@SuppressWarnings({ "unchecked", "deprecation" })
	public static List<Integer> getFilteredStudentsByProjections(Session
session) {
		Criteria c = session.createCriteria(Student.class);
		c.setProjection(Projections.max("rollNumber"));
		return c.list();
	}

	// get the records whose rollNumber between 100 and 500
	@SuppressWarnings({ "unchecked", "deprecation" })
	public static List<Student> getFilteredStudentsBetween(Session session) {
		Criteria createCriteria = session.createCriteria(Student.class);
		createCriteria.add(Restrictions.between("rollNumber", 100,
500)).addOrder(Order.asc("rollNumber"));
		return createCriteria.list();

	}

	// restrictions or
	@SuppressWarnings({ "unchecked", "deprecation" })
	public static List<Student> getFilteredStudentsOrRestriction(Session
session) {
		Criteria createCriteria = session.createCriteria(Student.class);
		createCriteria.add(
				Restrictions.or(Restrictions.between("rollNumber",
100, 300), Restrictions.between("id", 90, 100)));
		return createCriteria.list();

	}
	// restrictions or
	@SuppressWarnings({ "unchecked", "deprecation" })
	public static List<Student> getFilteredStudentsAndRestriction(Session
session) {
		Criteria createCriteria = session.createCriteria(Student.class);
		createCriteria.add(
				Restrictions.and(Restrictions.between("rollNumber",
100, 300), Restrictions.between("id", 90, 100)));
		return createCriteria.list();

	}

	// queryby
	@SuppressWarnings({ "unchecked", "deprecation" })
	public static List<Student> getStudentsQueryBy(Session session) {
		Student s = new Student();
		s.setName("name :%");
		s.setRollNumber(565);
		Criteria createCriteria = session.createCriteria(Student.class);
		Example example = Example.create(s).enableLike();
		createCriteria.add(example);
		return createCriteria.list();

	}

}

public class HibernateCriteriaQueryLanguage {

	public static void main(String[] args) {

		SessionFactory sessionFactory =
HibernateConfiguration.getSessionFactory();
		Session session = sessionFactory.openSession();
		session.beginTransaction();

		// getAllStudents
		List<Student> allStudents = StudentDAO.getAllStudents(session);
		allStudents.forEach(System.out::println);

		// getAllStudentsPagination
		List<Student> allStudentsPagination =
StudentDAO.getAllStudentsPagination(session, 1, 2);
		allStudentsPagination.forEach(System.out::println);

		// getAllStudents rollNumber greater than 500
		List<Student> rollNumber =
StudentDAO.getFilteredStudents(session);
		rollNumber.forEach(System.out::println);

		// getAllStudents with projections
		List<Integer> projections =
StudentDAO.getFilteredStudentsByProjections(session);
		projections.forEach(System.out::println);

		// getAllStudents rollNumber between 100 and 500
		List<Student> rollNumberBetween =
StudentDAO.getFilteredStudentsBetween(session);
		rollNumberBetween.forEach(System.out::println);

		// or criteria
		List<Student> getFilteredStudentsOrRestriction =
StudentDAO.getFilteredStudentsOrRestriction(session);
		getFilteredStudentsOrRestriction.forEach(System.out::println);

		// and criteria
		List<Student> getFilteredStudentsAndRestriction =
StudentDAO.getFilteredStudentsAndRestriction(session);
		getFilteredStudentsAndRestriction.forEach(System.out::println);

		// queryBy
		List<Student> getStudentsQueryBy =
StudentDAO.getStudentsQueryBy(session);
		getStudentsQueryBy.forEach(System.out::println);
```

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

        }

}
```

12)Hibernate Named Queries

The hibernate named query is a way to use any query by some meaningful name.

It is like using alias names.

