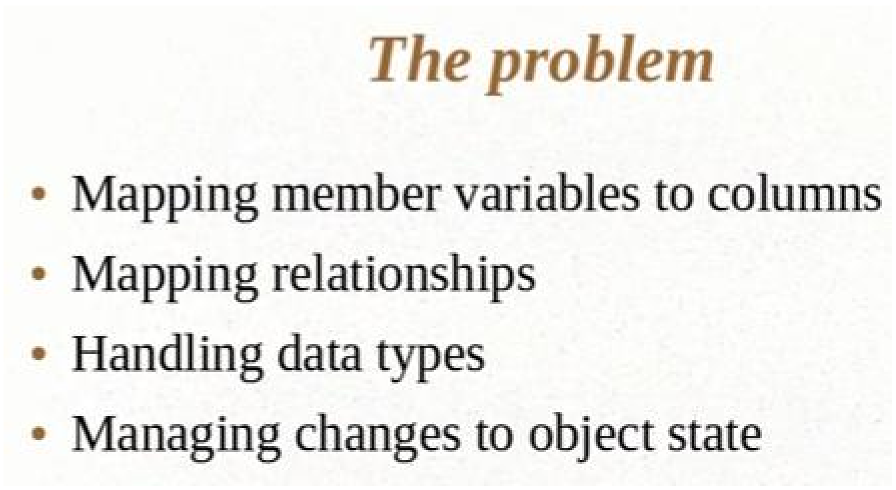
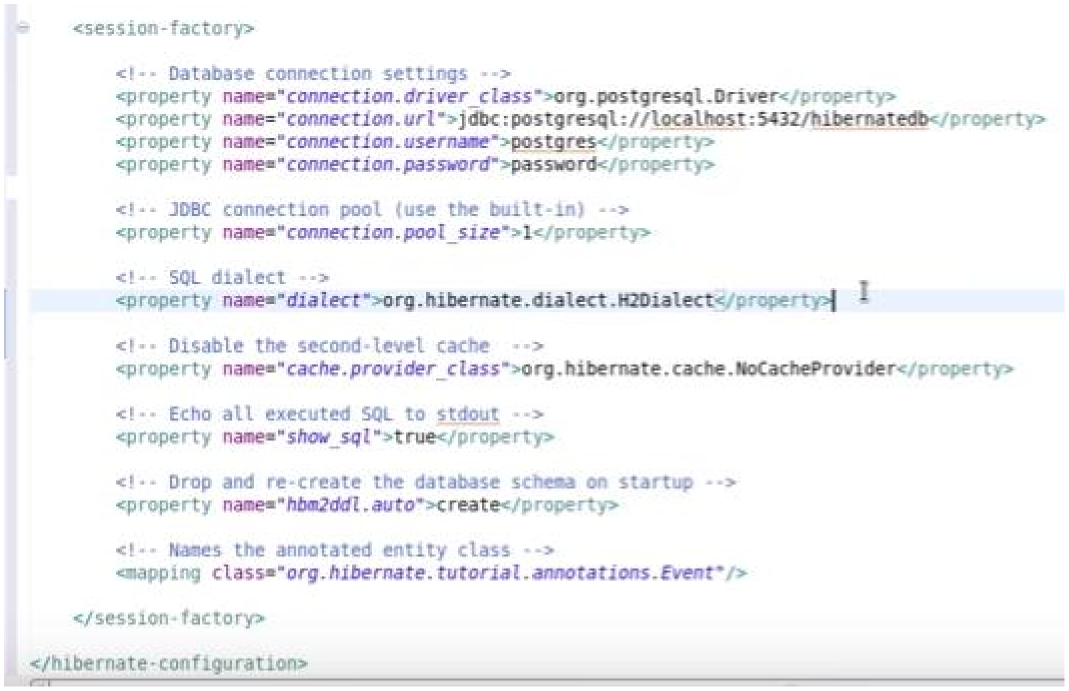


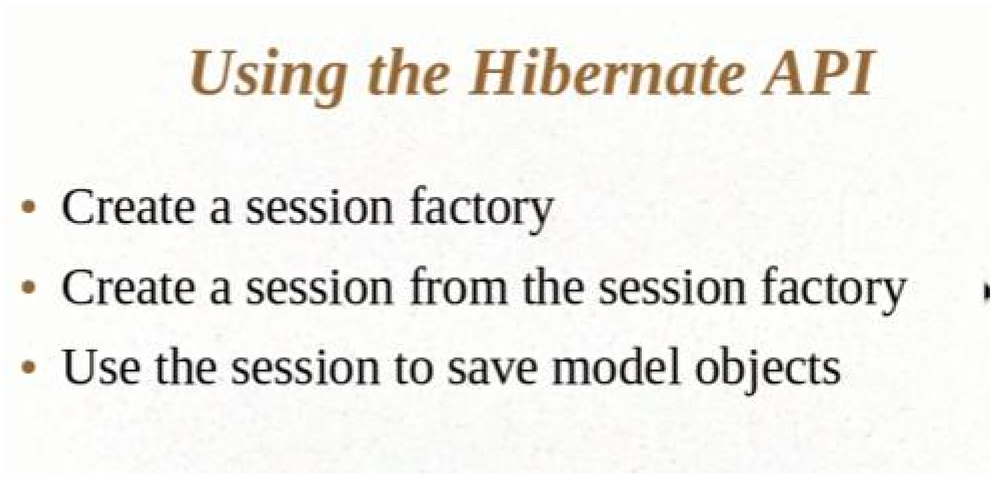
The problem

- Mapping member variables to columns
- Mapping relationships
- Handling data types
- Managing changes to object state

hibernate.cfg.xml



```
    <session-factory>

        <!-- Database connection settings -->
        <property name="connection.driver_class">org.postgresql.Driver</property>
        <property name="connection.url">jdbc:postgresql://localhost:5432/hibernatedb</property>
        <property name="connection.username">postgres</property>
        <property name="connection.password">password</property>

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

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.H2Dialect</property>

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

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

        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">create</property>

        <!-- Names the annotated entity class -->
        <mapping class="org.hibernate.tutorial.annotations.Event"/>

    </session-factory>

</hibernate-configuration>
```



Using the Hibernate API

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

Object to Relational database mapping.

Hibernate:

Meaning sleep mode.

Designed for enterprise applications.

Meaning for business applications.

Hibernate Features:

1. Auto DDL(create, alter)
2. HQL Support(Database independent query)
3. cache support.
4. Primary key generator support.

5.Validation support(in the form of annotation)

6. Exception handling support.(No need to handle any compiletime exceptions.)

7. ORM Support.(isA, hasA, one-to-one, one-to-many,many-to-many, many-to-one)(RDBMS)

8. OGM Support(object graph mapping)(For non relational databases.)(MongoDB or Hadoop hbase)

9. Hibernate search support.(index based search algorithems.)(apache lucine search implementations)

10. connection pool implementations.

Object Types:

1. Persistent
2. Detached(evict)
3. Transient

Things Needed to configure hibernate:

1. A Pojo class
2. Hibernate mapping file.(hibernate.cofig.xml) (<hibernate-configuration>)

3. Hibernate configuration file.(<hibernate-mapping>)

Hbmtodllauto

DDL operations:

- Create
- Alter
- Drop
- Truncate
- Rename

Hbm2ddl.auto

1. Create(drop and create)
    - Drop existing tables.
    - Create fresh new tables.
2. Update(alter and create if needed(not exists))
    - Wouldn’t drop existing tables.
    - If any table required, it will create.
    - If any table requires, it will do alter operations.

- If table already contains data then new not null columns not alterable.

3. Validate

   - Checks mapping schema against table schema.

4. Create-drop

   - Drop table if exists.
   - Create new table.
   - Drop existing table.(when we call sessionFactory.close)

Curd operations:

Insert record. (save, persist, saveorupdate)

- Save(), method return type is primary key.(can execute without transaction boundary)
- Persist(), will return void.(can execute with in transaction boundary)
- Saveorupdate(), returns void

Update record.(update and merge)

- If any case update fails execute merge.

