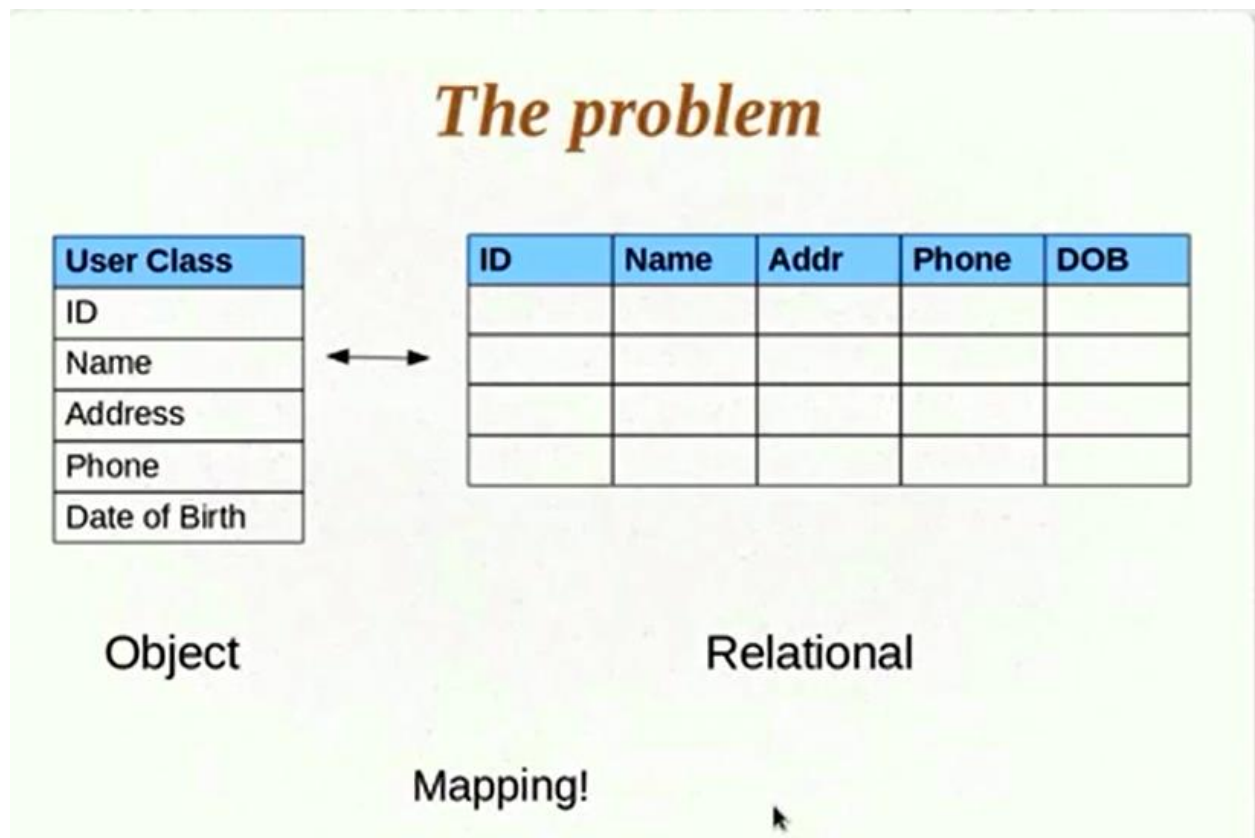


# HIBERNATE

## (1) Introduction to Hibernate

- An ORM tool
- Used in the data layer of applications
- Implements JPA (JPA is a reference/standard that an ORM framework follows. If you want to change your JPA implemented ORM tool at a later point of time, you can achieve it with minimal code change)

The problems:



- (1) Mapping member variables(in Java) to columns(in DB)  
Suppose you need to create an extra field in Java, then you have to go to the recordset (DAO) and pull the new DB column.
- (2) Mapping relationships  
Let's say a user object in java has a reference to an address object. In the DB we will have to create a foreign key (address) to a primary key (user). So this relationship we have to take care of.
- (3) Handling data types

Suppose we have a Boolean data type in my user object. How we will convert this data type into a DB data type as most of the DB do not have a Boolean data type. This we will have to manage.

(4) Managing changes to the object state

(2) Setting up Hibernate

Download the binaries (jar) and add them to classpath.

Supply Hibernate the corresponding JDBC driver for your chosen database. This will help hibernate to connect to the database.

(3) Hibernate Tutorial 03 Part 1- Writing a Hibernate Application

## ***Saving Without Hibernate***

- JDBC Database configuration
- The Model object
- Service method to create the model object
- Database design
- DAO method to save the object using SQL queries

## *The Hibernate way*

- JDBC Database configuration – Hibernate configuration
- The Model object – Annotations
- Service method to create the model object – Use the Hibernate API
- Database design – Not needed!
- DAO method to save the object using SQL queries – Not needed!

(4) Hibernate Tutorial 03 Part 3- Saving Objects using Hibernate APIs

## *Using the Hibernate API*

- Create a session factory
- Create a session from the session factory
- Use the session to save model objects

---

(5) Hibernate Tutorial 04 - hbm2ddl Configuration and Name Annotations

## (5) Hibernate Tutorial 05 - More Annotations

Most of the times Hibernate maps a java data type to a DB data type automatically. What if we want to take more control of data types?

@Basic → Treat it as a field that should be persisted and apply hibernate defaults.

## (6) Hibernate Tutorial 06 - Retrieving Objects using session.get

a) hibernate.cfg.xml

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

<hibernate-configuration>

    <session-factory>

        <!-- Database connection settings -->
        <property
name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
        <property
name="hibernate.connection.url">jdbc:oracle:thin:@localhost:1521:orcl</property>
        <property name="hibernate.connection.username">scott</property>
        <property name="hibernate.connection.password">tiger</property>

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

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

        <!-- SQL dialect -->
        <property name="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect</property>

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

        <!-- Drop and re-create the database schema on startup -->
        <!-- <property name="hbm2ddl.auto">create</property> -->
        <!-- 'update' will ensure that the schema is not dropped and recreated if there is
no change in schema -->
        <property name="hbm2ddl.auto">create</property>
```

```

        <!-- Names the annotated entity class (these classes hibernate needs to persist in DB) -->
        <mapping class="com.kunj.personal.dto.UserDetails" />
    </session-factory>
</hibernate-configuration>

```

b) HibernateTest.java

```
package com.kunj.hibernate;
```

```
import java.util.Date;
```

```
import org.hibernate.Session;
```

```
import org.hibernate.SessionFactory;
```

```
import org.hibernate.cfg.Configuration;
```

```
import com.kunj.personal.dto.UserDetails;
```

```
public class HibernateTest {
```

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

```
        UserDetails userDetails = new UserDetails();
```

```
        userDetails.setUserID(1);
```

```
        userDetails.setUserName("Kunj");
```

```
        userDetails.setJoinedDate(new Date());
```

```
        userDetails.setAddress("First user's address");
```

```
        userDetails.setDescription("Description");
```

```
        /*
```

```
        * A SessionFactory is created once per application.It is an expansive object.
```

```
        * It takes lot of resource to be created.
```

```
        */
```

```
        SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
```

```
        Session session = sessionFactory.openSession();
```

```
        session.beginTransaction();
```

```
        session.save(userDetails);
```

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

```
        session.close(); // Ideally to be written in a finally block after transaction rollback
```

```
        session = sessionFactory.openSession(); // A SessionFactory is created only once
```

```
        session.beginTransaction();
```

```
        // Fetch data from table 'UserDetails' where value of primary key (userID) is 1
```

```

        userDetails = (UserDetails) session.get(UserDetails.class, 1); // This will trigger
the select query
        System.out.println("Username is : " + userDetails.getUserName());
    }
}

```

c) UserDetails.java

```
package com.kunj.personal.dto;
```

```
import java.util.Date;
```

```
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Lob;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
```

```

/* (1) @Entity(name="USER_DETAILS") VS (2) @Table(name="USER_DETAILS")
 * (1) Changes table name as well as entity name (2) changes just the table name, not the
entity name
 */

```

```

@Entity
@Table(name = "USER_DETAILS")
public class UserDetails {

```

```

    @Id
    private int userID;

```

```

/*
 * @Basic()--> Treat 'userName' as a field that should be persisted and apply
 * the hibernate defaults.
 *
 * @Basic can be used with 'fetch', 'optional' parameter. But without these 2
 * annotations, @Basic is as good as using no annotation.
 */
/*
 * What if we want to tell hibernate not to save a particular property. Use
 * 'Transient'. If you have any transient or static property in the class.
 * Hibernate will not automatically save it. Suppose you have 'userName' as
 * static. So all the objects will be sharing the same value for the userName.
 * In this case, this does not make sense to create a separate column for

```

```

* 'userName'.
*
* Another way to do the above: If you do not want to change the userName filed,
* but want hibernate to ignore the field. Use '@Transient'.
*/
// @Basic
// @Transient
private String userName;
// By default Date will be saved as Timestamp in the DB. But you can change it
// as below.
@Temporal(TemporalType.DATE)
private Date joinedDate;
private String address;
/*
* By default, hibernate create a varchar2 of size 255. What if the description
* contains more than 255 character. Use '@Lob'. @Lob means it is a large object
* two types of Lob: (1) Clob -- Character large object (2) Blob -- Byte stream
* large object
*/
@Lob
private String description;

public Date getJoinedDate() {
    return joinedDate;
}

public void setJoinedDate(Date joinedDate) {
    this.joinedDate = joinedDate;
}

public String getAddress() {
    return address;
}

public void setAddress(String address) {
    this.address = address;
}

public String getDescription() {
    return description;
}

public void setDescription(String description) {
    this.description = description;
}

```

```

    }

    public int getUserID() {
        return userID;
    }

    public void setUserID(int userID) {
        this.userID = userID;
    }

    public String getUsername() {
        return userName;
    }

    public void setUsername(String userName) {
        this.userName = userName;
    }
}

```

---

## (7) Hibernate Tutorial 07 - Primary Keys

Different ways in which Hibernate provides support for primary keys.

`@Id` → Creates a primary key column in table

Natural key vs Surrogate key :

Natural key → When you already know that a column value has to be unique. It has a business use. ex: emailID

Surrogate key → You do not have a column that can be marked as unique or you anticipate that that can change in future. In this case, you add a new column. The purpose of this new column is to act entirely as a primary key column. It does not carry any business significance. Ex: SerialNO

If a column is a primary key then we need to provide a value for it, otherwise we can ask hibernate to provide a value for surrogate key.

What if we want to provide surrogate key? → We have to see the last created surrogate key and provide our own key based on that. `@GeneratedValue` annotation will be used.

```

@Id @GeneratedValue
private int userID;

```



Now we do not have to write the below statement:  
`userDetails.setUserID(1);` // Hibernate will take care of it.

### Console O/P:

```
Hibernate: select hibernate_sequence.nextval from dual
Hibernate: select hibernate_sequence.nextval from dual
Hibernate: insert into USER_DETAILS (address, description, joinedDate,
userName, userID) values (?, ?, ?, ?, ?)
Hibernate: insert into USER_DETAILS (address, description, joinedDate,
userName, userID) values (?, ?, ?, ?, ?)
```

`@GeneratedValue(strategy = GenerationType.AUTO)` → Let hibernate decide the strategy to come up with the primary key value

`@GeneratedValue(strategy = GenerationType.IDENTITY)` → Hibernate is going to use identity columns to generate unique primary keys. IDENTITY is a column feature provided in some of the databases.

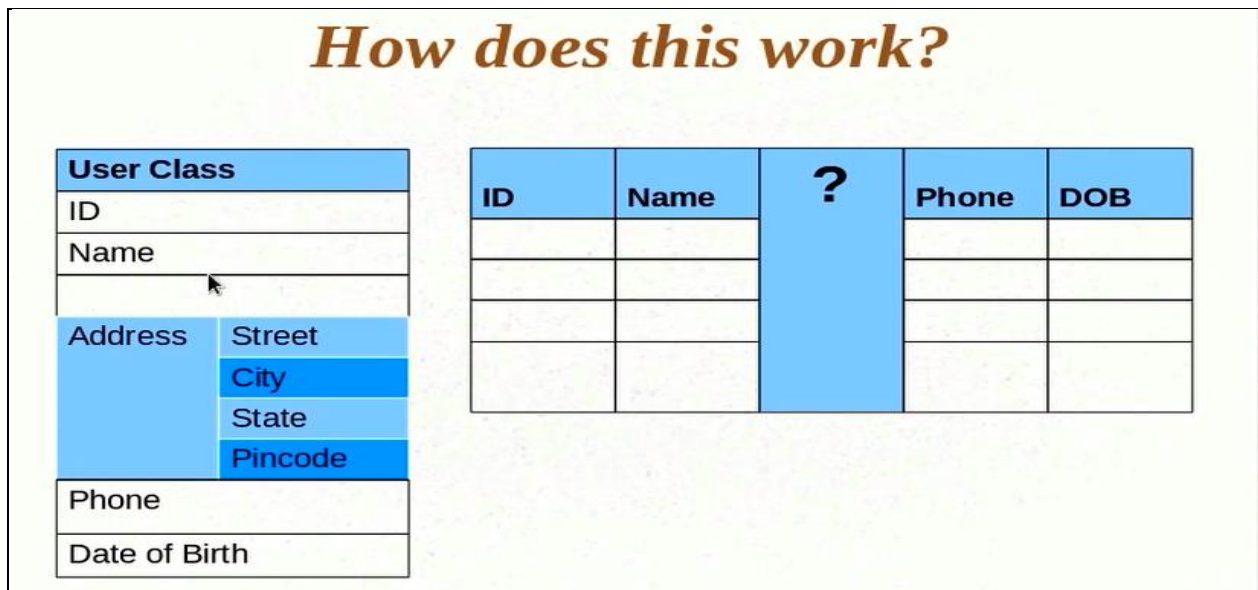
`@GeneratedValue(strategy = GenerationType.SEQUENCE)` → Uses SEQUENCE in database. SEQUENCE is something that database manages by themselves.

`@GeneratedValue(strategy = GenerationType.TABLE)` → You can have a separate table that may have record of the last primary key used so that you can increment it to get the next value. In this case, hibernate is going to create a separate column and use it.

---

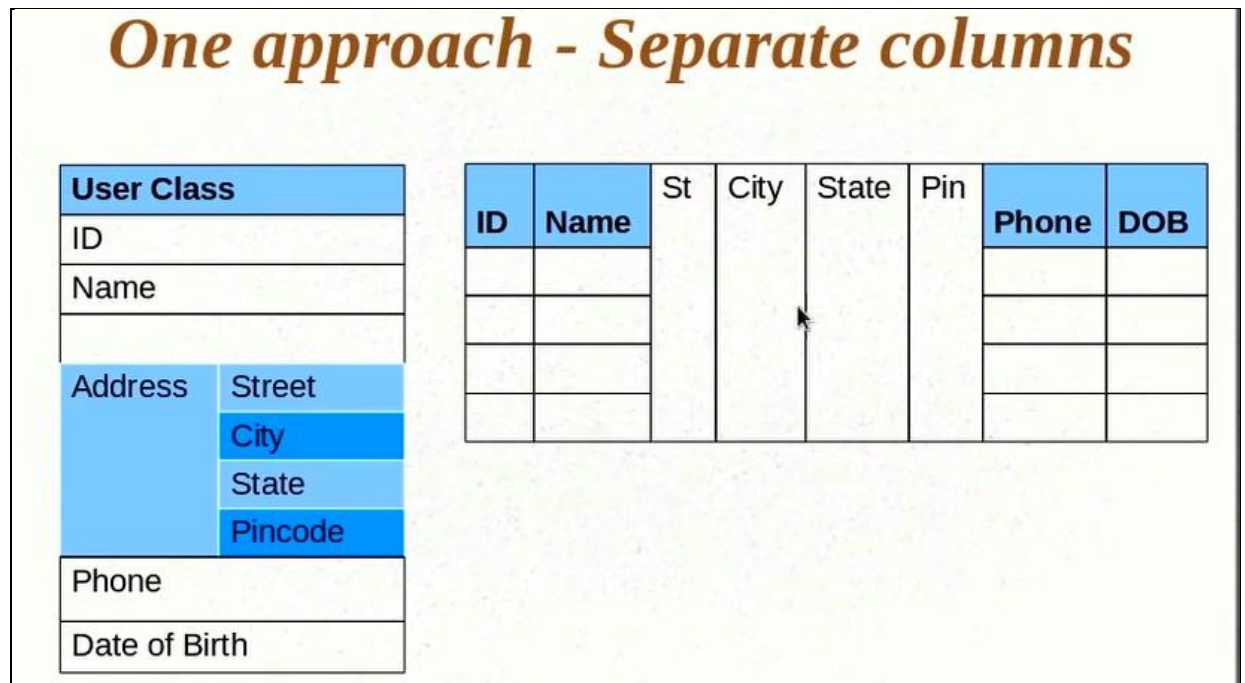
## Hibernate Tutorial 08 - Value Types and Embedding Objects

In the tutorials so far, we are assuming that all the member variables in java can go to database as a single column. What if one of the member variables is an object? How to persist them using hibernate?



(1) Simplest approach

Treat member variables of Address class as other member variables of the User class i.e. that the member variables of Address class will have separate columns in the User table. We will assume that there is no Address object, only the member variables as, Street, City, State, Pincode.



The above solution works fine if the Address object is a value object inside User class.

2 types of objects while dealing with hibernate:

- (1) Entity → An object that has to be saved as a separate entity in the database. It is independent and it contains data that provides meaning about itself. Ex: User
- (2) Value object → Has data and that has to be saved into database but it does not have meaning in itself. It provides meaning to some other object. Ex: Address

You can have an entity object within other entity object.

@Embeddable → Tells hibernate that this object needs to be embedded inside something else. Do not create a new table for this object.

@Embeddable