The Hibernate framework provides the concept of named queries so that application programmers need not to scatter queries to all the java code.

```
@NamedQueries({ @NamedQuery(name = "findEmployeeByName", query =
"from Employee e where e.name=:name"),
                @NamedQuery(name = "findEmployeeByAge", query = "from
Employee e where e.age=:age") })
@NamedNativeQueries({ @NamedNativeQuery(name = "findEmployeeByID",
query = "select * from employee where empId=:empId"),
                @NamedNativeQuery(name = "findEmployeeBySalary", query =
"select * from employee where salary=:salary", resultClass = Employee.class) })
@Entity
@Table(name = "employee")
public class Employee {

        @Id
        @GeneratedValue
        private Integer empId;
        @Column
        private String name;

        @Column
        private Integer salary;

        @Column
        private Integer age;

//getters setters
}

public class EmployeeDAO {

        public static List<Employee> getEmployeeListByName(Session session,
String query, String name) {
                @SuppressWarnings("unchecked")
                Query<Employee> namedQuery = session.getNamedQuery(query);
                namedQuery.setParameter("name", name);
                return namedQuery.list();
        }

        public static List<Employee> getEmployeeListByAge(Session session,
String query, int age) {
                @SuppressWarnings("unchecked")
                Query<Employee> namedQuery = session.getNamedQuery(query);
                namedQuery.setParameter("age", age);
                return namedQuery.list();
        }

        @SuppressWarnings("unchecked")
        public static Employee
getEmployeeListByNativeNamedQueryEmpId(Session session, String query, int
empId) {
                NativeQuery<Employee> namedNativeQuery =
session.getNamedNativeQuery(query);
                namedNativeQuery.addEntity(Employee.class);
                namedNativeQuery.setParameter("empId", empId);
```

```
		return namedNativeQuery.uniqueResult();
	}

	public static List<Employee> getEmployeeListByNativeQuerySal(Session
session, String query, int salary) {
		@SuppressWarnings("unchecked")
		Query<Employee> namedQuery = session.getNamedQuery(query);
		namedQuery.setParameter("salary", salary);
		return namedQuery.list();
	}

}


public class HibernateNamedQueries {

	public static void main(String[] args) {

		SessionFactory sessionFactory =
HibernateConfiguration.getSessionFactory();
		Session session = sessionFactory.openSession();
		session.beginTransaction();

		// get an employee by name
		List<Employee> employeeListByName =
EmployeeDAO.getEmployeeListByName(session, "findEmployeeByName", "Emp
:1");
		employeeListByName.forEach(System.out::println);

		// get an employee by age
		List<Employee> employeeListByAge =
EmployeeDAO.getEmployeeListByAge(session, "findEmployeeByAge", 20);
		employeeListByAge.forEach(System.out::println);

		// get an employee by id
		Employee employeeListByNativeNamedQueryEmpId =
EmployeeDAO.getEmployeeListByNativeNamedQueryEmpId(session,
				"findEmployeeByID", 102);
		System.out.println(employeeListByNativeNamedQueryEmpId);

		// get an employee by sal
		List<Employee> employeeListByNativeNamedQuerySalary =
EmployeeDAO.getEmployeeListByNativeQuerySal(session,
				"findEmployeeBySalary", 8379);
		System.out.println(employeeListByNativeNamedQuerySalary);

		session.getTransaction().commit();
	}

}
```

13)Hibernate SQL

We can use native SQL to express database queries if you want to utilize database-specific features such as query hints or the CONNECT keyword in Oracle.

```
@SuppressWarnings("deprecation")
@Entity
@Table(name = "laptop")
@org.hibernate.annotations.Entity(selectBeforeUpdate = true)
public class Laptop {
	@Id
	@GeneratedValue
	private Integer id;

	@Column
	private String brand;

	@Column
	private Double price;
```

```
}

@SuppressWarnings("deprecation")
public class LaptopDAO {

        @SuppressWarnings("unchecked")
        public static List<Laptop> getAllLaptops(Session session, String query) {
                Query<Laptop> reqList = session.createQuery(query);
                return reqList.list();

        }

        @SuppressWarnings("unchecked")
        public static Laptop getRequiredLaptop(Session session, String query) {
                Query<Laptop> reqLaptop = session.createQuery(query);
                return reqLaptop.uniqueResult();
        }

        @SuppressWarnings("unchecked")
        public static Object[] getRequiredData(Session session, String query) {
                Query<Object[]> reqData = session.createQuery(query);
                return reqData.uniqueResult();
        }

        @SuppressWarnings("unchecked")
        public static List<Object[]> getRequiredDataList(Session session, String
query) {
                Query<Object[]> reqList = session.createQuery(query);
                return reqList.list();

        }

        @SuppressWarnings("unchecked")
        public static Double getTotals(Session session, String query) {
                Query<Double> reqData = session.createQuery(query);
                return reqData.uniqueResult();
        }
```

```
	@SuppressWarnings("unchecked")
	public static Double getTotalsUsingPositionalParameters(Session session,
String query, double b) {
		Query<Double> reqData = session.createQuery(query);
		reqData.setParameter("b", b);
		return reqData.uniqueResult();
	}

	@SuppressWarnings("unchecked")
	public static List<Laptop> getAllLaptopsPagination(Session session, String
query) {
		Query<Laptop> reqList = session.createQuery(query);
		reqList.setFirstResult(1);
		reqList.setMaxResults(5);
		return reqList.list();

	}

	@SuppressWarnings("unchecked")
	public static int updateLaptop(Session session, String query, double price,
String brand) {
		Query<Double> reqData = session.createQuery(query);
		reqData.setParameter("price", price);
		reqData.setParameter("brand", brand);
		return reqData.executeUpdate();
	}

	@SuppressWarnings("unchecked")
	public static int deleteLaptop(Session session, String query, String brand) {
		Query<Double> reqData = session.createQuery(query);
		reqData.setParameter("brand", brand);
		return reqData.executeUpdate();
	}

	@SuppressWarnings({ "unchecked" })
	public static List<Laptop> getListAsNativeSQL(Session session, String
query) {
		SQLQuery<Laptop> reqList = session.createSQLQuery(query);
		reqList.addEntity(Laptop.class);
		return reqList.list();

	}

	@SuppressWarnings({  "rawtypes" })
	public static List getRequiredFieldListAsNativeSQL(Session session, String
query) {
		SQLQuery reqList = session.createSQLQuery(query);
		reqList.setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);
		return reqList.list();

	}

}

public class HibernateSQL {

	@SuppressWarnings("unchecked")
	public static void main(String[] args) {

		SessionFactory sessionFactory =
HibernateConfiguration.getSessionFactory();
		Session session = sessionFactory.openSession();
		session.beginTransaction();

		// getAllLaptops whose price greater than 30000
		List<Laptop> filteredList = LaptopDAO.getListAsNativeSQL(session,
"select * from laptop where price>30000");
		filteredList.forEach(System.out::println);

		// getAllLaptops whose price greater than 30000
		@SuppressWarnings("rawtypes")
		List filteredLists =
LaptopDAO.getRequiredFieldListAsNativeSQL(session,
				"select brand,price from laptop where price>30000");
		filteredLists.forEach(obj -> {
			@SuppressWarnings("rawtypes")
```