Delete Record.

- Delete

Select one row(get method):

- Object o = session.get(Student.class,111)
- When calling get method itself it will fire query and fetch record from DB
- For these select operation transactions are not required.
- Only for insert, update and delete operations we need transactions.
- If we pass some id(PK) which is not present in DB, get method will return null to us.

Load() Method:

- After getting object, when you typecast and call non primary key getter methods, then only it will fire select query.
- When we pass id(PK), not present in DB, call on non

primary key getter methods will give no object found exception.

- Get() is eager select method and Load is lazy.

Primary Key Auto generators:

1. Assigned(default)
2. Increment(select max id from db and do id++)
3. Sequence(both db and application layer responsible.)
4. Identity (DB is responsible for increment.)
5. Native
6. Hilo
7. Foreign
8. Custom generators

HQL (Hibernate query language)

1. Insert (only possible to insert one table data to another table.)
2. Update
3. Delete

4. Select

HQL queries are object oriented queries.

Database independent.

- Session.createQuery(Hql);
- Return type is query()(Depricated).
- Query has multiple methods like, executeupdate(),uniqueResult()(Deprecated) and List()(Depricated), method.
- `getResultList(), in hibernate 5.2.2`
- Executeupdate(), is for DML operations(insert, update delete).
- It will return how many rows affected by your query.

Criterias:

- We can make only select operations using criterias.
-

HQL Select operations:

1. One row select operation
2. Restrictions and projections.
3. Restrictions are for conditions.(=,>,<,between,like etc.)
4. Projections are for aggregate function and particular column selection.
5. Projections and restrictions we have to add to the criteria's.

ORM Relations:

1. Inheritance(IS-A) support
    - Table-per-class
    - Table-per-subclass
    - Table-per-concreteClass
2. Association(HAS-A)

- One-to-many
- Many-to-one
- Many-to-many
- One-to-one

3. Discriminator columns are special columns in DB Tables, for which we do not need to maintain a property in bean class.

One-to-many:

- Cascade=”all”, save parent object only, child objects automatically saved.
- Whenever we delete parent, corresponding child object also deleted.

One –to-one:

- Generator type foreign, is used to make one primary key as a primary key in other table.

@Annotations and XML mapping:

- Through configuration class we need to xml mapping files.

- In hibernate.cfg.xml, we have <mapping resource=" hbm.xml"/>
- In case of annotation based mapping in hibernate.cfg.xml, we have<mapping class="pojo"/>
- To process xml based mappings,

  Configuration cfg = new Configuration();

  Cfg.configure("hibernate.cfg.xml");
- To process annotation based mappings,

  Configuration cfg = new AnnotationConfiguration();

  Cfg.configure("hibernate.cfg.xml");
- For hibernate 4.x onwards, for xml and for annotation,

  Configuration cfg = new Configuration();

  Cfg.configure("hibernate.cfg.xml");

Hibernate validations:

- Jsr303 validation
- Use validator factory

Pagination:

- We can achieve it by using HQL and criteria.
- Using HQL

  Session s = sf.opensession();

  Query q = s.createQuery("From Students");

  q.setFirstResult(1);

  q.setMaxResult(5);

- Using Criteria

  Criteria cr = s.createCriteria(Student.class);

  cr.setFirstResult(1);

  cr.setMaxResult(5);

Cache Support:

In hibernate we have 3 types of cache support.

1. Session level(1st level)(applicable for 1 user)
2. SessionFactory level(2nd level)(application for all user)

3. Query level cache.(applicable for one instance)

- The main aim of cache is to reduce the number of database calls, to improve application performance.
- Session level cache is default cache.

Query cache:

```
Query q = session.createQuery("select…");

q.setCacheable(true);
```

=====================================

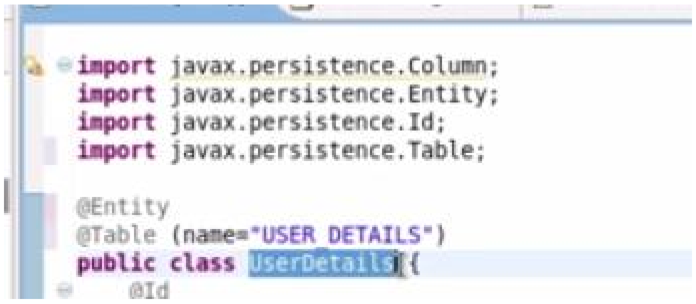
```
@OneToMany(mappedBy="goal",
cascade=CascadeType.ALL)

private List<Exercise> exer = new ArryList<>();
```

To Fix lazy initialization errors:

- we use entity manager in case of JPA to save and retrive the data.
- Need to implement this filter called OpenEntityManagerInViewFilter

Projection:

=======================



```java
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table (name="USER DETAILS")
public class UserDetails {
    @Id
```

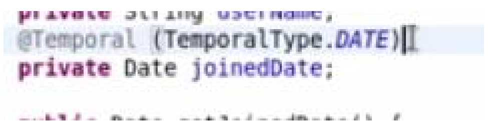
The difference between giving name like @Entity(name="USER_DETAILS") and above is first approach renames the entity class itself.

But latter approach only uses different name for the table.

You can encounter this while writing the HQL queries.

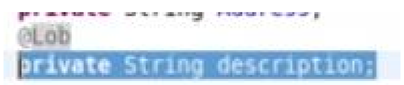
@Basic Annotation tells apply hibernate basics and cretate the column. it is by default applied. Use it to set some other properties if you want.

If we don't want to save some fields we can mark them **transient or static.**



```java
@Temporal (TemporalType.DATE)
private Date joinedDate;
```

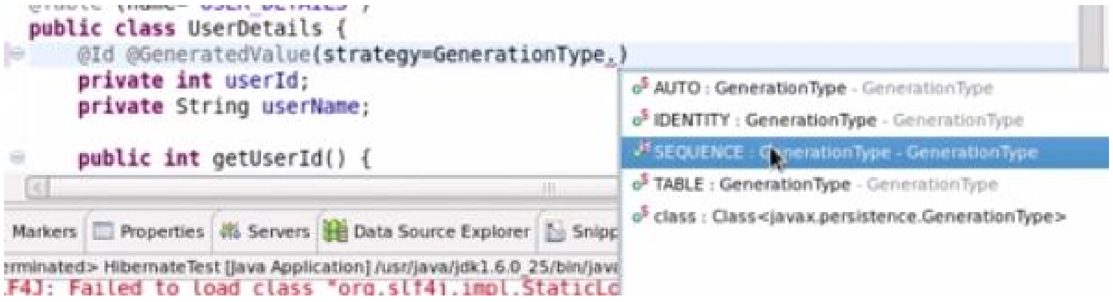
To save only date not date and time.



```java
@Lob
private String description;
```

instruct hibernate to select either CLOB(Character large Object) or BLOB(Byte stream large object) .