```
public class Address {

    private String street;
    private String city;
    private String state;
    private String pincode;

}
```

```

@Entity
@Table(name = "USER_DETAILS")
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int userID;
    private String userName;
    @Embedded // Another clue to hibernate that it is an
    embedded object. It is not mandatory. You just need to annotate
    the Address class with @Embeddable.

    private Address address;
}

```

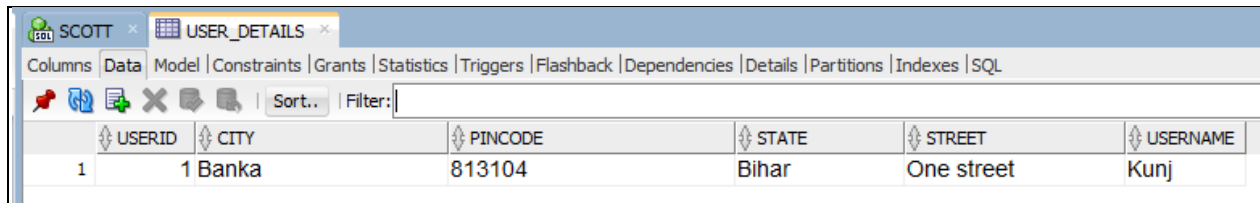
Console o/p:

```

Hibernate: select hibernate_sequence.nextval from dual
Hibernate: select hibernate_sequence.nextval from dual
Hibernate: insert into USER_DETAILS (city, pincode, state, street, userName,
userID) values (?, ?, ?, ?, ?, ?)
Hibernate: insert into USER_DETAILS (city, pincode, state, street, userName,
userID) values (?, ?, ?, ?, ?, ?)

```

Database table:



The screenshot shows the Oracle SQL Developer interface with the 'USER\_DETAILS' table selected. The table has six columns: USERID, CITY, PINCODE, STATE, STREET, and USERNAME. The data row shows USERID 1, CITY 1 Banka, PINCODE 813104, STATE Bihar, STREET One street, and USERNAME Kunj.

USERID	CITY	PINCODE	STATE	STREET	USERNAME
1	1 Banka	813104	Bihar	One street	Kunj

## Hibernate Tutorial 09 - Attribute Overrides and Embedded Object Keys

How to configure column names of an embedded object?

(1) First way (@Column="Column\_Name")

```

@Embeddable
public class Address {

    @Column(name = "STREET_NAME")
    private String street;
    @Column(name = "CITY_NAME")
    private String city;
    @Column(name = "STATE_NAME")
    private String state;
    @Column(name = "PIN_CODE")

```

```

    private String pincode;

}

```

(2) What if we have requirement for homeAddress and officeAddress? 2 address objects inside the same UserDetails class.

How to solve this? →

(a) By overriding the default name when we are embedding so that it does not take the defaults

```

@Entity
@Table(name = "USER_DETAILS")
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int userID;
    private String userName;

    @Embedded
    @AttributeOverrides({ @AttributeOverride(name = "street",
column = @Column(name = "HOME_STREET_NAME")),
        @AttributeOverride(name = "city", column =
@Column(name = "HOME_CITY_NAME")),
        @AttributeOverride(name = "state", column =
@Column(name = "HOME_STATE_NAME")),
        @AttributeOverride(name = "pincode", column =
@Column(name = "HOME_PIN_CODE")) })
    private Address homeAddress;

    @Embedded
    private Address officeAddress; // Similarly we can do for
officeAddress

}

```

Console o/p:

```

Hibernate: select hibernate_sequence.nextval from dual
Hibernate: insert into USER_DETAILS (HOME_CITY_NAME, HOME_PIN_CODE,
HOME_STATE_NAME, HOME_STREET_NAME, CITY_NAME, PIN_CODE, STATE_NAME,
STREET_NAME, userName, userID) values (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)

```

Database table:

	USERID	HOME_CITY_NAME	HOME_PIN_CODE	HOME_STATE_NAME	HOME_STREET_NAME	CITY_NAME	PIN_CODE	STATE_NAME	STREET_NAME	USERNAME
1	1	Banka	813104	Bihar	One street	(null)	(null)	(null)	(null)	Kunj

(b) If we have multiple such objects (for ex, homeAddress, officeAddress), we can give unique names to each of these objects so that there is no conflict. And then each of these objects can have a set of columns which are to be created for the address object.

*Special Case:* Where the primary key (userId) of a model object (UserDetails) can itself be an object. Let us say the combination of fields (name, ssn, address) put together of an object (User) forms a primary key. Below is the way to use an embedded object as a primary key:

`@EmbeddedId`

```
Private loginName userId; // LoginName has fields that put together forms a
primary key
```

Also, LoginName class has to be annotated with `@Embeddable`.

Even for `@EmbeddedId` we can define `@AttributeOverrides` and different columns.

---

## Hibernate Tutorial 10 - Saving Collections

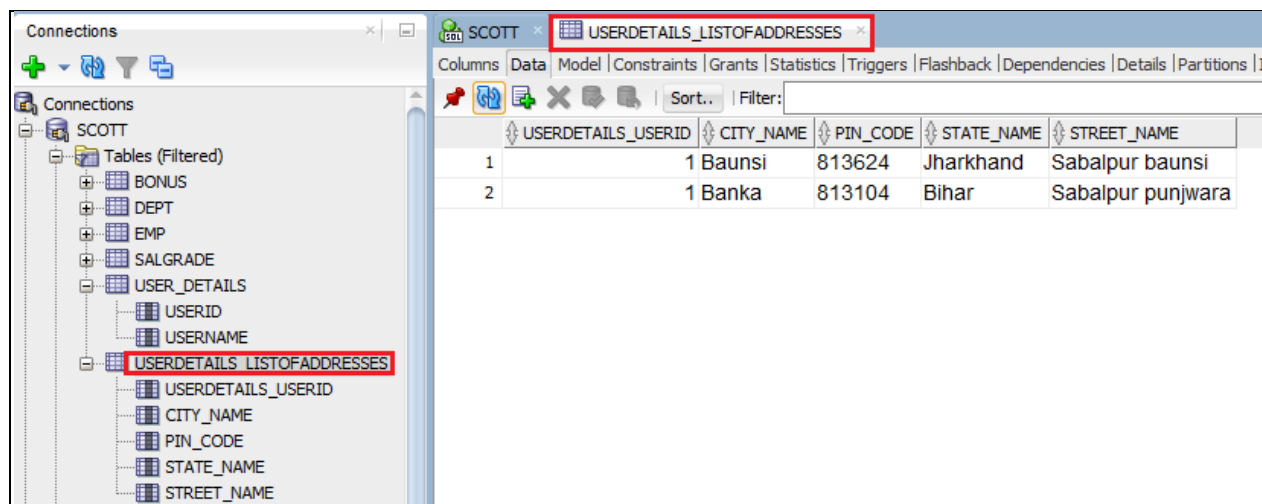
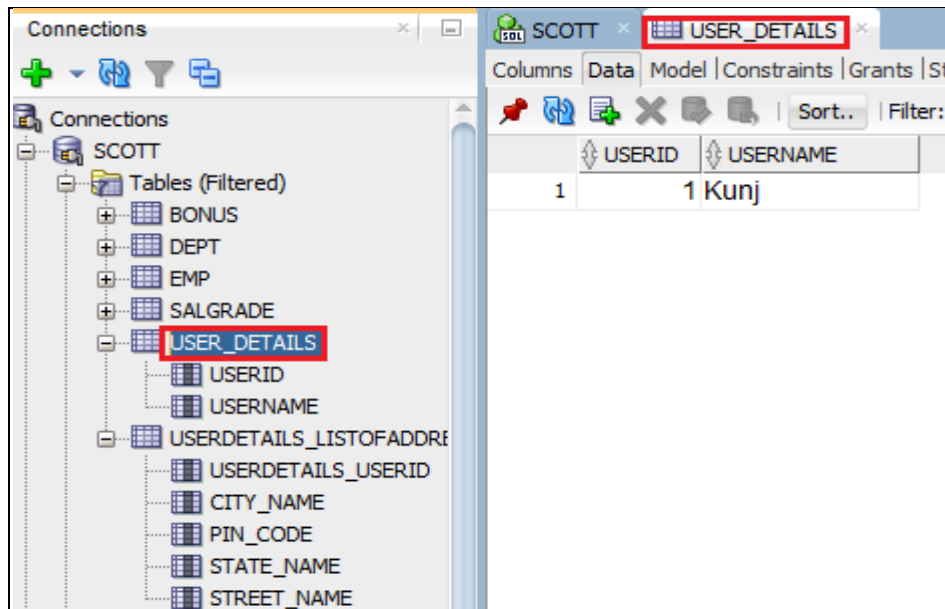
```
@Entity
@Table(name = "USER_DETAILS")
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int userID;
    private String userName;
    @ElementCollection // Tells hibernate that this member has
to be treated as list and I want it to be saved
    private Set<Address> listOfAddresses = new HashSet<>();
}
```

Console o/p:

```
Hibernate: select hibernate_sequence.nextval from dual
Hibernate: insert into USER_DETAILS (userName, userID) values (?, ?)
Hibernate: insert into UserDetails_listOfAddresses (UserDetails_userID,
CITY_NAME, PIN_CODE, STATE_NAME, STREET_NAME) values (?, ?, ?, ?, ?)
Hibernate: insert into UserDetails_listOfAddresses (UserDetails_userID,
CITY_NAME, PIN_CODE, STATE_NAME, STREET_NAME) values (?, ?, ?, ?, ?)
```

Database Table:



Pattern of database table name by default: ClassName\_MemberVarName

---

## Hibernate Tutorial 11 - Configuring Collections and Adding Keys

Q) How to change the default table name of the previous step?

```
@JoinTable(name = "USER_ADDRESS")
private Set<Address> listOfAddresses = new HashSet<>();
```

Note: Here the JoinTable is UserDetails\_listOfAddresses.

Console o/p:

```

Hibernate: select hibernate_sequence.nextval from dual
Hibernate: insert into USER_DETAILS (userName, userID) values (?, ?)
Hibernate: insert into USER_ADDRESS (UserDetails_userID, CITY_NAME, PIN_CODE,
STATE_NAME, STREET_NAME) values (?, ?, ?, ?, ?)
Hibernate: insert into USER_ADDRESS (UserDetails_userID, CITY_NAME, PIN_CODE,
STATE_NAME, STREET_NAME) values (?, ?, ?, ?, ?)

```

Q) How to change the default primary key name (UserDetails\_userID) of the previous step?

```

@JoinTable(name = "USER_ADDRESS",
    joinColumns = @JoinColumn(name = "USER_ID")
)
private Set<Address> listOfAddresses = new HashSet<>();

```

Q) We have a foreign key (USER\_ID) but there is no primary key in the table USER\_ADDRESS. What if we want to define an index/id for this table?

We want to define an id for a collection which is inside your entity class. In order to have an index column, we need to have a data type that supports index.