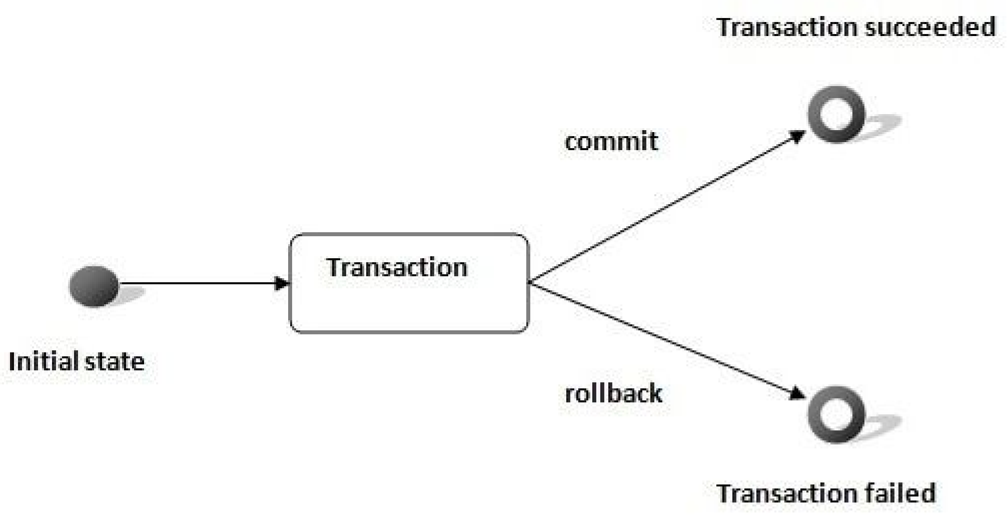
```
                    Map map = (Map) obj;
                    System.out.println(map.get("brand") + " " +
map.get("price"));
            });

            session.getTransaction().commit();

      }

}
```

14)Hibernate Transaction Management

A **transaction** simply represents a unit of work.

In such a case, if one step fails, the whole transaction fails (which is termed as atomicity).

A transaction can be described by ACID properties (Atomicity, Consistency, Isolation and Durability).



In the hibernate framework, we have a Transaction interface that defines the unit of work. It maintains abstraction from the transaction implementation (JTA,JDBC).

A transaction is associated with Session and instantiated by calling **session.beginTransaction()**.

The methods of Transaction interface are as follows:

1. **void begin()** starts a new transaction.
2. **void commit()** ends the unit of work unless we are in FlushMode.NEVER.
3. **void rollback()** forces this transaction to rollback.
4. **void setTimeout(int seconds)** it sets a transaction timeout for any transaction started by a subsequent call to begin on this instance.
5. **boolean isAlive()** checks if the transaction is still alive.
6. **void registerSynchronization(Synchronization s)** registers a user synchronization callback for this transaction.