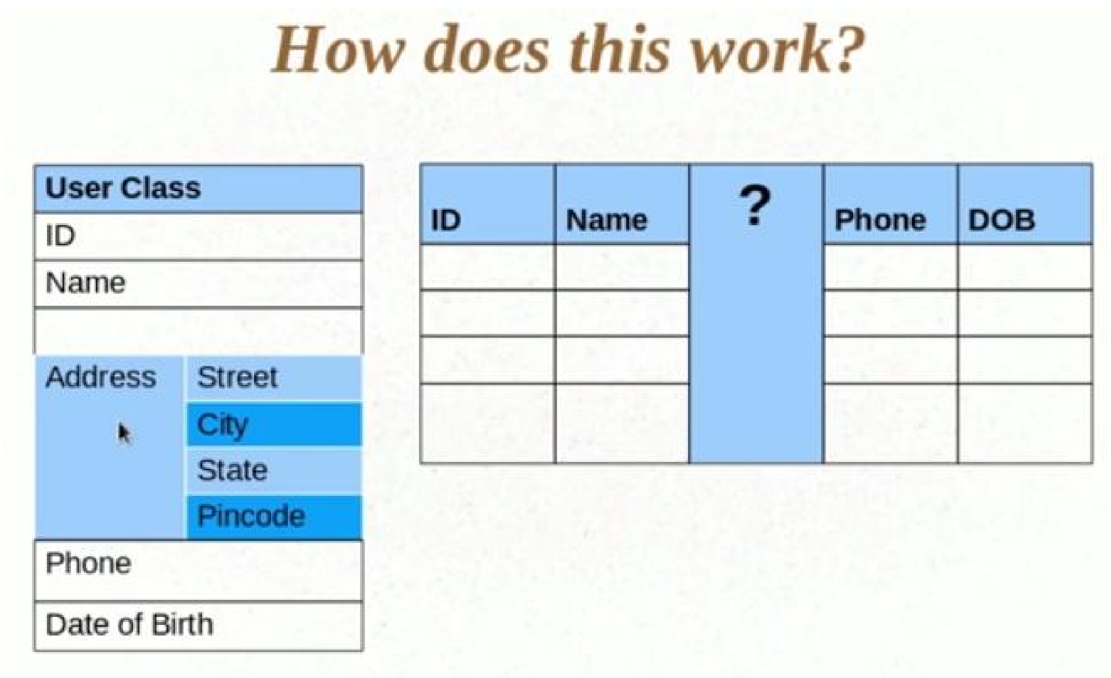
Natural Primery Key(e.g. emailId) and sarrogate key(developer provided Id, does not have business use)



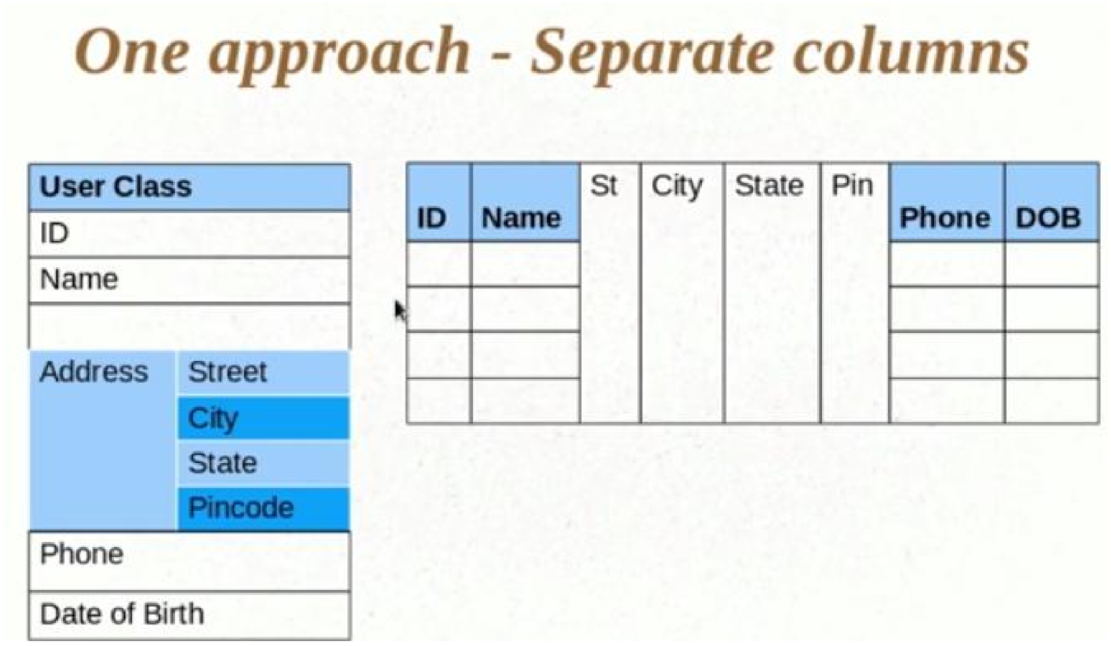
```
public class UserDetails {
    @Id @GeneratedValue(strategy=GenerationType.)
    private int userId;
    private String userName;

    public int getUserId() {
```

```
rminated> HibernateTest [Java Application] /usr/java/jdk1.6.0_25/bin/jav
F4J: Failed to load class "org.slf4j.impl.StaticLo
```

AUTO : GenerationType - GenerationType
IDENTITY : GenerationType - GenerationType
SEQUENCE : GenerationType - GenerationType
TABLE : GenerationType - GenerationType
class : Class<javax.persistence.GenerationType>
Markers
Properties
Servers
Data Source Explorer
Snipp

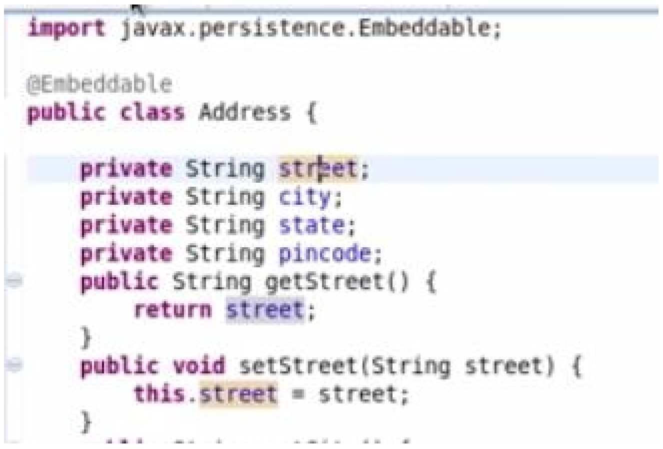


How does this work?

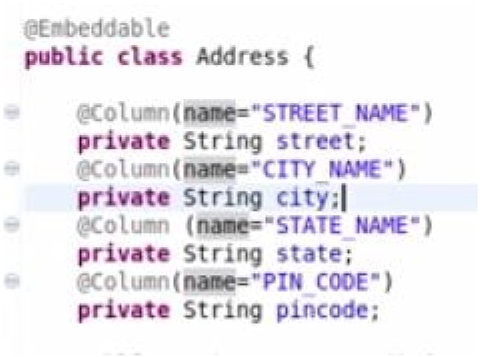


One approach - Separate columns

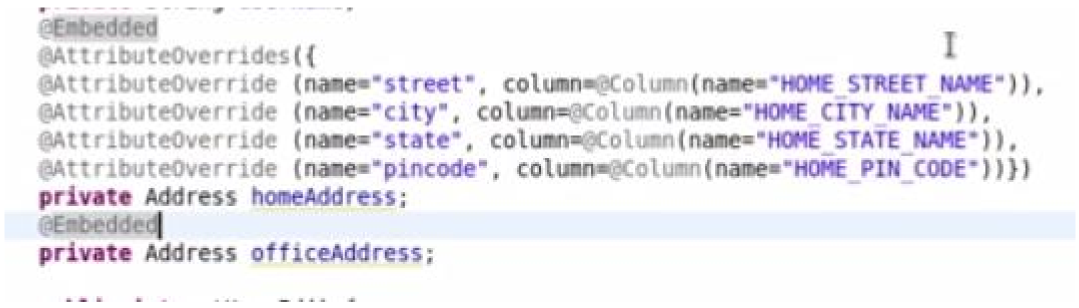
To treat address as value type @Embeddable



```
import javax.persistence.Embeddable;

@Embeddable
public class Address {

    private String street;
    private String city;
    private String state;
    private String pincode;
    public String getStreet() {
        return street;
    }
    public void setStreet(String street) {
        this.street = street;
    }
```



```
@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;
```