```

@ElementCollection
@JoinTable(name = "USER_ADDRESS",
    joinColumns = @JoinColumn(name = "USER_ID")
)
private Collection<Address> listOfAddresses = new ArrayList<>();

```

Also, we need to define primary key configuration.

```

@CollectionId(columns = { @Column }, generator = "", type =
@Type)
columns = {@Column (name=" What is the column that you want to
define as primary key")}

```

```

generator = "How is the primary key generated (name of the
generator type)"

```

```

type = @Type(type="type of the primary key")

```

Also, you need to define a generator.

```

@GenericGenerator(name = "hilo-gen", strategy = "hilo")

```

Hilo is common generator type that hibernate supplies.

```

@ElementCollection
@JoinTable(name = "USER_ADDRESS",
    joinColumns = @JoinColumn(name = "USER_ID")
)
@GenericGenerator(name = "hilo-gen", strategy = "hilo")

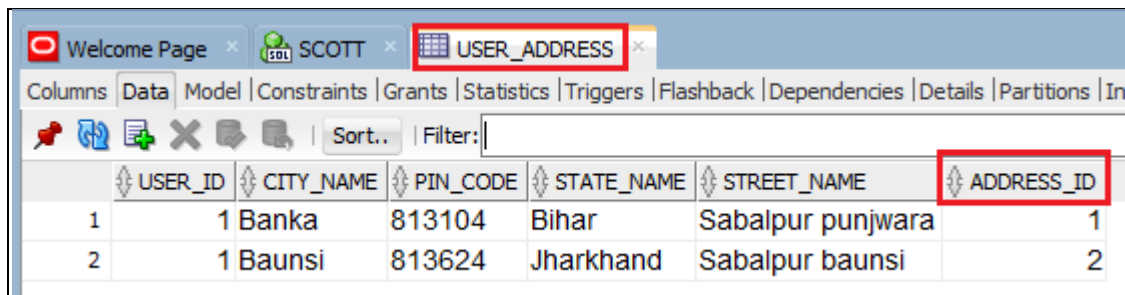
```

```
@CollectionId(columns = { @Column(name = "ADDRESS_ID") },
generator = "hilo-gen", type = @Type(type = "long"))
private Collection<Address> listOfAddresses = new ArrayList<>();
```

Cnsole o/p:

```
Hibernate: select hibernate_sequence.nextval from dual
Hibernate: insert into USER_DETAILS (userName, userID) values (?, ?)
Hibernate: insert into USER_ADDRESS (USER_ID, ADDRESS_ID, CITY_NAME,
PIN_CODE, STATE_NAME, STREET_NAME) values (?, ?, ?, ?, ?, ?)
Hibernate: insert into USER_ADDRESS (USER_ID, ADDRESS_ID, CITY_NAME,
PIN_CODE, STATE_NAME, STREET_NAME) values (?, ?, ?, ?, ?, ?)
```

Database Table:



	USER_ID	CITY_NAME	PIN_CODE	STATE_NAME	STREET_NAME	ADDRESS_ID
1	1	Banka	813104	Bihar	Sabalpur punjwara	1
2	1	Baunsi	813624	Jharkhand	Sabalpur baunsi	2

## HibernateTest.java

```
package com.kunj.hibernate;
```

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

```
import com.kunj.personal.dto.Address;
import com.kunj.personal.dto.UserDetails;
```

```
public class HibernateTest {
```

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

```
        UserDetails user = new UserDetails();
        user.setUserName("Kunj");
```

```
        Address address1 = new Address();
        address1.setStreet("Sabalpur punjwara");
        address1.setCity("Banka");
        address1.setState("Bihar");
        address1.setPincode("813104");
```



```

        Address address2 = new Address();
        address2.setStreet("Sabalpur baunsi");
        address2.setCity("Baunsi");
        address2.setState("Jharkhand");
        address2.setPincode("813624");

        user.getListOfAddresses().add(address1);
        user.getListOfAddresses().add(address2);

        SessionFactory sessionFactory = new
        Configuration().configure().buildSessionFactory();
        Session session = sessionFactory.openSession();
        session.beginTransaction();
        session.save(user);
        session.getTransaction().commit();
        session.close();
    }
}

```

## **UserDetails.java**

```

package com.kunj.personal.dto;

import java.util.ArrayList;
import java.util.Collection;

import javax.persistence.Column;
import javax.persistence.ElementCollection;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.Table;

import org.hibernate.annotations.CollectionId;
import org.hibernate.annotations.GenericGenerator;
import org.hibernate.annotations.Type;

```

```

@Entity
@Table(name = "USER_DETAILS")
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int userID;
    private String userName;

    @ElementCollection
    @JoinTable(name = "USER_ADDRESS", joinColumns = @JoinColumn(name
= "USER_ID"))
    @GenericGenerator(name = "hilo-gen", strategy = "hilo")
    @CollectionId(columns = { @Column(name = "ADDRESS_ID") }, generator =
"hilo-gen", type = @Type(type = "long"))
    private Collection<Address> listOfAddresses = new ArrayList<>();

    public void setListOfAddresses(Collection<Address> listOfAddresses) {
        this.listOfAddresses = listOfAddresses;
    }

    public Collection<Address> getListOfAddresses() {
        return listOfAddresses;
    }

    public int getUserID() {
        return userID;
    }

    public void setUserID(int userID) {
        this.userID = userID;
    }

    public String getUserName() {
        return userName;
    }
}

```

```

        public void setUserName(String userName) {
            this.userName = userName;
        }
    }
}

```

## **Address.java**

```

package com.kunj.personal.dto;

import javax.persistence.Column;
import javax.persistence.Embeddable;

@Embeddable
public class Address {

    @Column(name = "STREET_NAME")
    private String street;
    @Column(name = "CITY_NAME")
    private String city;
    @Column(name = "STATE_NAME")
    private String state;
    @Column(name = "PIN_CODE")
    private String pincode;

    public String getStreet() {
        return street;
    }

    public void setStreet(String street) {
        this.street = street;
    }

    public String getCity() {
        return city;
    }
}

```

```

    public void setCity(String city) {
        this.city = city;
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }

    public String getPincode() {
        return pincode;
    }

    public void setPincode(String pincode) {
        this.pincode = pincode;
    }
}

```

---

## Hibernate Tutorial 12 - Proxy Objects and Eager and Lazy Fetch Types

Q) How do I retrieve the persisted data from database in HibernateTest.java?

```

    user = null;
    session = sessionFactory.openSession();

```

// The below user object has list (listOfAddresses) inside it. What is the data getting pulled up when we do session.get()? The UserDetails object has listOfAddresses in it. Does the whole listOfAddresses get pulled up? If the ans is yes, then assume the user has 100 addresses. Now do all the addresses get pulled up when we retrieve user? If yes, then what is the cost/performance impact? Let us say, we are only interested in user\_name, then why do we need to pull all the addresses? To answer this qn we need to look at the fetch strategy. Hibernate employs a strategy to solve this problem. If you want to use just the user details, you can ask hibernate not to pull up all the address info. In fact, this is the default behavior of hibernate. So

by default, in the below command hibernate is not pulling out all the address info despite that fact that UserDetails has listOfAddresses member variable. But if use getter() method here, after the session.get() like, user.getListOfAddresses(), then hibernate again runs a query and fetches the list of addresses. Why and how hibernate does it?

WHY? → To save the fetch time. You get it only when you use it. This strategy is called Lazy initialization.

Lazy initialization → You do not initialize the entire object. You only initialize the first level member variables of the object.

Eager initialization → Fetch all the related data. This takes lot of resources/time. We should only do it when we need it.

HOW? → user.getListOfAddresses(). Here we are calling a getter() of user object. When this getter() runs, hibernate pulls out all the values. How is it possible?

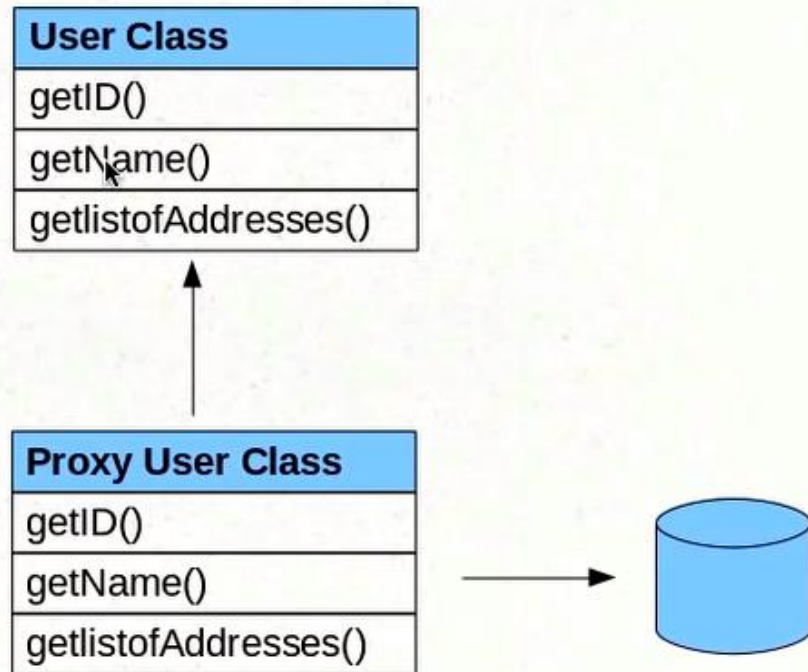
```
public Collection<Address> getListOfAddresses() {  
    return listOfAddresses;  
}
```

The getter() in the above line of code just return listOfAddresses. How can hibernate get all the values when we call user.getListOfAddresses? The way hibernate does it is by using proxy class. Hibernate instead of getting the actual object that you are looking for it gets you a proxy for the first time. It is a dynamic sub-class of your actual object.

Refer below diagram:

When you do session.get() of the UserDetails class, hibernate instead of giving you the actual class creates a proxy UserDetails class. It is a sub-class of the user class having the same methods. What proxy user class does is, that instead of fetching the values and populating the user class and sending back to user, it fills all the first level values, say listOfAddresses it does not get (bcz it is a huge list). It gets proxy user class created then it hands you the object of proxy user class, not the actual user class. You would not know bcz proxy User class implements the same methods as that of Actual user class. getID() and getName() in the proxy class will probably call the same parent methods.

# Hibernate Proxy



Now, assume that you make a call to **user.getListOfAddresses()**; Since you have the proxy object, you would be making a call to the method of the proxy class. **getListOfAddresses()** will have code to first fetch the data from DB. This is the first thing it does. **getListOfAddresses()** fetches the data from Db and populates the member variable. After it does that, it does a **parent.getListOfAddresses()**. This ensures 2 things:

- 1) You do not have to worry about what objects you are getting bcz it provides the correct data.

```
user = null;
session = sessionFactory.openSession();
user = (UserDetails) session.get(UserDetails.class, 1);
System.out.println(user.getListOfAddresses().size()); // 2
```

Now, in order to identify that the above user object is a proxy object, do the followings. We will close the session before the proxy object gets a chance to get the listofAddresses from DB. After the session is closed, the proxy object does not have a session for it to get the listofAddresses.

```
user = null;
```

```

session = sessionFactory.openSession();
user = (UserDetails) session.get(UserDetails.class, 1);
session.close();
System.out.println(user.getListOfAddresses().size());

```

Console o/p:

```

Exception in thread "main" org.hibernate.LazyInitializationException: failed
to lazily initialize a collection of role:
com.kunj.personal.dto.UserDetails.listOfAddresses, no session or session was
closed at
org.hibernate.collection.AbstractPersistentCollection.throwLazyInitialization
Exception(AbstractPersistentCollection.java:383) at
org.hibernate.collection.AbstractPersistentCollection.throwLazyInitialization
ExceptionIfNotConnected(AbstractPersistentCollection.java:375) at
org.hibernate.collection.AbstractPersistentCollection.readSize(AbstractPersis
tentCollection.java:122)
    at org.hibernate.collection.PersistentBag.size(PersistentBag.java:248)
    at com.kunj.hibernate.HibernateTest.main(HibernateTest.java:43)

```

There are 2 ways to solve the above problem:

1) If we need the session to be closed and then there is a possibility that we can access list and we want hibernate to pull up everything. Go to the @ElementCollection in UserDetails.java where we have defined the listOfAddresses to be an ElementCollection, we can configure hibernate to fetch listOfAddresses eagerly. In this case whenever I do a session.get() , it is going to fetch all the values.

```

@ElementCollection(fetch = FetchType.EAGER)
@JoinTable(name = "USER_ADDRESS",
    joinColumns = @JoinColumn(name = "USER_ID")
)

```

```

private Collection<Address> listOfAddresses = new ArrayList<>();

```

Now, if we run HibernateTest.java the console o/p is:

```

user = null;
session = sessionFactory.openSession();
user = (UserDetails) session.get(UserDetails.class, 1);
session.close();
System.out.println(user.getListOfAddresses().size());

```

```

Hibernate: select hibernate_sequence.nextval from dual
Hibernate: insert into USER_DETAILS (userName, userID) values (?, ?)
Hibernate: insert into USER_ADDRESS (USER_ID, CITY_NAME, PIN_CODE,
STATE_NAME, STREET_NAME) values (?, ?, ?, ?, ?)
Hibernate: insert into USER_ADDRESS (USER_ID, CITY_NAME, PIN_CODE,
STATE_NAME, STREET_NAME) values (?, ?, ?, ?, ?)
Hibernate: select userdetail0_.userID as userID0_0_, userdetail0_.userName as
userName0_0_, listofaddr1_.USER_ID as USER1_0_2_, listofaddr1_.CITY_NAME as
CITY2_2_, listofaddr1_.PIN_CODE as PIN3_2_, listofaddr1_.STATE_NAME as
STATE4_2_, listofaddr1_.STREET_NAME as STREET5_2_ from USER_DETAILS

```

```
userdetail0_ left outer join USER_ADDRESS listofaddr1_ on
userdetail0_.userID=listofaddr1_.USER_ID where userdetail0_.userID=?
2
```

The above o/p 2 is bcz of the eager fetch, even though the session is closed the eager configuration makes sure that the session.get() gets the entire object. Even though it returning with a proxy, we are not getting an actual object, we are getting a proxy but still it fetches all the values.

Why does it return proxy object even if the fetch type is eager?  
There could be another collection in UserDetails that hibernate will fetch lazily by default, while fetching other collections eagerly which are configured as eager fetch.