7. **boolean wasCommited()** checks if the transaction is committed successfully.
8. **boolean wasRolledBack()** checks if the transaction is rolledback successfully.

```
@Entity
@Table(name = "student")
public class Student {

	@Id
	@GeneratedValue
	private int id;

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

	@Column(name = "rollnumber")
	private Integer rollNumber;

	@Column(name = "gender")
	private Boolean gender;
//getters setters
}

public class HibernateTM {

	public static void main(String[] args) {
		Transaction tx = null;
		Session session = null;
		try {
			SessionFactory sessionFactory =
HibernateConfiguration.getSessionFactory();
```

```
                        session = sessionFactory.openSession();
                        tx = session.beginTransaction();
                        Student s = new Student();
                        s.setName("Rocky");
                        s.setGender(true);
                        s.setRollNumber(818899);
                        session.save(s);

                        if ("Rocky".equalsIgnoreCase(s.getName())) {
                                throw new RuntimeException("Checking RTE");
                        }
                        tx.commit();
                } catch (RuntimeException e) {
                        try {
                                tx.rollback();
                        } catch (RuntimeException rbe) {
                                System.out.println(rbe.getMessage());
                        }
                } finally {
                        if (session != null) {
                                session.close();
                        }
                }

        }

}
```

15)Hibernate Inheritance Mapping

We can map the inheritance hierarchy classes with the table of the database. There are three inheritance mapping strategies defined in the hibernate:

1. Table Per Hierarchy
2. Table Per Concrete class
3. Table Per Subclass

Table Per Hierarchy

In table per hierarchy mapping, a single table is required to map the whole hierarchy, an extra column (known as discriminator column) is added to identify the class. But nullable values are stored in the table .
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.

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

public class Teacher {

        @Id
        @GeneratedValue(strategy=GenerationType.AUTO)

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

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

}

@Entity
@DiscriminatorValue("contractteacher")
public class ContractTeacher extends Teacher {

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

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

@Entity
@DiscriminatorValue("permanentteacher")
public class PermanentTeacher extends Teacher {

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

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

}

public class TablePerHierarchyInheritence {
public static void main(String[] args) {
SessionFactory sessionFactory = HibernateConfiguration.getSessionFactory();
                Session session = sessionFactory.openSession();
                session.beginTransaction();
                Teacher t = new Teacher();
                t.setName("principal");
                ContractTeacher c = new ContractTeacher();
                c.setContract_duration("60 days");
                c.setPay_per_hour(1200);
                c.setName("contract teacher");
                PermanentTeacher p = new PermanentTeacher();
                p.setName("permanent teacher");
                p.setSalary(50000);
                p.setBonus(20);
                session.save(t);
                session.save(c);
                session.save(p);
session.getTransaction().commit();
}}
```

Table Per Class

In the 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.

```
@Entity
@Table(name = "animal")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Animal {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        private Integer id;
        @Column
        private String name;
```

```
//setters getters
}

@Entity
@Table(name="wild_animal")
public class WildAnimal extends Animal {

        private int weight;
        private String color;
       //setters getters
}

public class TablePerClassInheritence {

        public static void main(String[] args) {
                SessionFactory sessionFactory
=HibernateConfiguration.getSessionFactory();
                Session session = sessionFactory.openSession();
                session.beginTransaction();

                Animal a = new Animal();
                a.setName("animal");

                WildAnimal w = new WildAnimal();
                w.setName("lion");
                w.setColor("yellow");
                w.setWeight(350);

                PetAnimal p = new PetAnimal();
                p.setBreed("Labrador");
                p.setCost(15000);
                p.setName("Dog");

                session.save(a);
                session.save(w);
                session.save(p);

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

```
	}

}
```

Table Per Subclass

In the case of table per subclass strategy, tables are created as per persistent classes but they are treated using primary and foreign keys.

So there will not be any duplicate columns in the relation.

```
@Entity
@Table(name="vehicle")
@Inheritance(strategy = InheritanceType.JOINED)
public class Vehicle {

	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)

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

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

@Entity
@Table(name="two_wheeler")
public class TwoWheeler extends Vehicle{

	private String controlType;
	private float price;
}
```

```
@Entity
@Table(name="four_wheeler")
public class FourWheeler extends Vehicle{
	private String controlType;
	private float price;
}

public class JoinedStrategyInheritence {

	public static void main(String[] args) {
		SessionFactory sessionFactory =
HibernateConfiguration.getSessionFactory();
		Session session = sessionFactory.openSession();
		session.beginTransaction();

		Vehicle v = new Vehicle();
		v.setName("Vehicle");

		TwoWheeler t = new TwoWheeler();
		t.setName("bike");
		t.setControlType("handle");
		t.setPrice(80000);

		FourWheeler f = new FourWheeler();
		f.setName("car");
		f.setPrice(4500000);
		f.setControlType("steering");

		session.save(f);
		session.save(t);
		session.save(v);

		session.getTransaction().commit();

	}

}
```

16)Hibernate Relational Mapping

17)Hibernate Loading

18)Hibernate Caching