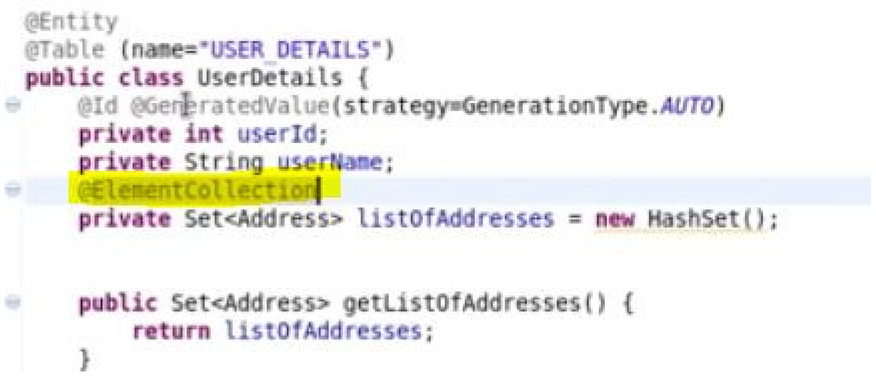


```
@Embedded
@AttributeOverrides({
@AttributeOverride (name="street", column=@Column(name="HOME_STREET_NAME")),
@AttributeOverride (name="city", column=@Column(name="HOME_CITY_NAME")),
@AttributeOverride (name="state", column=@Column(name="HOME_STATE_NAME")),
@AttributeOverride (name="pincode", column=@Column(name="HOME_PIN_CODE"))})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

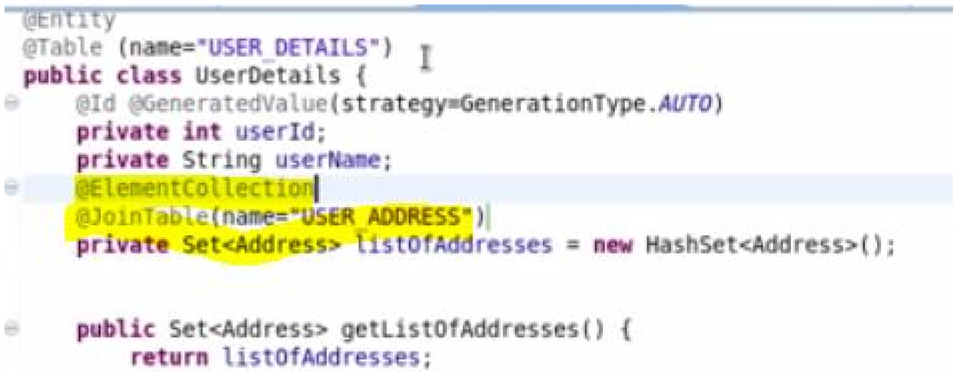
Composite primary Key:

@EmbededId

Saving Collections:



```
@Entity
@Table (name="USER_DETAILS")
public class UserDetails {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int userId;
    private String userName;
    @ElementCollection
    private Set<Address> listOfAddresses = new HashSet();


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



```
@Entity
@Table (name="USER_DETAILS")
public class UserDetails {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int userId;
    private String userName;
    @ElementCollection
    @JoinTable(name="USER_ADDRESS")
    private Set<Address> listOfAddresses = new HashSet<Address>();


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

Another e.g.

EMPLOYEE (table)

EMP_ID	F_NAME	L_NAME	SALARY
1	Bob	Way	50000
2	Joe	Smith	35000

PHONE (table)

OWNER_ID	TYPE	AREA_CODE	P_NUMBER
1	home	613	792-0001
1	work	613	494-1234
2	work	416	892-0005

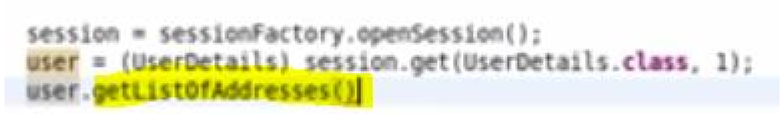
Example of an ElementCollection relationship annotations [edit]

```
@Entity
public class Employee {
  @Id
  @Column(name="EMP_ID")
  private long id;
  ...
  @ElementCollection
  @CollectionTable(
        name="PHONE",
        joinColumns=@JoinColumn(name="OWNER_ID")
  )
  private List<Phone> phones;
  ...
}
```

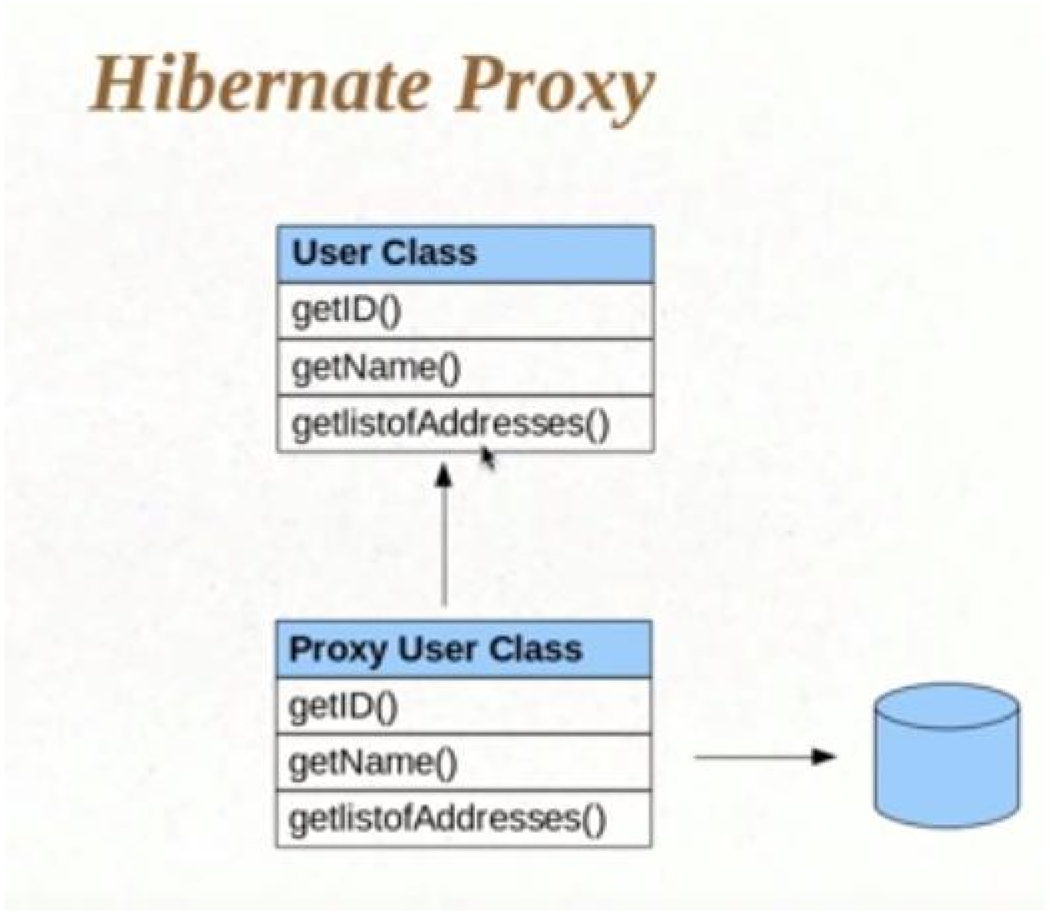
```
@Embeddable
public class Phone {
  private String type;
  private String areaCode;
  @Column(name="P_NUMBER")
  private String number;
  ...
}
```

Proxy Objects and Eager and Lazy Fetch types:

Lazy initialization,

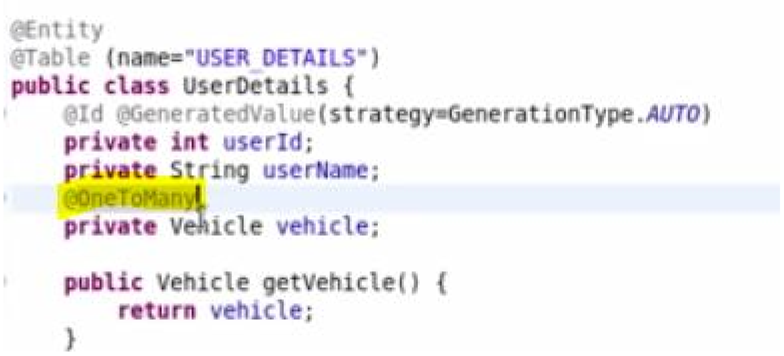