```
user = (UserDetails) session.get(UserDetails.class, 1);
```

---

## **Hibernate Tutorial 13 - One To One Mapping**

### **What happens if we have one entity inside other entity?**

Let us assume one to one mapping between user and vehicle.

Here Vehicle is a separate entity. It needs a separate table. We certainly need a mapping between UserDetails and Vehicle tables. The way hibernate does it is by having a column in the UserDetails table to point to the vehicle table.

### **HibernateTest.java**

```
package com.kunj.hibernate;

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

import com.kunj.personal.dto.UserDetails;
import com.kunj.personal.dto.Vehicle;

public class HibernateTest {

    public static void main(String[] args) {

        UserDetails user = new UserDetails();
        user.setUserName("Kunj");

        Vehicle vehicle = new Vehicle();
```



```
vehicle.setVehicleName("car");
```

user.setVehicle(vehicle); // To associate user with vehicle so that there is a mapping between the two

```
SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
Session session = sessionFactory.openSession();
session.beginTransaction();
session.save(user);
session.save(vehicle);
session.getTransaction().commit();
session.close();
    }
}
```

#### **UserDetails.java**

```
package com.kunj.personal.dto;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "USER_DETAILS")
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int userID;
    private String userName;
    @OneToOne
    private Vehicle vehicle;

    public Vehicle getVehicle() {
        return vehicle;
    }

    public void setVehicle(Vehicle vehicle) {
        this.vehicle = vehicle;
    }

    public int getUserID() {
```

```

        return userID;
    }

    public void setUserID(int userID) {
        this.userID = userID;
    }

    public String getUsername() {
        return userName;
    }

    public void setUsername(String userName) {
        this.userName = userName;
    }
}

```

### Console o/p:

```

Hibernate: select hibernate_sequence.nextval from dual
Hibernate: select hibernate_sequence.nextval from dual
Hibernate: insert into USER_DETAILS (userName, vehicle_vehicleID, userID)
values (?, ?, ?)
Hibernate: insert into Vehicle (vehicleName, vehicleID) values (?, ?)
Hibernate: update USER_DETAILS set userName=?, vehicle_vehicleID=? where
userID=?

```

*vehicle\_vehicleID in USER\_DETAILS maps to vehicleID in Vehicle table. In User\_Details table it has a reference to vehicleID, whatever primary key has been generated for vehicleID, it will insert with update command.*

Q) What if we want to change the column name from vehicle\_vehicleID to **VEHICLE\_ID**.

```

@Entity
@Table(name = "USER_DETAILS")
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int userID;
    private String userName;
    @OneToOne
    @JoinColumn(name="VEHICLE_ID") // TO CHANGE THE COLUMN NAME
    private Vehicle vehicle;
}

```

### Console o/p:

```

Hibernate: insert into USER_DETAILS (userName, VEHICLE_ID, userID) values (?,
?, ?)
Hibernate: insert into Vehicle (vehicleName, vehicleID) values (?, ?)

```

```
Hibernate: update USER_DETAILS set userName=?, VEHICLE_ID=? where userID=?
```

---

## **Hibernate Tutorial 14 - One To Many Mapping**

Single user object with multiple vehicle objects.

User to vehicle is OnetoMany relationship and vehicle to user is ManytoOne.

### **HibernateTest.java**

```
package com.kunj.hibernate;

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

import com.kunj.personal.dto.UserDetails;
import com.kunj.personal.dto.Vehicle;

public class HibernateTest {

    public static void main(String[] args) {

        UserDetails user = new UserDetails();
        user.setUserName("Kunj");

        Vehicle vehicle1 = new Vehicle();
        vehicle1.setVehicleName("car");

        Vehicle vehicle2 = new Vehicle();
        vehicle2.setVehicleName("car");

        user.getVehicle().add(vehicle1);
        user.getVehicle().add(vehicle2);

        SessionFactory sessionFactory = new
        Configuration().configure().buildSessionFactory();
        Session session = sessionFactory.openSession();
        session.beginTransaction();
```

```

        session.save(user);
        session.save(vehicle1);
        session.save(vehicle2);
        session.getTransaction().commit();
        session.close();
    }
}

```

### **UserDetails.java**

```

package com.kunj.personal.dto;

import java.util.ArrayList;
import java.util.Collection;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table(name = "USER_DETAILS")
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int userID;
    private String userName;
    @OneToMany
    private Collection<Vehicle> vehicle = new
    ArrayList<Vehicle>();

    public Collection<Vehicle> getVehicle() {
        return vehicle;
    }

    public void setVehicle(Collection<Vehicle> vehicle) {
        this.vehicle = vehicle;
    }

    public int getUserID() {
        return userID;
    }
}

```

```

    public void setUserID(int userID) {
        this.userID = userID;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }
}

```

### Vehicle.java

```

package com.kunj.personal.dto;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Vehicle {
    @Id
    @GeneratedValue
    private int vehicleID;
    private String vehicleName;

    public int getVehicleID() {
        return vehicleID;
    }

    public void setVehicleID(int vehicleID) {
        this.vehicleID = vehicleID;
    }

    public String getVehicleName() {
        return vehicleName;
    }

    public void setVehicleName(String vehicleName) {
        this.vehicleName = vehicleName;
    }
}

```

Console o/p:

```
Hibernate: insert into USER_DETAILS (userName, userID) values (?, ?)
Hibernate: insert into Vehicle (vehicleName, vehicleID) values (?, ?)
Hibernate: insert into Vehicle (vehicleName, vehicleID) values (?, ?)
Hibernate: insert into USER_DETAILS_Vehicle (USER_DETAILS_userID,
vehicle_vehicleID) values (?, ?)
Hibernate: insert into USER_DETAILS_Vehicle (USER_DETAILS_userID,
vehicle_vehicleID) values (?, ?)
```

**Q) How to override the column names** USER\_DETAILS\_userID and vehicle\_vehicleID and table name USER\_DETAILS\_Vehicle?

```
@OneToMany
@JoinTable(name = "USER_VEHICLE", joinColumns = @JoinColumn(name
= "USER_ID"), inverseJoinColumns = @JoinColumn(name =
"VEHICLE_ID"))
private Collection<Vehicle> vehicle = new ArrayList<Vehicle>();
```

Concole o/p:

```
Hibernate: insert into USER_DETAILS (userName, userID) values (?, ?)
Hibernate: insert into Vehicle (vehicleName, vehicleID) values (?, ?)
Hibernate: insert into Vehicle (vehicleName, vehicleID) values (?, ?)
Hibernate: insert into USER_VEHICLE (USER_ID, VEHICLE_ID) values (?, ?)
Hibernate: insert into USER_VEHICLE (USER_ID, VEHICLE_ID) values (?, ?)
```

**Note:** We can have a reverse relationship. As we have OneToMany for UserDetails, we can have ManyToOne for Vehicles table.

### HibernateTest.java

```
package com.kunj.hibernate;

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

import com.kunj.personal.dto.UserDetails;
import com.kunj.personal.dto.Vehicle;

public class HibernateTest {

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

```

        UserDetails user = new UserDetails();
        user.setUserName("Kunj");

        Vehicle vehicle1 = new Vehicle();
        vehicle1.setVehicleName("car");

        Vehicle vehicle2 = new Vehicle();
        vehicle2.setVehicleName("car");

        user.getVehicle().add(vehicle1);
        user.getVehicle().add(vehicle2);

        // Setting up 2 way relationship
        vehicle1.setUser(user);
        vehicle2.setUser(user);

        SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
        Session session = sessionFactory.openSession();
        session.beginTransaction();
        session.save(user);
        session.save(vehicle1);
        session.save(vehicle2);
        session.getTransaction().commit();
        session.close();
    }
}

```

### **UserDetails.java**

```

package com.kunj.personal.dto;

import java.util.ArrayList;
import java.util.Collection;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;

```

```

import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table(name = "USER_DETAILS")
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int userID;
    private String userName;
    @OneToMany
    @JoinTable(name = "USER_VEHICLE", joinColumns = @JoinColumn(name =
"USER_ID"), inverseJoinColumns = @JoinColumn(name = "VEHICLE_ID"))
    private Collection<Vehicle> vehicle = new ArrayList<Vehicle>();

    public Collection<Vehicle> getVehicle() {
        return vehicle;
    }

    public void setVehicle(Collection<Vehicle> vehicle) {
        this.vehicle = vehicle;
    }

    public int getUserID() {
        return userID;
    }

    public void setUserID(int userID) {
        this.userID = userID;
    }

    public String getUserName() {

```



```

        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }
}

```

### **Vehicle.java**

```

package com.kunj.personal.dto;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToOne;

@Entity
public class Vehicle {
    @Id
    @GeneratedValue
    private int vehicleID;
    private String vehicleName;
    @ManyToOne
    private UserDetails user;

    public UserDetails getUser() {
        return user;
    }

    public void setUser(UserDetails user) {
        this.user = user;
    }

    public int getVehicleID() {
        return vehicleID;
    }
}

```

```

    public void setVehicleID(int vehicleID) {
        this.vehicleID = vehicleID;
    }

    public String getVehicleName() {
        return vehicleName;
    }

    public void setVehicleName(String vehicleName) {
        this.vehicleName = vehicleName;
    }
}

```

#### Console o/p:

```

Hibernate: insert into USER_DETAILS (userName, userID) values (?, ?)
Hibernate: insert into Vehicle (user_userID, vehicleName, vehicleID) values
(?, ?, ?)
Hibernate: insert into Vehicle (user_userID, vehicleName, vehicleID) values
(?, ?, ?)
Hibernate: insert into USER_VEHICLE (USER_ID, VEHICLE_ID) values (?, ?)
Hibernate: insert into USER_VEHICLE (USER_ID, VEHICLE_ID) values (?, ?)

```

---

## Hibernate Tutorial 15 - mappedBy and Many To Many Mapping

There is one other way to represent OneToMany and ManyToOne relationship. The object on the many side of the relationship has a reference to the other object. The UserDetails cannot have a column for VEHICLE\_ID as one user can have multiple vehicles, but the Vehicle table can have a USER\_ID column bcz any vehicle will have only one user. It is ManyToOne as far as Vehicle side of the relationship is concerned. So we can have a USER\_ID column in Vehicle table and we can do the mapping there without having to have a separate table which has the mapping for USER\_ID and VEHICLE\_ID.

#### UserDetails.java

```

@Entity
@Table(name = "USER_DETAILS")
public class UserDetails {

```

```

@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private int userID;
private String userName;
/*
    * mapped by says where you want this mapping to happen. We
want mapping to
    * happen in user field of Vehicle class. i do not want a
separate table.
    */
@OneToMany(mappedBy = "user")
private Collection<Vehicle> vehicle = new
ArrayList<Vehicle>();
}

```

### Vehicle.java

```

@Entity
public class Vehicle {
    @Id
    @GeneratedValue
    private int vehicleID;
    private String vehicleName;

    @ManyToOne
    /*
        * Naming the join column that we are creating in the
Vehicle table. The below
        * annotation says that the mapping is for this below user
and join column is in
        * Vehicle table, so instead of creating a new table, it
will just create a
        * column inside vehicle table and it will have the id of
the user saved
        */
    @JoinColumn(name = "USER_ID")
    private UserDetails user;
}

```

Console o/p:

```

Hibernate: insert into USER_DETAILS (userName, userID) values (?, ?)
Hibernate: insert into Vehicle (USER_ID, vehicleName, vehicleID) values (?,
?, ?)
Hibernate: insert into Vehicle (USER_ID, vehicleName, vehicleID) values (?,
?, ?)

```

-----

### @ManyToMany

Just like we have a collection Vehicle object inside UserDetails, we need to have a collection user object inside Vehicle object.

```
@Entity
@Table(name = "USER_DETAILS")
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int userID;
    private String userName;
    @ManyToMany
    private Collection<Vehicle> vehicle = new
ArrayList<Vehicle>();

-----
@Entity
public class Vehicle {
    @Id
    @GeneratedValue
    private int vehicleID;
    private String vehicleName;
    @ManyToMany
    private Collection<UserDetails> userList = new ArrayList();
}
```

Console o/p:

```
Hibernate: insert into USER_DETAILS (userName, userID) values (?, ?)
Hibernate: insert into Vehicle (vehicleName, vehicleID) values (?, ?)
Hibernate: insert into Vehicle (vehicleName, vehicleID) values (?, ?)
Hibernate: insert into USER_DETAILS_Vehicle (USER_DETAILS_userID,
vehicle_vehicleID) values (?, ?)
Hibernate: insert into USER_DETAILS_Vehicle (USER_DETAILS_userID,
vehicle_vehicleID) values (?, ?)
Hibernate: insert into Vehicle_USER_DETAILS (Vehicle_vehicleID,
userList_userID) values (?, ?)
Hibernate: insert into Vehicle_USER_DETAILS (Vehicle_vehicleID,
userList_userID) values (?, ?)
```

In the above o/p, hibernate has created 2 relationship tables USER\_DETAILS\_Vehicle, Vehicle\_USER\_DETAILS. Hibernate does not know that it just need to do one mapping. We need to tell hibernate to map only one table. How?

Just pick one class. It may be UserDetails or Vehicle. Let us pick UserDetails class. We need to tell hibernate that this class will have the mapping from UserDetails to Vehicle.

Now go to the Vehicle.java and do the below:

```

@Entity
public class Vehicle {
    @Id
    @GeneratedValue
    private int vehicleID;
    private String vehicleName;
    @ManyToMany(mappedBy = "vehicle")// This ManyToMany is not
    doing the mapping
    private Collection<UserDetails> userList = new ArrayList();
}

```

```

@Entity
@Table(name = "USER_DETAILS")
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int userID;
    private String userName;
    @ManyToMany // This ManyToMany is doing the mapping

    private Collection<Vehicle> vehicle = new
    ArrayList<Vehicle>();
}

```

Console o/p:

```

Hibernate: insert into USER_DETAILS (userName, userID) values (?, ?)
Hibernate: insert into Vehicle (vehicleName, vehicleID) values (?, ?)
Hibernate: insert into Vehicle (vehicleName, vehicleID) values (?, ?)
Hibernate: insert into USER_DETAILS_Vehicle (userList_userID,
vehicle_vehicleID) values (?, ?)
Hibernate: insert into USER_DETAILS_Vehicle (userList_userID,
vehicle_vehicleID) values (?, ?)

```

@JoinTable has to be done in userDetails class.