```
session = sessionFactory.openSession();
user = (UserDetails) session.get(UserDetails.class, 1);
user.getListOfAddresses()
```



One-to-One Mapping:

One-to-Many

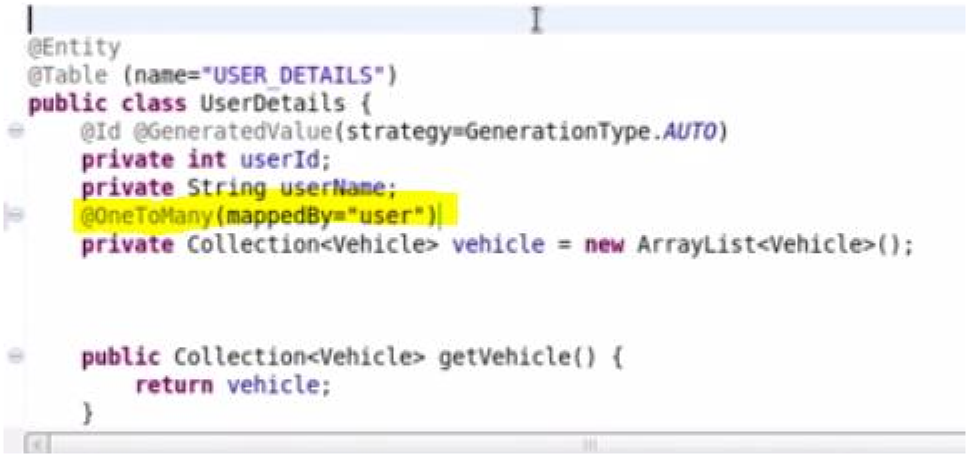


```
@Entity
@Table (name="USER_DETAILS")
public class UserDetails {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int userId;
    private String userName;
    @OneToMany
    private Vehicle vehicle;

    public Vehicle getVehicle() {
        return vehicle;
    }
```

One user can have many vehicles. From vehicle side it would be @ManyToOne.

it creates a separate mapping table with userid and vehicleId in DB.

If we don't want to create a separate mapping table then,



```
@Entity
@Table (name="USER_DETAILS")
public class UserDetails {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int userId;
    private String userName;
    @OneToMany(mappedBy="user")
    private Collection<Vehicle> vehicle = new ArrayList<Vehicle>();


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

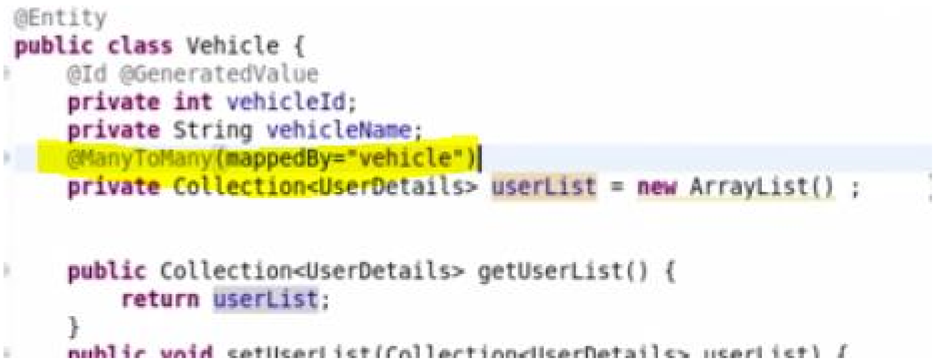
inside vehicle entity,

@ManyToOne

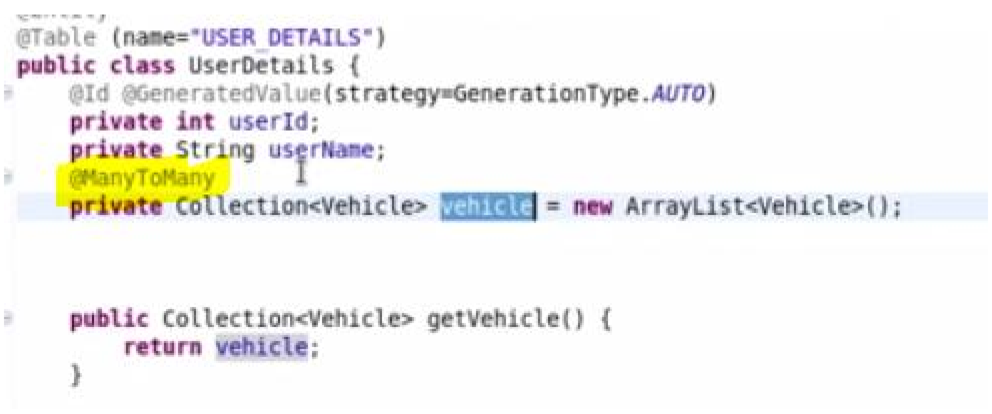
@joinColumn(name="USER_ID")

private UserDetails user;

Many-To-Many:(user and rented vehicle)

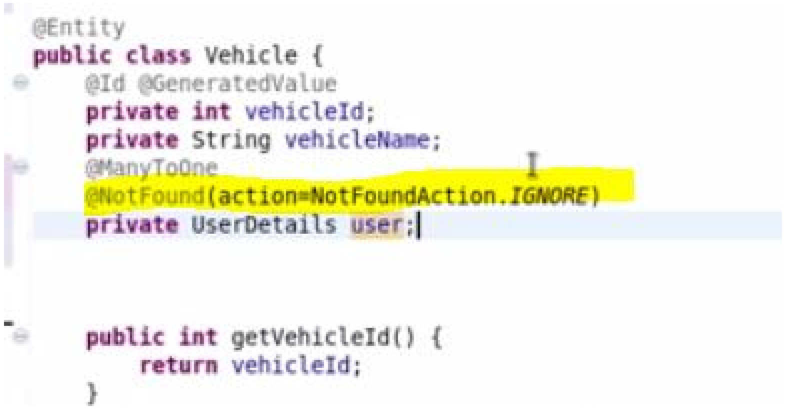


```java
@Entity
public class Vehicle {
    @Id @GeneratedValue
    private int vehicleId;
    private String vehicleName;
    @ManyToMany(mappedBy="vehicle")
    private Collection<UserDetails> userList = new ArrayList() ;


    public Collection<UserDetails> getUserList() {
        return userList;
    }
    public void setUserList(Collection<UserDetails> userList) {
```

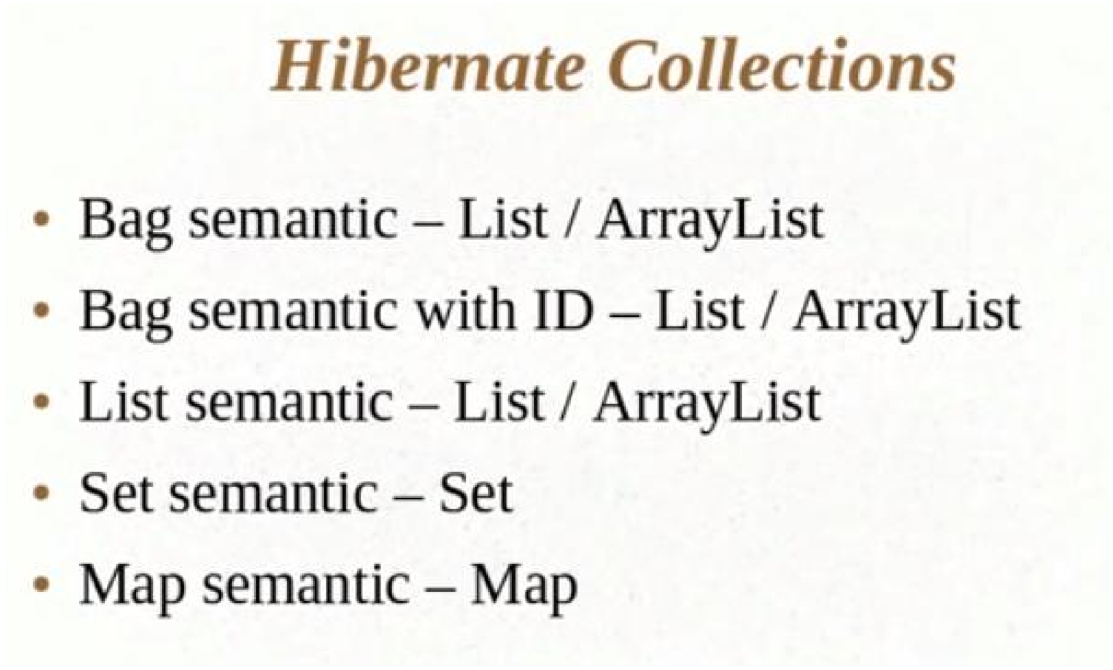


```java
@Table (name="USER_DETAILS")
public class UserDetails {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int userId;
    private String userName;
    @ManyToMany
    private Collection<Vehicle> vehicle = new ArrayList<Vehicle>();



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

Hibernate annotation @NotFound:



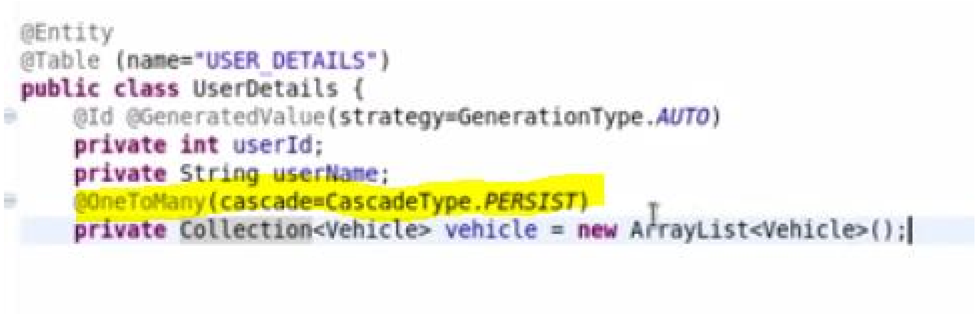
```java
@Entity
public class Vehicle {
    @Id @GeneratedValue
    private int vehicleId;
    private String vehicleName;
    @ManyToOne
    @NotFound(action=NotFoundAction.IGNORE)
    private UserDetails user;



    public int getVehicleId() {
        return vehicleId;
    }
```



Hibernate Collections

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

CascadeTypes:

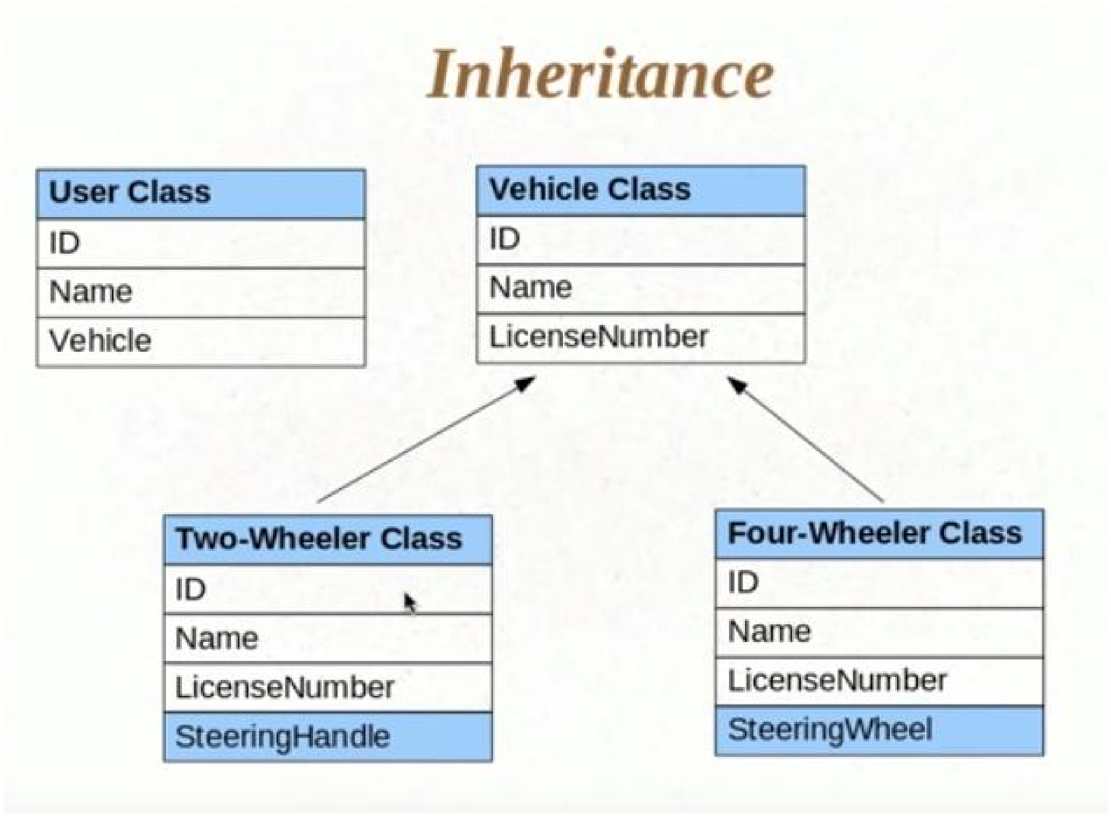