---

## Hibernate Tutorial 16 - CascadeTypes and Other Things

Let us assume a ManyToOne relationship from vehicle to user. There is a possibility that the car may not have an owner. This vehicle is not yet rented or purchased. Imagine what will happen if your code is trying to do getUser() for a vehicle but there is no user corresponding to a vehicle. Hibernate will throw an exception. How to suppress this exception? By using the below line of code:

```
@NotFound(action = NotFoundAction.IGNORE)
```

```

@Entity
public class Vehicle {
    @Id
    @GeneratedValue
    private int vehicleID;
    private String vehicleName;
    @ManyToOne
    @NotFound(action = NotFoundAction.IGNORE) // if user is not
    found for a particular vehicle, just ignore it. Do not throw an
    exception
    private UserDetails user;
}

```

### Cascade:

In the example so far, user has vehicle object. We are explicitly saving user object as well as vehicle objects. What if we just save user object?

```
session.save(user);
```

Console o/p:

```

Hibernate: insert into USER_DETAILS (userName, userID) values (?, ?)
Hibernate: insert into USER_DETAILS_Vehicle (USER_DETAILS_userID,
vehicle_vehicleID) values (?, ?)
Exception in thread "main" org.hibernate.TransientObjectException: object
references an unsaved transient instance - save the transient instance before
flushing: com.kunj.personal.dto.Vehicle
    at
org.hibernate.engine.ForeignKeys.getEntityIdentifierIfNotUnsaved(ForeignKeys.
java:243)
    at org.hibernate.type.EntityType.getIdentifier(EntityType.java:456)
    at org.hibernate.type.ManyToOneType.nullSafeSet(ManyToOneType.java:121)
    at
org.hibernate.persister.collection.AbstractCollectionPersister.writeElement(A
bstractCollectionPersister.java:815)
    at
org.hibernate.persister.collection.AbstractCollectionPersister.recreate(Abstr
actCollectionPersister.java:1203)
    at
org.hibernate.action.CollectionRecreateAction.execute(CollectionRecreateActio
n.java:58)
    at org.hibernate.engine.ActionQueue.execute(ActionQueue.java:273)
    at
org.hibernate.engine.ActionQueue.executeActions(ActionQueue.java:265)
    at
org.hibernate.engine.ActionQueue.executeActions(ActionQueue.java:188)
    at
org.hibernate.event.def.AbstractFlushingEventListener.performExecutions(Abstr
actFlushingEventListener.java:321)

```

```

        at
org.hibernate.event.def.DefaultFlushEventListener.onFlush(DefaultFlushEventLi
stener.java:51)
        at org.hibernate.impl.SessionImpl.flush(SessionImpl.java:1216)
        at org.hibernate.impl.SessionImpl.managedFlush(SessionImpl.java:383)
        at
org.hibernate.transaction.JDBCTransaction.commit(JDBCTransaction.java:133)
        at com.kunj.hibernate.HibernateTest.main(HibernateTest.java:33)

```

The above o/p says that you have an object user but that has references another object vehicle that you have not saved. Why does hibernate not save the Vehicle object automatically? Bcz Vehicle is a separate entity. Hibernate cannot make the assumption and save the entity. That might be something that you want to do differently. Sometimes you do not want to save those entities, you want more control over it. If vehicle is a value type, hibernate will save it automatically.

Now the problem is that if you have huge list of vehicles, you need to do session.save() for all of the objects. This we can avoid by using the cascade. In UserDetails.java, we would like to tell hibernate that we do not want to save each and every object here, if you come across vehicle collection and if you have unsaved objects, when I am doing a save for user class, go and save all the objects in the vehicle collection. How to say it to hibernate?

```

@Entity
@Table(name = "USER_DETAILS")
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int userID;
    private String userName;
    @OneToMany(cascade = CascadeType.PERSIST) // When the user
is being saved, go ahead and save the unsaved vehicle collection
    private Collection<Vehicle> vehicle = new
ArrayList<Vehicle>();

    //getters and setters

}

```

In the HibernateTest.java we will write session.persist() in place of session.save(). This session.persist() ties in with the @OneToMany(cascade = CascadeType.[PERSIST](#)) change made in UserDetails.java. Whenever a persist happens, a cascade needs to happen.

Console o/p:

```
Hibernate: insert into USER_DETAILS (userName, userID) values (?, ?)
Hibernate: insert into Vehicle (vehicleName, vehicleID) values (?, ?)
Hibernate: insert into Vehicle (vehicleName, vehicleID) values (?, ?)
Hibernate: insert into USER_DETAILS_Vehicle (USER_DETAILS_userID,
vehicle_vehicleID) values (?, ?)
Hibernate: insert into USER_DETAILS_Vehicle (USER_DETAILS_userID,
vehicle_vehicleID) values (?, ?)
```

## **Hibernate Collections:**

### ***Hibernate Collections***

- Bag semantic – List / ArrayList
- Bag semantic with ID – List / ArrayList
- List semantic – List / ArrayList
- Set semantic – Set
- Map semantic – Map

---

## **Hibernate Tutorial 17 - Implementing Inheritance**

As there is no equivalent concept of inheritance in relational database, it is challenging to implement inheritance. We will use vehicle as the base class and use it in some other classes like TwoWheeler, fourWheeler etc. Why do we need inheritance? Why we cannot just make the TwoWheeler, fourWheeler classes as separate entities and save them as different tables. Let us take the below ex to understand the importance of inheritance.

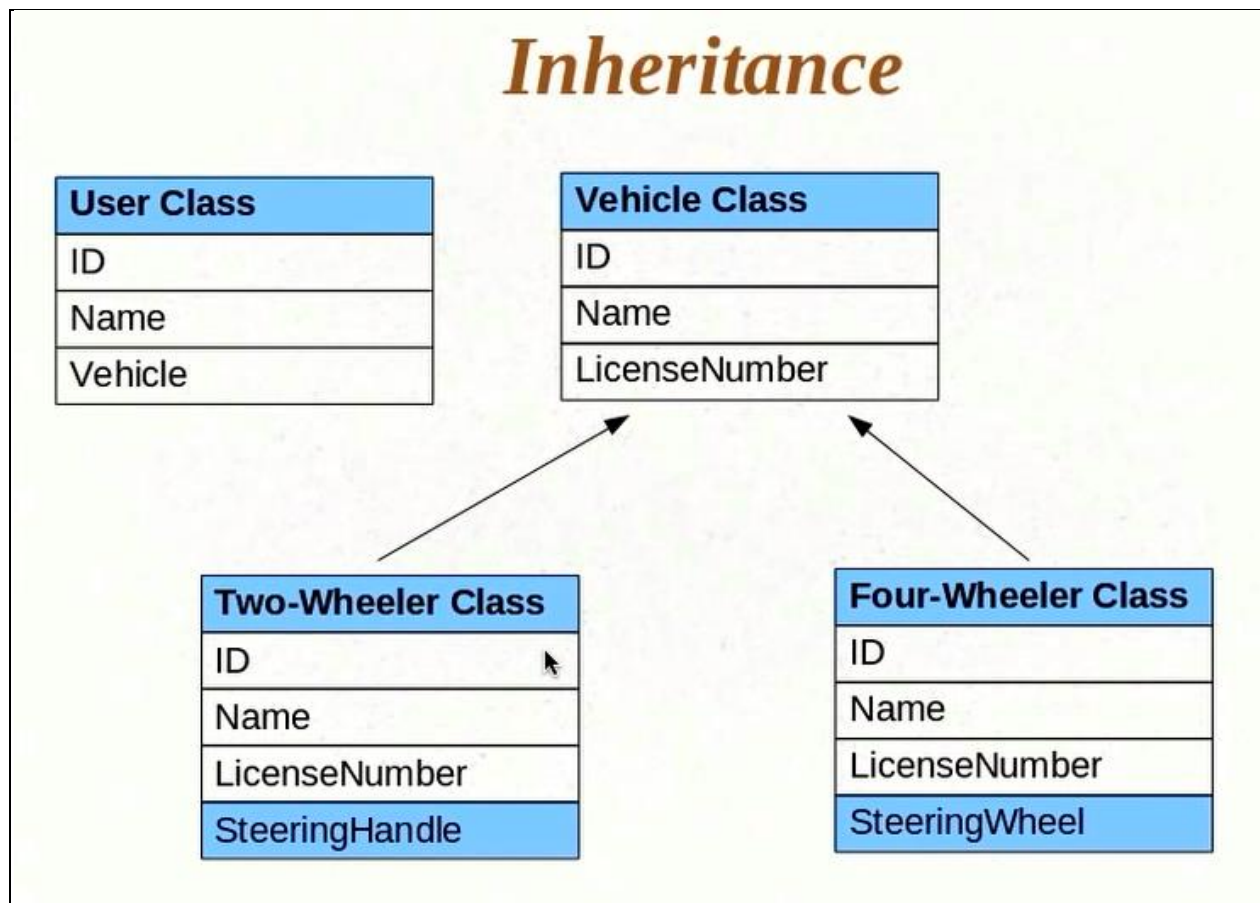
We have a base abstract Vehicle class. We cannot have an object of Vehicle class. We want to create objects of TwoWheeler and FourWheeler class and persist



them into DB. One option is to make these 2 classes as separate entities, where both the classes will go to DB as different tables. There are problems with this approach.

Q) What is one of the benefits of inheritance provided by java?

Ans: Polymorphism. Let us say we have a user object and this user object has a reference to vehicle class. I want to dynamically assign either a TwoWheeler or a FourWheeler object to the user class. The vehicle member var in the User class is a placeholder for any kind of vehicle and I would not know until runtime what it is going to be. Now the problem is, if we want to persist User class, it has a reference to let's say TwoWheeler. Now when I am saving this User class to DB, how would I map out this relationship? What will be the foreign key in the User class? Will I save the id of TwoWheeler or FourWheeler? I cannot have foreign key relationship to the both. While saving User class if it has a reference to TwoWheeler, then the vehicle in the user class will have a reference to TwoWheeler. So we cannot use polymorphism if we treat these 2 tables as separate entities.



The second reason to implement inheritance:

Let us say we change the vehicle class later on and add a different field there. The 2 classes inheriting from vehicle class will automatically get that new field inherited. But when it comes to the DB, you will not see the new column in the 2 tables bcz the Vehicle class is a separate entity/table/object. Whatever changes you do the Vehicle class will not be propagated to the 2 classes in the DB. You would have to do this manually.

First of all, declare all the 3 classes as separate entities and see what hibernate does by default for inheritance. There is nothing we have done to tell hibernate that we want to implement inheritance.

### **HibernateTest.java**

```
package com.kunj.hibernate;

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

import com.kunj.personal.dto.FourWheeler;
import com.kunj.personal.dto.TwoWheeler;
import com.kunj.personal.dto.Vehicle;

public class HibernateTest {

    public static void main(String[] args) {

        Vehicle vehicle = new Vehicle();
        vehicle.setVehicleName("car");

        TwoWheeler bike = new TwoWheeler();
        bike.setVehicleName("Bike");
        bike.setSteeringHandle("Bike steering handle");

        FourWheeler car = new FourWheeler();
        car.setVehicleName("Porsche");
```

```

        car.setSteeringWheel("Porsche steering wheel");

        SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
        Session session = sessionFactory.openSession();
        session.beginTransaction();
        session.save(vehicle);
        session.save(bike);
        session.save(car);

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

```

### **Vehicle.java**

```

package com.kunj.personal.dto;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Vehicle {
    @Id
    @GeneratedValue
    private int vehicleID;
    private String vehicleName;

    public int getVehicleID() {
        return vehicleID;
    }

    public void setVehicleID(int vehicleID) {
        this.vehicleID = vehicleID;
    }
}

```

```

    public String getVehicleName() {
        return vehicleName;
    }

    public void setVehicleName(String vehicleName) {
        this.vehicleName = vehicleName;
    }
}

```

TwoWheeler.java

```

package com.kunj.personal.dto;

import javax.persistence.Entity;

@Entity
public class TwoWheeler extends Vehicle {

    private String steeringHandle;

    public String getSteeringHandle() {
        return steeringHandle;
    }

    public void setSteeringHandle(String steeringHandle) {
        this.steeringHandle = steeringHandle;
    }

}

```

FourWheeler.java

```

package com.kunj.personal.dto;

import javax.persistence.Entity;

@Entity
public class FourWheeler extends Vehicle {

    // No @Id as it is inheriting from the Vehicle
    private String steeringWheel;

    public String getSteeringWheel() {

```

```

        return steeringWheel;
    }

    public void setSteeringWheel(String steeringWheel) {
        this.steeringWheel = steeringWheel;
    }
}

```

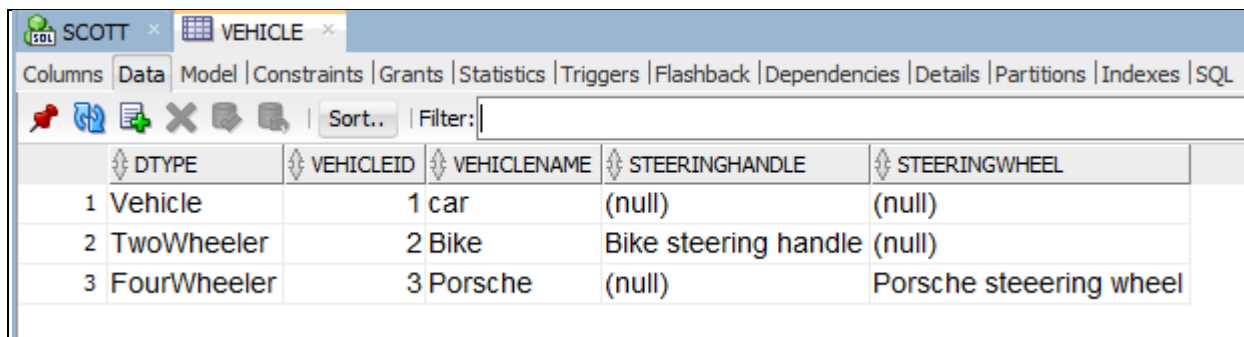
### Console o/p:

```

Hibernate: select hibernate_sequence.nextval from dual
Hibernate: select hibernate_sequence.nextval from dual
Hibernate: select hibernate_sequence.nextval from dual
Hibernate: insert into Vehicle (vehicleName, DTYPE, vehicleID) values (?, 'Vehicle', ?)
Hibernate: insert into Vehicle (vehicleName, steeringHandle, DTYPE, vehicleID) values (?, ?, 'TwoWheeler', ?)
Hibernate: insert into Vehicle (vehicleName, steeringWheel, DTYPE, vehicleID) values (?, ?, 'FourWheeler', ?)