```
@Entity
@Table (name="USER_DETAILS")
public class UserDetails {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int userId;
    private String userName;
    @OneToMany(cascade=CascadeType.PERSIST)
    private Collection<Vehicle> vehicle = new ArrayList<Vehicle>();
```

Tells hibernate to save the Vehicle when we save the UserDetail Object.

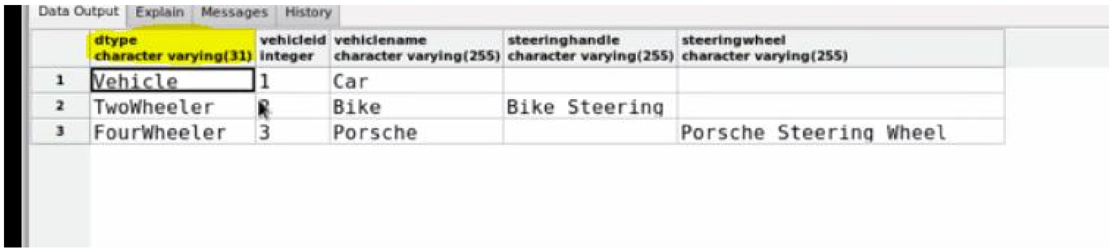
To apply all cascade types, cascade=CascadeType.ALL

Implementing Inheritance:



Single Table statergy by default provided by hibernate:

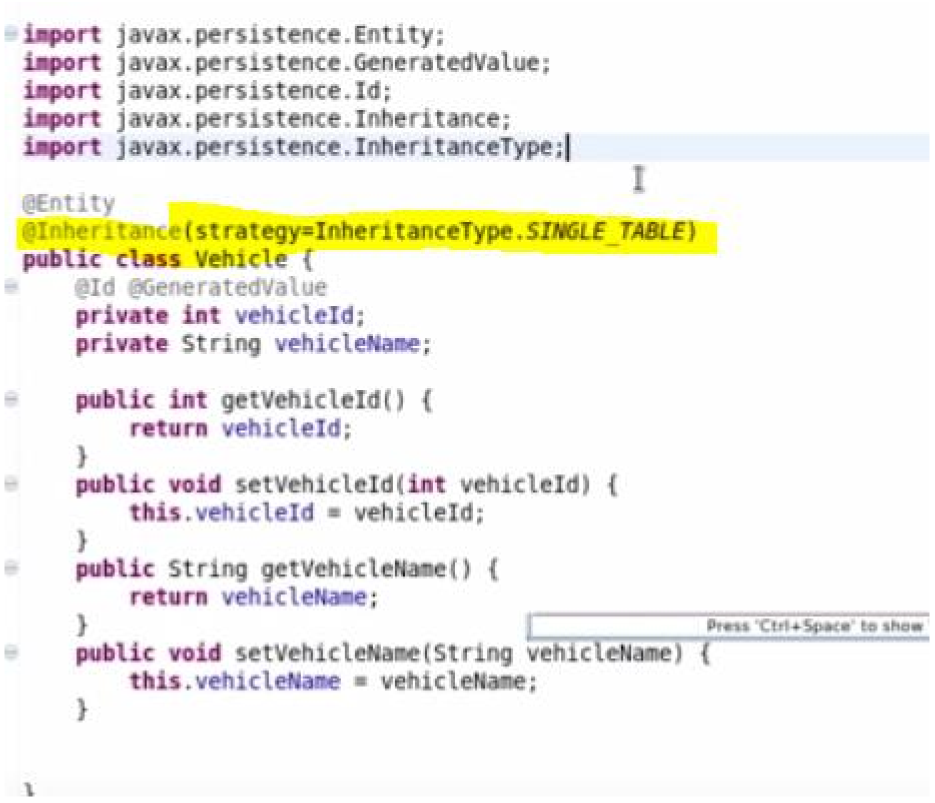
It is least normalized.



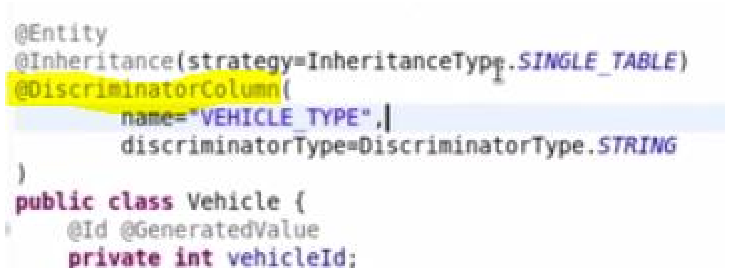
	dtype character varying(31)	vehicleid integer	vehiclename character varying(255)	steeringhandle character varying(255)	steeringwheel character varying(255)
1	Vehicle	1	Car		
2	TwoWheeler	2	Bike	Bike Steering	
3	FourWheeler	3	Porsche		Porsche Steering Wheel

dType is the discreminater type, to distinguish the data.

Optional Annotation

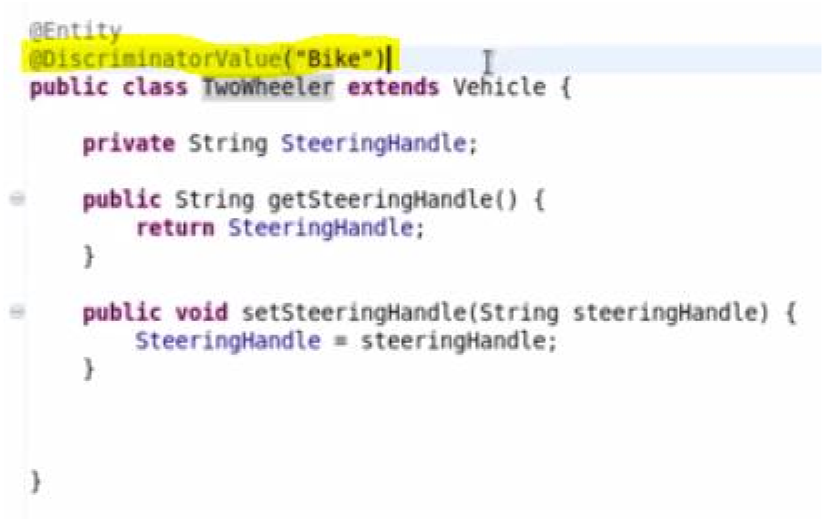


```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;

@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
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;
    }


}
```



```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
        name="VEHICLE_TYPE",
        discriminatorType=DiscriminatorType.STRING
)
public class Vehicle {
    @Id @GeneratedValue
    private int vehicleId;
```

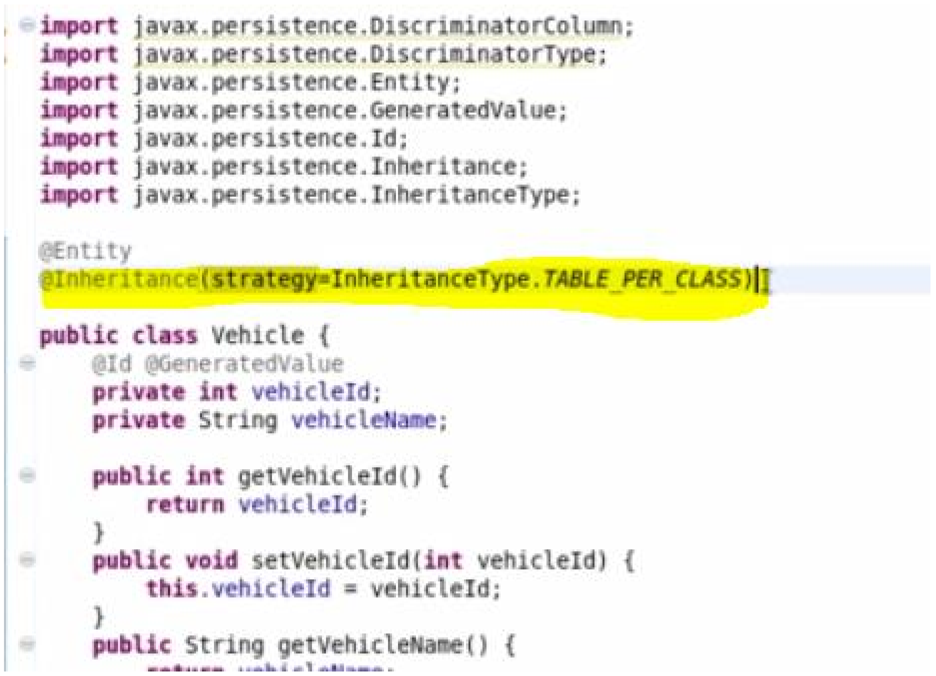
If we don't want to use class name as value,



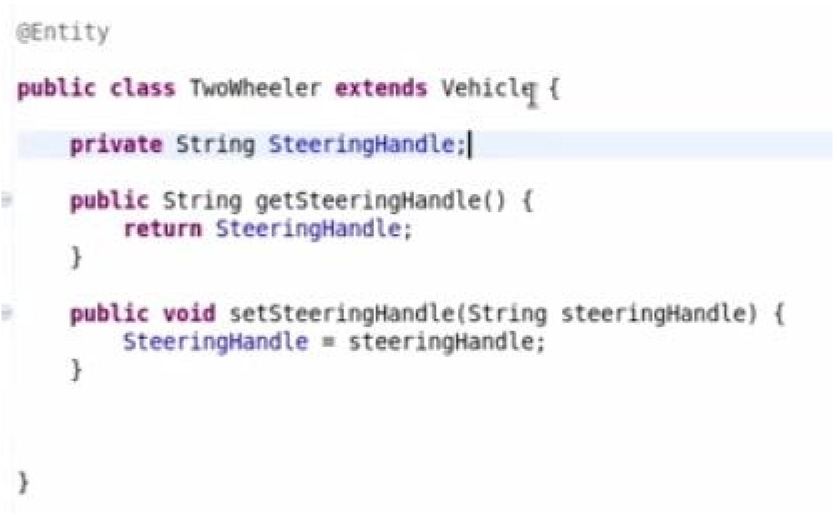
```
@Entity
@DiscriminatorValue("Bike")
public class TwoWheeler extends Vehicle {

    private String SteeringHandle;

    public String getSteeringHandle() {
        return SteeringHandle;
    }

    public void setSteeringHandle(String steeringHandle) {
        SteeringHandle = steeringHandle;
    }