```

### DB Table:



	DTYPE	VEHICLEID	VEHICLENAM	STEERINGHANDLE	STEERINGWHEEL
1	Vehicle	1	car	(null)	(null)
2	TwoWheeler	2	Bike	Bike steering handle	(null)
3	FourWheeler	3	Porsche	(null)	Porsche steering wheel

### Explanation of the above behavior:

Hibernate has created only one table vehicle even though we have marked TwoWheeler and FourWheeler as separate entities.. It has created a column called DTYPE. It is also inserting SteeringHandle and SteeringWheel to the same table. Hibernate has mapped everything to the base class table Vehicle. Hibernate has by default implemented a strategy for inheritance called Single table strategy, which means that no matter what your inheritance strategy is, it will create one common table for all the objects.

---

## Hibernate Tutorial 18 - Implementing Inheritance - Single Table Strategy

The DTYPE tells you what class it is. DTYPE is called discriminator. DTYPE identifies what type of the object you are saving to DB. Suppose we do not have an entry in

the SteeringHandle column for a Bike, then how would you discriminate between a bike and a car. DTYPE helps here. Single table strategy is Ok only for simple tables.

Q) How to configure inheritance strategy?

Ans: Go to the base class and annotate as below:

We can change the name and data type of the column DTYPE to a custom one. (Go to the parent class and annotate as shown below)

By default class names are getting populated in the DTYPE column. We can change that also. (Go to the child class and annotate as shown below)

**Vehicle.java**

```
package com.kunj.personal.dto;

import javax.persistence.DiscriminatorColumn;
import javax.persistence.DiscriminatorType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;

@Entity

@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
// Changing the Column name from DTYPE to custom one
@DiscriminatorColumn(name = "VEHICLE_TYPE", // NAME OF THE COLUMN
                    // datatype of the VEHICLE_TYPE column
                    discriminatorType = DiscriminatorType.STRING)

)

public class Vehicle {
    @Id
    @GeneratedValue
    private int vehicleID;
    private String vehicleName;
    // getters and setters
}
```

**TwoWheeler.java**

```

package com.kunj.personal.dto;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
@Entity
// This will replace default class name entry (TwoWheeler) in
DTYPE with Bike
@DiscriminatorValue("Bike")
public class TwoWheeler extends Vehicle {

    private String steeringHandle;
// Getters and setters
}

```

#### FourWheeler.java

```

package com.kunj.personal.dto;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
// This will replace default class name entry (FourWheeler) in
DTYPE with Car
@DiscriminatorValue("Car")
public class FourWheeler extends Vehicle {

    // No @Id as it is inheriting from the Vehicle
    private String steeringWheel;
// getters and setters
}

```

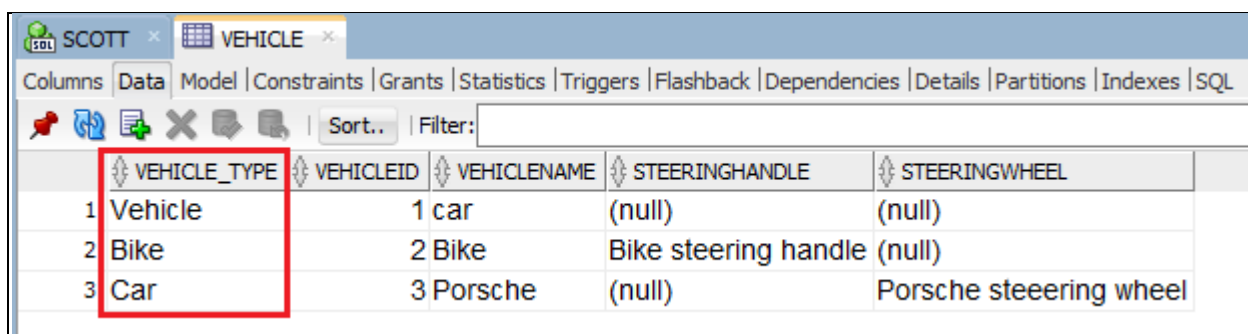
#### Console o/p:

```

Hibernate: insert into Vehicle (vehicleName, VEHICLE_TYPE, vehicleID) values
(?, 'Vehicle', ?)
Hibernate: insert into Vehicle (vehicleName, steeringHandle, VEHICLE_TYPE,
vehicleID) values (?, ?, 'Bike', ?)
Hibernate: insert into Vehicle (vehicleName, steeringWheel, VEHICLE_TYPE,
vehicleID) values (?, ?, 'Car', ?)

```

#### DB Table:



	VEHICLE_TYPE	VEHICLEID	VEHICLENAM	STEERINGHANDLE	STEERINGWHEEL
1	Vehicle	1	car	(null)	(null)
2	Bike	2	Bike	Bike steering handle	(null)
3	Car	3	Porsche	(null)	Porsche steering wheel

---

## Hibernate Tutorial 19 - Implementing Inheritance with Table Per Class Strategy

- 1) Separate tables for each of the classes that we are going to persist.
- 2) The parent class properties are inherited and they form separate columns even in the child tables.
- 3) The annotation (`@GeneratedValue`) for the id (`vehicleID`) is also inherited in the child class.
- 4) The advantage of this method is that you do not need a discriminator to identify the type of the object. A `TwoWheeler` table has only `TwoWheeler` object, same for `FourWheeler`. It is in normalized form i.e., you do not have extra columns that do not have data.

Console o/p:

```
Hibernate: insert into Vehicle (vehicleName, vehicleID) values (?, ?)
Hibernate: insert into TwoWheeler (vehicleName, steeringHandle, vehicleID)
values (?, ?, ?)
Hibernate: insert into FourWheeler (vehicleName, steeringWheel, vehicleID)
values (?, ?, ?)
```

DB Table:

Table	VehicleID	VehicleName	SteeringHandle	SteeringWheel
VEHICLE	1	1 car		
TWOWHEELER	1	2 Bike	Bike steering handle	
FOURWHEELER	1	3 Porsche		Porsche steering wheel

---

## Hibernate Tutorial 20 - Implementing Inheritance with Joined Strategy

*Single Table strategy*: Least normalized (Many column entries are null)



*Table per class strategy:* Better as there is no column that is not applicable (so no null value). But the problem here is that the parent table columns are repeating in all the child tables. This problem can be resolved by using the **JOINED** strategy.

In this strategy, in order to get the FourWheeler data, we need to look up into vehicle as well as FourWheeler table. Similarly for TwoWheeler. We will have to do a join to get all the data for a sub-class.

The child class specific columns are going to go into child class tables.

Console o/p:

```
Hibernate: insert into Vehicle (vehicleName, vehicleID) values (?, ?)
Hibernate: insert into Vehicle (vehicleName, vehicleID) values (?, ?)
Hibernate: insert into TwoWheeler (steeringHandle, vehicleID) values (?, ?)
Hibernate: insert into Vehicle (vehicleName, vehicleID) values (?, ?)
Hibernate: insert into FourWheeler (steeringWheel, vehicleID) values (?, ?)
```

DB Tables:

SCOTT × VEHICLE ×	
VEHICLEID	VEHICLENAME
1	1 car
2	2 Bike
3	3 Porsche

SCOTT × TWOWHEELER ×	
STEERINGHANDLE	VEHICLEID
1 Bike steering handle	2

SCOTT × FOURWHEELER ×	
STEERINGWHEEL	VEHICLEID
1 Porsche steering wheel	3

---

## Hibernate Tutorial 21 - CRUD Operations

---

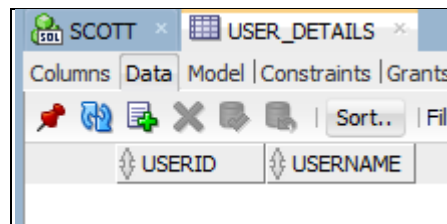
## Hibernate Tutorial 22 - Transient, Persistent and Detached Objects

If we do not do `session.save(object)` for a particular object i.e.; we are not asking hibernate to save this particular object. In this case the object is said to be a transient object. On the other hand if we do `session.save(object)` for a particular

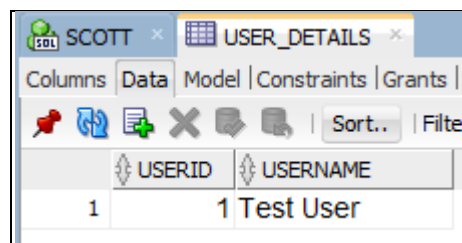
object, the object is said to be a persistent object. When an object is persistent, hibernate tracks that object and saves it.

In HibernateTest.java:

If we do not do `session.save(user)`, the UserDetails table will not have any record.



Now if we do `session.save(user)`.



What if we change the properties of an object after the object is saved?  
package com.kunj.hibernate;

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

```
import com.kunj.personal.dto.UserDetails;
```

```
public class HibernateTest {
```

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

```
        UserDetails user = new UserDetails();  
        user.setUserName("Test User");
```

```

SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
Session session = sessionFactory.openSession();
session.beginTransaction();

    session.save(user);
    // User is updated after the session.save() operation
    user.setUsername("Updated User");

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

```

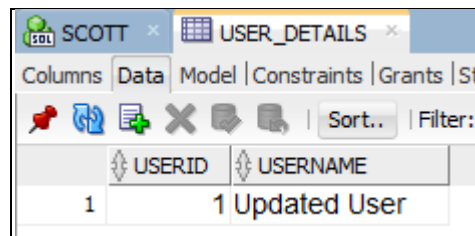
Console o/p:

```

Hibernate: select hibernate_sequence.nextval from dual
Hibernate: insert into USER_DETAILS (userName, userID) values (?, ?)
Hibernate: update USER_DETAILS set userName=? where userID=?

```

DB Table:



USERID	USERNAME
1	1 Updated User

**Note:** Even if we are not triggering any save or update command, hibernate is updating the already saved data when we update the attribute of the object (user).

Q) How does hibernate know what changes to get reflected in the database?

Ans: Once you pass an entity object to session.save(), after that any change you make to that object will go as an update statement to the database.

Q) What if you do the following?

```

user.setUsername("Updated User");
user.setUsername("Updated User 1");

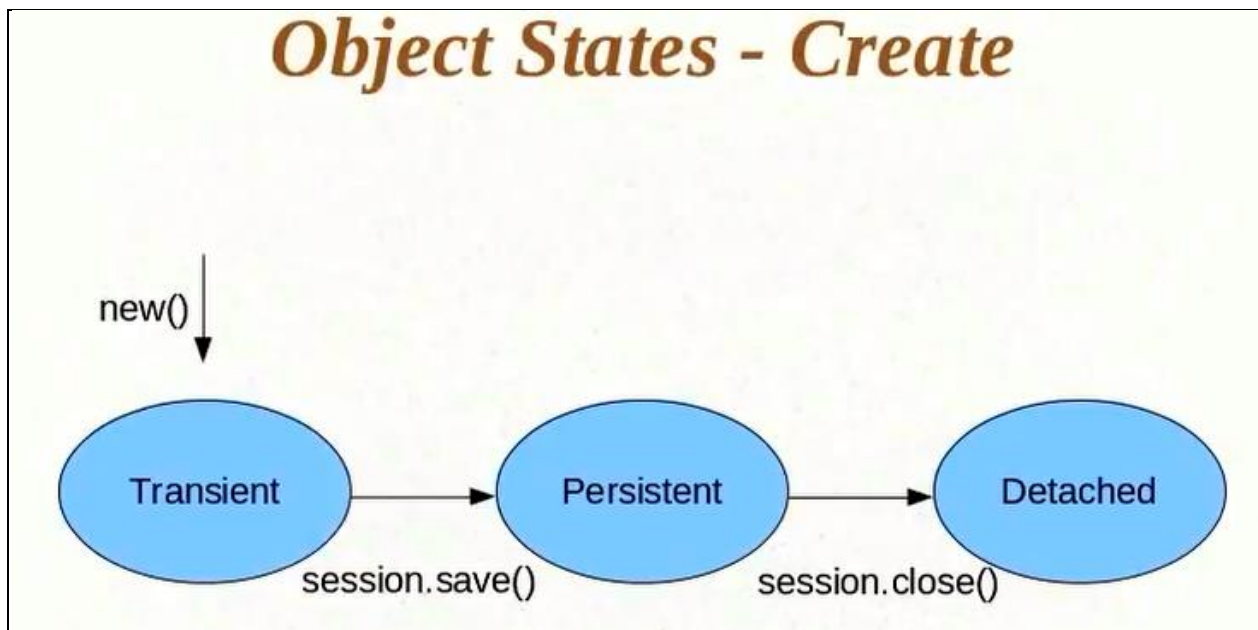
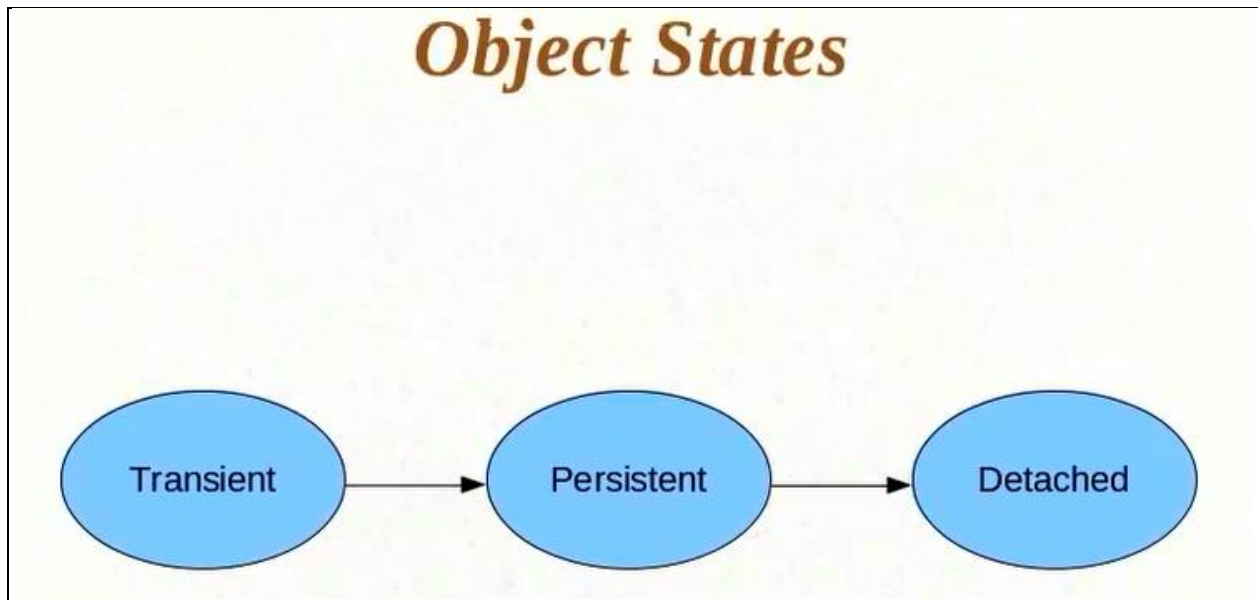
```

Ans: Hibernate will only fire the one update statement in the above case. Hibernate intelligently detects the least number of updates to be made.

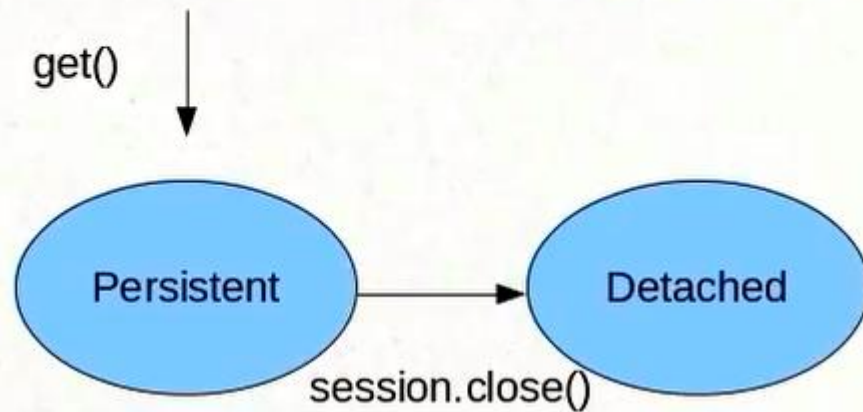
Once we do `session.close()` the object becomes detached. Detached objects are similar to the transient object in the sense that hibernate is not going to track the changes. Detached object means that it was tracked by hibernate before, now that the session has been closed it is no longer tracked by hibernate and it is in a detached state. In the detached state hibernate will not persist any changes made to the object into the DB.

---

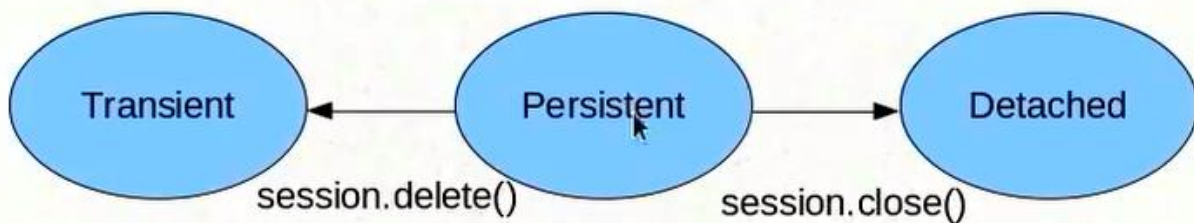
## Hibernate Tutorial 23 - Understanding State Changes



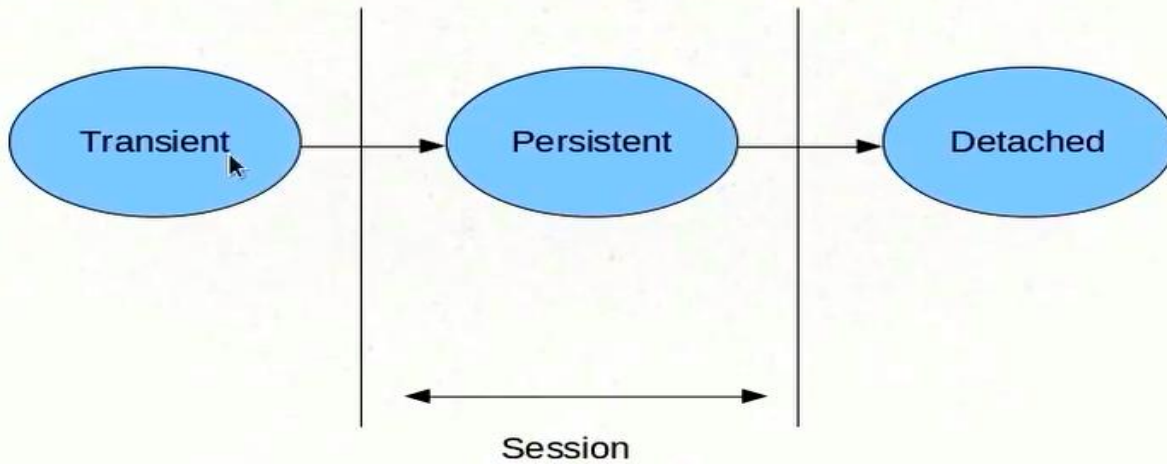
## ***Object States - Read***



## ***Object States - Delete***



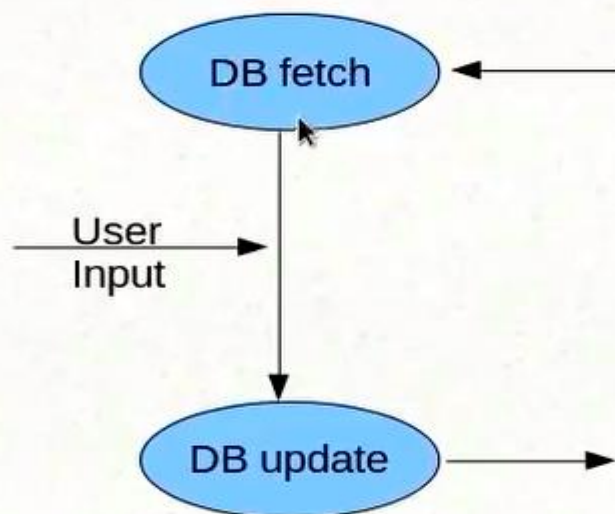
## *Object States*



---

### Hibernate Tutorial 24 - Persisting Detached Objects

## *Detached to Persistent*



In the above fig, we have fetched some data from the DB and rendered some page for user to input values. We need to take the user i/p and update the DB. The problem here is that user might take some time to enter the values. So we need to wait for that duration of time. It is not a good idea to do the above operations in a single transaction. So we will open a session and have one transaction and get the data, then close the transaction and close the session. And then let the user take his own time and when we get the user i/p we will open another session and do the update.

The problem is, the moment I close the session after DB fetch, the object goes into detached state. Now I wait for the user i/p, and then I need the detached object back into persistent state in order to save that into DB.

How do I move an object from detached state to persistent state?

### **HibernateTest.java**

```
package com.kunj.hibernate;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import com.kunj.personal.dto.UserDetails;

public class HibernateTest {

    public static void main(String[] args) {
        SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
        Session session = sessionFactory.openSession();
        session.beginTransaction();
        UserDetails user = (UserDetails) session.get(UserDetails.class, 1);
        session.getTransaction().commit();
        session.close();
        // User is providing some i/p to update the values after session close
        user.setUserName("Updated user name after session close");

        // trying to bring a detached object to persistent state
        session = sessionFactory.openSession();
        session.beginTransaction();
```

```

        session.update(user);
        session.getTransaction().commit();
        session.close();
    }
}

```

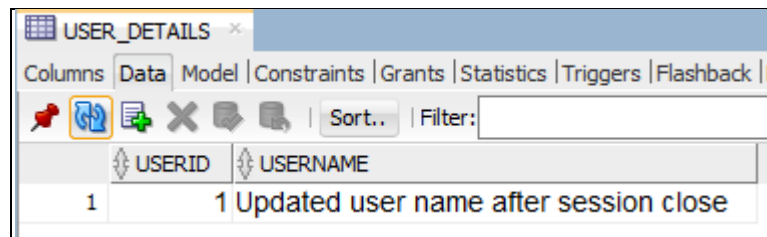
#### Console o/p:

```

Hibernate: select userdetail0_.userID as userID0_0_, userdetail0_.userName as
userName0_0_ from USER_DETAILS userdetail0_ where userdetail0_.userID=?
Hibernate: update USER_DETAILS set userName=? where userID=?

```

#### DB Table:



USERID	USERNAME
1	1 Updated user name after session close

#### Points:

- 1) If we make changes to user object after session.update(user) is fired, hibernate is automatically going to persist it.

```

session.update(user);

user.setUserName("Changed after update");

```

- 2) Let us say we do not make any changes to user object after we execute the below command:

```

UserDetails user = (UserDetails) session.get(UserDetails.class, 1);

```

And then run session.update(user);

#### Console o/p:

```

Hibernate: select userdetail0_.userID as userID0_0_, userdetail0_.userName as
userName0_0_ from USER_DETAILS userdetail0_ where userdetail0_.userID=?
Hibernate: update USER_DETAILS set userName=? where userID=?

```

Note that even though we have not made any changes to user object, hibernate runs the update query. Why? Hibernate does not know if anything has changed in the user object. As long as user object is persisted, hibernate keeps track of changes. Here, since the object is coming from another session, hibernate does



not know the value saved in the DB already. Unless it does an update, it cannot be sure that the current value of the user gets reflected to the record in the table.

There is a way to avoid this. If you want to make sure that the object has changed before doing an update. You have to ask hibernate to first do a select and see if the existing user object is same or different from the object already saved in the DB and trigger an update query only if the object in the DB is different from the current one. How to achieve this?

Annotate UserDetails class with the below:

```
@org.hibernate.annotations.Entity(selectBeforeUpdate = true)
```

---

## **Hibernate Tutorial 25 - Introducing HQL and the Query Object**

In SQL while getting data we think about tables, but in HQL we think about objects. We need to use query object while writing HQL.

### **HibernateTest.java**

```
package com.kunj.hibernate;

import java.util.List;

import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import com.kunj.personal.dto.UserDetails;

public class HibernateTest {

    public static void main(String[] args) {

        SessionFactory sessionFactory = new
        Configuration().configure().buildSessionFactory();

        Session session = sessionFactory.openSession();
```

```

        session.beginTransaction();

        // In place of Table Name and column Name we pass Class Name and
        property Name

        Query createQuery = session.createQuery("from UserDetails");

        List<UserDetails> list = (List<UserDetails>) createQuery.list();

        session.getTransaction().commit();

        session.close();

        for (UserDetails u : list)

            System.out.println(u.getUserName());

    }
}

```

The good thing about the above program is that hibernate pulls the data and automatically provides us the entity object. But the problems are:

- 1) I do not want all the users to be retrieved. I want to do pagination. I want to get the first set of users and fire another query to get the second set of users and so on.

### Pagination:

```

Query query = session.createQuery("from UserDetails");
// What is the start (row no) of the result I am interested in.

        query.setFirstResult(5);
// What is the maximum no of records hibernate needs to pull up.

        query.setMaxResults(4);

```

- 2) We do not have much control over it. We do not have control over what columns gets pulled up. You do not always want to pull all the columns. You can do this by writing the whole select query like below:

### HibernateTest.java

```

package com.kunj.hibernate;

```

```

import java.util.List;

import org.hibernate.Query;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

public class HibernateTest {

    public static void main(String[] args) {

        SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();

        Session session = sessionFactory.openSession();

        session.beginTransaction();

        Query query = session.createQuery("select userName from
UserDetails");

        query.setFirstResult(5);

        query.setMaxResults(4);

        List<String> userNames = (List<String>) query.list();

        session.getTransaction().commit();

        session.close();

        for (String u : userNames)

            System.out.println(u);

    }

}

```

Console o/p:

```

Hibernate: select * from ( select row_.*, rownum rownum_ from ( select
userdetail0_.userName as col_0_0_ from USER_DETAILS userdetail0_ ) row_ where
rownum <= ?) where rownum_ > ?

```

```
User4  
User5  
User6  
User7
```

**Note:**

// This will return list of map.

```
session.createQuery("select new map(userID, username) from UserDetails");
```

// We can also use all the aggregate functions

```
session.createQuery("select max(userID) from UserDetails");
```

---

## **Hibernate Tutorial 27 - Understanding Parameter Binding and SQL Injection**

### **Parameter Binding:**

It is a way in which you can bind specific values to parameters in your sql query.

### **SQL Injection example:**

#### **HibernateTest.java**

```
package com.kunj.hibernate;  
  
import java.util.List;  
  
import org.hibernate.Query;  
  
import org.hibernate.Session;  
  
import org.hibernate.SessionFactory;  
  
import org.hibernate.cfg.Configuration;  
  
import com.kunj.personal.dto.UserDetails;  
  
public class HibernateTest {  
  
    public static void main(String[] args) {  
  
        SessionFactory sessionFactory = new  
Configuration().configure().buildSessionFactory();
```

```

        Session session = sessionFactory.openSession();
        session.beginTransaction();

        String minuserID = " 5 or 1=1 ";

        Query query = session.createQuery("from UserDetails where userID >
" + minuserID);

        List<UserDetails> users = (List<UserDetails>) query.list();

        session.getTransaction().commit();

        session.close();

        for (UserDetails u : users)

            System.out.println(u.getUserName());

    }
}

```

The above program will pull all the records from UserDetails table.

The solution to the above problem is the parameter substitution/binding.

#### (1) 1<sup>st</sup> way

```

String minuserID = "5";
Query query = session.createQuery("from UserDetails where userID
> ?");
query.setInteger(0, Integer.parseInt(minuserID));

```

#### (2) 2<sup>nd</sup> way

```

String minuserID = "5";
Query query = session.createQuery("from UserDetails where userID
> :userID");
query.setInteger("userID", Integer.parseInt(minuserID));

```

---

### **Hibernate Tutorial 28 - Named Queries**

It is usually best practice to consolidate all queries to one common place.

Named Queries in the way to consolidate all the queries at one place. It allows you to write the queries at entity level.

### **UserDetails.java**

```
package com.kunj.personal.dto;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQuery;
import javax.persistence.Table;

@Entity
@NamedQuery(name = "UserDetails.byId", query = "from UserDetails
where userId=?")
@Table(name = "USER_DETAILS")
public class UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int userID;
    private String userName;
    // getters and setters
}
```

### **HibernateTest.java**

```
package com.kunj.hibernate;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import com.kunj.personal.dto.UserDetails;

public class HibernateTest {

    public static void main(String[] args) {

        SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
        Session session = sessionFactory.openSession();
        session.beginTransaction();

        Query query =
session.getNamedQuery("UserDetails.byId");
```

```

        query.setInteger(0, 12);
        List<UserDetails> users = (List<UserDetails>)
query.list();
        session.getTransaction().commit();
        session.close();
        for (UserDetails u : users)
            System.out.println(u.getUserName());
    }
}

```

Note: In the above program, within the `@NamedQuery` annotation, we are using HQL. We also have a way to use native SQL query. See below:

```
@NamedNativeQuery(name = "UserDetails.byName", query = "select * from
User_Details where username=?", resultClass = UserDetails.class)
```

Here we use table name instead of class name. Also we use `resultClass` to specify what the class of the resultset is. In `@NamedQuery`, we do not need to specify `resultClass` as the query already has class name in it. The advantage here is that if you have a stored procedure to run, you can run this way. Otherwise, it is always advantageous to go HQL way. Bcz the whole point of using hibernate is go away from SQL.

## Hibernate Tutorial 29 - Introduction to Criteria API

Another way to extract data from the DB.

Problems with HQL:

- 1) HQL after a point is not very different from SQL. As the query becomes more complex, it becomes tougher to maintain. Making a change in a complex query is difficult.

A criteria is more like a where clause where you say that these are the restrictions that I want to impose. Criteria object needs to know which object you are applying restrictions on.

**HibernateTest.java**

```

package com.kunj.hibernate;
import java.util.List;
import org.hibernate.Criteria;
import org.hibernate.Session;

```

```

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import org.hibernate.criterion.Restrictions;
import com.kunj.personal.dto.UserDetails;

public class HibernateTest {

    public static void main(String[] args) {

        SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
        Session session = sessionFactory.openSession();
        session.beginTransaction();
        // Creating criteria for UserDetails.class
        Criteria criteria =
session.createCriteria(UserDetails.class);
        criteria.add(Restrictions.eq("userName", "User 1"));

        List<UserDetails> users = (List<UserDetails>)
criteria.list();
        session.getTransaction().commit();
        session.close();
        for (UserDetails u : users)
            System.out.println(u.getUserName());
    }
}

```

Console o/p:

```

Hibernate: select this_.userID as userID0_0_, this_.userName as userName0_0_
from USER_DETAILS this_ where this_.userName=?
User 1

```

---

## Hibernate Tutorial 30 - Understanding Restrictions

Criteria supports chaining. We can keep on adding restrictions in the same criteria objects.

```

// Creating criteria for UserDetails.class
Criteria criteria = session.createCriteria(UserDetails.class);

criteria.add(Restrictions.eq("userName", "User
1")).add(Restrictions.gt("UserID", 15))
.add(Restrictions.between("userID", 2, 20));

```

Note: All the above criteria.add is doing "AND". What if you want to do an "OR"?



```
criteria.add(Restrictions.or(Restrictions.gt("UserID", 15),
Restrictions.between("userID", 2, 20)));
```

---

## **Hibernate Tutorial 31 - Projections and Query By Example**

**These are also criteria API features.**

At a very high level, one of the uses of projections is to implement any aggregation/grouping function.

// This will only return user ID

```
Criteria criteria =
session.createCriteria(UserDetails.class).setProjection(Projections.property("userID"));
```

```
Criteria criteria =
session.createCriteria(UserDetails.class).setProjection(Projections.max("userID"));
```

```
Criteria criteria =
session.createCriteria(UserDetails.class).setProjection(Projections.count("userID"));
```

You can also sort using projections.

```
Criteria criteria =
session.createCriteria(UserDetails.class).addOrder(Order.desc("userID"));
```

### **Query By Example:**

It is handy when you have too many properties/criteria values to satisfy. Let us say we have an object that has 10 fields and you have values for 5 fields. Let us say we want to pull up all the records with id 1 with age 20 and pincode 1000 etc. So I have specific values for half of the properties. Now, creating a criteria object for all of them is much task. A handy way to do this is query by example.

### **HibernateTest.java**

```

package com.kunj.hibernate;

import java.util.List;

import org.hibernate.Criteria;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import org.hibernate.criterion.Example;
import com.kunj.personal.dto.UserDetails;

public class HibernateTest {

    public static void main(String[] args) {

        SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();

        Session session = sessionFactory.openSession();
        session.beginTransaction();

        UserDetails exampleUser = new UserDetails();
        exampleUser.setUserID(12);
        exampleUser.setUserName("User 1");
        // create an example of exampleUser object
        Example example = Example.create(exampleUser);

        Criteria criteria =
session.createCriteria(UserDetails.class).add(example);

        List<UserDetails> users = (List<UserDetails>) criteria.list();
        session.getTransaction().commit();
        session.close();

        for (UserDetails u : users)

```

```

        System.out.println(u.getUserName());
    }
}

```

### Console o/p:

```

Hibernate: select this_.userID as userID0_0_, this_.userName as userName0_0_
from USER_DETAILS this_ where (this_.userName=?)
User 1

```

Q) What if we comment the lines one by one?

```
// exampleUser.setUserName("User 1");
```

### Console o/p:

```

Hibernate: select this_.userID as userID0_0_, this_.userName as userName0_0_
from USER_DETAILS this_ where (1=1)
User 0
User 1
User 2
User 3
User 4
User 5
User 6
User 7
User 8
User 9

```

```
// exampleUser.setUserID(12);
```

### Console o/p:

```

Hibernate: select this_.userID as userID0_0_, this_.userName as userName0_0_
from USER_DETAILS this_ where (this_.userName=?)
User 1

```

Why so? → The concept here is that hibernate ignores 2 things when it comes to taking an example.

- (1) If any of the properties has a value of null, it is not going to consider that.
- (2) If a property happens to be primary key, it is not going to consider that.

Suppose we have 10 properties in an object and we know that some property might have a value and we do not want hibernate to consider that value. In this case we can ask hibernate to ignore values of those properties.

```
Example example = Example.create(exampleUser).excludeProperty("userName");
```

We can achieve like operator also.

```
exampleUser.setUserName("User 1%");
```

```
Example example = Example.create(exampleUser).enableLike();
```

---

## Hibernate Tutorial 32 - Caching in Hibernate

**Hibernate** automatically provides first level cache and this is implemented by session object. When you update an object that is already in persistent state and you update it again, it does not necessarily result into 2 update queries. Hibernate automatically detects what is the minimum no of queries that are needed to run and based on that it triggers the update query. This is applicable even for select. If we do select once and then there is no change and we do select (session.get()) again then hibernate knows that is already has the data in its cache , it does not trigger a select again.

Q) Why do we need second level of cache?

Ans: The session is not something that you hold for a long period of time. The SessionFactory is there for the entire life of the application but session is something that you want to open when you need to talk to the database and then close it as soon as you are done talking to the database. So if you are accessing a data that you have accessed in the earlier session then you have to query the DB again. Here is the need for another level of cache that goes beyond the session.

The Second level cache can be implemented in different ways:

(1) Across sessions in an application

This addresses the problem of cache loss after closing a session. You open sessions at different points in the application, all those sessions will have individual cache as first level cache but then there is a second level of cache that applies across different sessions so if it is not there in first level of cache, it is likely that you will find it in second level of cache.

(2) Across applications

You can have cache across different applications if they are all using hibernate and they are working on same set of data, you might feel to have a cache that goes across applications. It is like different sessions in different applications.

(3) Across Clusters

You have different applications deployed in the different servers and they are all talking to the same database, you can have a hibernate cache that supplies cache data across all these applications

# *Hibernate Cache*

- First Level Cache – Session
- Second level cache
  - Across sessions in an application
  - Across applications
  - Across clusters

## (1) First Level Cache

Let us consider the below code:

### **HibernateTest.java**

```
package com.kunj.hibernate;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import com.kunj.personal.dto.UserDetails;

public class HibernateTest {

    public static void main(String[] args) {

        SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
        Session session = sessionFactory.openSession();
        session.beginTransaction();

        UserDetails user = (UserDetails)
session.get(UserDetails.class, 11);
```

```

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

```

Console o/p:

```

Hibernate: select userdetail0_.userID as userID0_0_, userdetail0_.userName as
userName0_0_ from USER_DETAILS userdetail0_ where userdetail0_.userID=?

```

Now, what if we add another session.get()?

```

UserDetails user = (UserDetails) session.get(UserDetails.class,
11);
UserDetails user1 = (UserDetails) session.get(UserDetails.class,
11);

```

Hibernate does run a query for the second session.get() as it already has the same data in its first level cache. Hibernate knows that between these 2 lines there is no code that updates the User data.

Console o/p;

```

Hibernate: select userdetail0_.userID as userID0_0_, userdetail0_.userName as
userName0_0_ from USER_DETAILS userdetail0_ where userdetail0_.userID=?

```

What if we update the user?

```

UserDetails user = (UserDetails) session.get(UserDetails.class,
11);
user.setUserName("Updated User");
UserDetails user1 = (UserDetails) session.get(UserDetails.class,
11);

```

Console o/p:

```

Hibernate: select userdetail0_.userID as userID0_0_, userdetail0_.userName as
userName0_0_ from USER_DETAILS userdetail0_ where userdetail0_.userID=?
Hibernate: update USER_DETAILS set userName=? where userID=?

```

(Q) Why does hibernate not issue 2 queries in the above case?

Ans: Since the object is updated in the session, this object is already there in the cache. So there is no need for hibernate to pull up the value from the DB. The user1 is actually the updated user

object. Hibernate intelligently detects all these things. It does not go to the database unless it is required.

(Q) What if we try to access the object in a different session.

#### **HibernateTest.java**

```
package com.kunj.hibernate;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import com.kunj.personal.dto.UserDetails;

public class HibernateTest {

    public static void main(String[] args) {

        SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
        Session session = sessionFactory.openSession();
        session.beginTransaction();

        UserDetails user = (UserDetails)
session.get(UserDetails.class, 11);

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

        Session session2 = sessionFactory.openSession();
        session2.beginTransaction();

        UserDetails user1 = (UserDetails)
session2.get(UserDetails.class, 11);

        session2.getTransaction().commit();
        session2.close();

    }
}
```

Console o/p:

```
Hibernate: select userdetail0_.userID as userID0_0_, userdetail0_.userName as
userName0_0_ from USER_DETAILS userdetail0_ where userdetail0_.userID=?
Hibernate: select userdetail0_.userID as userID0_0_, userdetail0_.userName as
userName0_0_ from USER_DETAILS userdetail0_ where userdetail0_.userID=?
```

Here hibernate is issuing 2 select queries. Since the session is closed, the cache is close as well. Once we implement second level cache, both the sessions are going to talk to second level cache.

---

## Hibernate Tutorial 33 - Configuring Second Level Cache

### Hibernate.cfg.xml

```
<!-- Enable the second-level cache -->
<property name="cache.use_second_level_cache">true</property>
<!-- Provide the class name of the implementor of 2nd level cache -->
<property name="cache.provider_class"> org.hibernate.cache.EhCacheProvider
</property>
```

In order to use EH cache, we need to supply the jars. Go to Eh cache site and download the EHCACHE jar.

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-ehcache</artifactId>
  <version>3.6.2.Final</version>
</dependency>
```

Now we have the cache provider that hibernate has included inside it by default, hibernate will use this EHCACHE.jar to provide the caching services.

Now we have to configure the entities that we want to be cached. Annotate entity with @Cacheable.

Now mention the configuration for the cache. This is also called the caching strategy.

@Cache(usage = CacheConcurrencyStrategy.READ\_ONLY)

CacheConcurrencyStrategy.READ\_ONLY → assumes that your application is only going to read the data. It is not going to write anything to DB for this entity.

Console o/p:

```
Hibernate: select userdetail0_.userID as userID0_0_, userdetail0_.userName as
userName0_0_ from USER_DETAILS userdetail0_ where userdetail0_.userID=?
```

Only one select query fired this time.

---

## Hibernate Tutorial 34 - Using Query Cache

Instead of using session.get(), let us say we are pulling the data using a query .

### HibernateTest.java



```

package com.kunj.hibernate;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateTest {

    public static void main(String[] args) {

        SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();

        Session session = sessionFactory.openSession();
        session.beginTransaction();
        Query query = session.createQuery("from UserDetails
user where user.userID=11");
        List users = query.list();
        session.getTransaction().commit();
        session.close();

        Session session2 = sessionFactory.openSession();
        session2.beginTransaction();
        Query query1 = session2.createQuery("from UserDetails
user where user.userID=11");
        users = query1.list();
        session2.getTransaction().commit();
        session2.close();
    }
}

```

Console o/p:

```

Hibernate: select userdetail0_.userID as userID0_, userdetail0_.userName as
userName0_ from USER_DETAILS userdetail0_ where userdetail0_.userID=11
Hibernate: select userdetail0_.userID as userID0_, userdetail0_.userName as
userName0_ from USER_DETAILS userdetail0_ where userdetail0_.userID=11

```

It is generating 2 select queries despite the Eh cache configuration. Why?

Hibernate differentiates queries as something else, we need to specify a query cache separately.

```
<property name="cache.use_query_cache">true</property>
```

second\_level\_cache is different from query\_cache. The result of the query is not stored in the same place in the cache as it would store the result of a session.get().

3 caches in hibernate:

- 1) Session cache
- 2) Second level cache
- 3) Query cache

Now make the query cacheable.

#### **HibernateTest.java**

```
package com.kunj.hibernate;

import java.util.List;

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

public class HibernateTest {

    public static void main(String[] args) {

        SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();

        Session session = sessionFactory.openSession();
        session.beginTransaction();
        Query query = session.createQuery("from UserDetails
user where user.userID=11");
        query.setCacheable(true);
        List users = query.list();
        session.getTransaction().commit();
        session.close();

        Session session2 = sessionFactory.openSession();
        session2.beginTransaction();
        Query query1 = session2.createQuery("from UserDetails
user where user.userID=11");
        /*
        * If query1.setCacheable(true) is not here, hibernate
will result in 2 select
        * queries setCacheable(true) not only caches the query
result, it also tells the
```

```

        * query to look at second level cache and see if it
is already there.
        */
        query1.setCacheable(true);
        users = query1.list();
        session2.getTransaction().commit();
        session2.close();
    }
}

```

**Console o/p:**

```

Hibernate: select userdetail0_.userID as userID0_, userdetail0_.userName as
userName0_ from USER_DETAILS userdetail0_ where userdetail0_.userID=11

```

**Do not use query cache for the data the is frequently updated.**