}
```

Table per class Strategy:

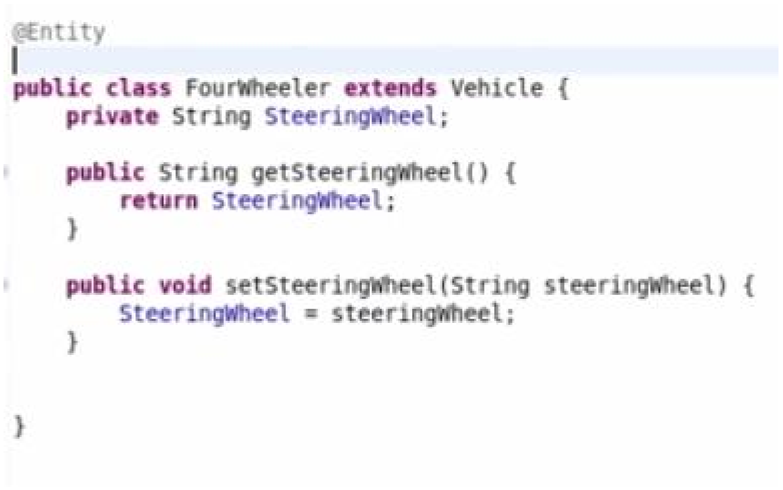
we don't need discriminator column here.



```
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.TABLE_PER_CLASS)

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() {
```

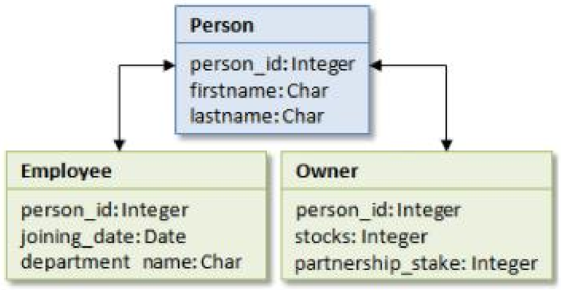


```
@Entity

public class TwoWheeler extends Vehicle {

    private String SteeringHandle;

    public String getSteeringHandle() {
        return SteeringHandle;
    }

    public void setSteeringHandle(String steeringHandle) {
        SteeringHandle = steeringHandle;
    }



}
```



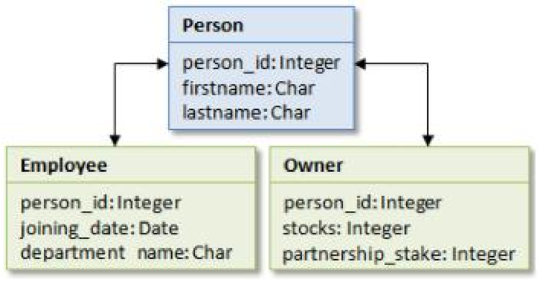
```
@Entity

public class FourWheeler extends Vehicle {
    private String SteeringWheel;

    public String getSteeringWheel() {
        return SteeringWheel;
    }

    public void setSteeringWheel(String steeringWheel) {
        SteeringWheel = steeringWheel;
    }


}
```

Another e.g.



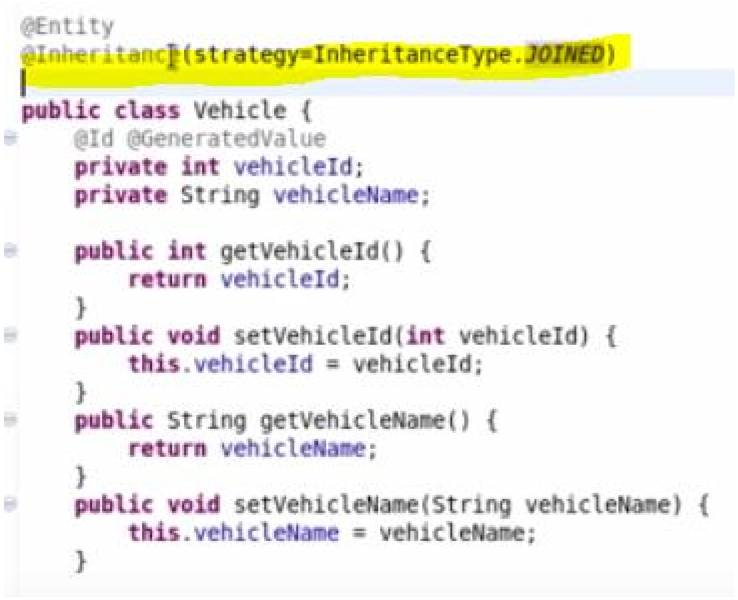
In One Table per Concrete class scheme, each concrete class is mapped as normal persistent class. Thus we have 3 tables; PERSON, EMPLOYEE and OWNER to persist the class data. In t scheme, the mapping of the subclass repeats the properties of the parent class.

Implementing Inheritance With Joined Strategy

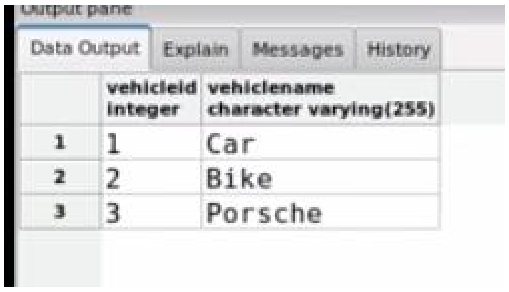


In One Table per Subclass scheme, each class persist the data in its own separate table. Thus we have 3 tables; PERSON, EMPLOYEE and OWNER to persist the class data. Note that a foreign key relationship exists between the subclass tables and super class table. Thus the common data is stored in PERSON table and subclass specific fields are stored in EMPLOYEE and OWNER tables.

another e.g.



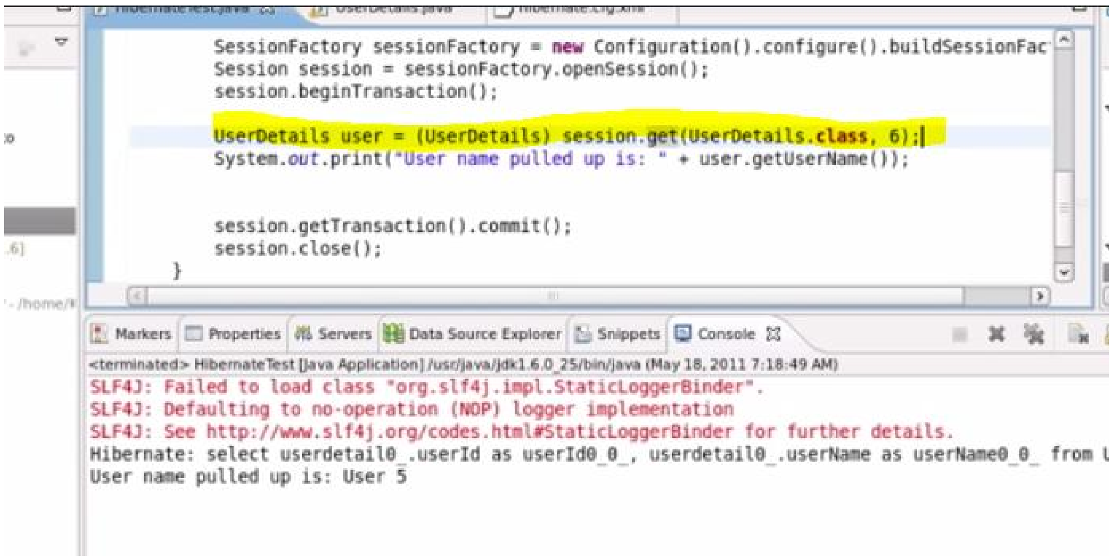
```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)

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;
    }
```



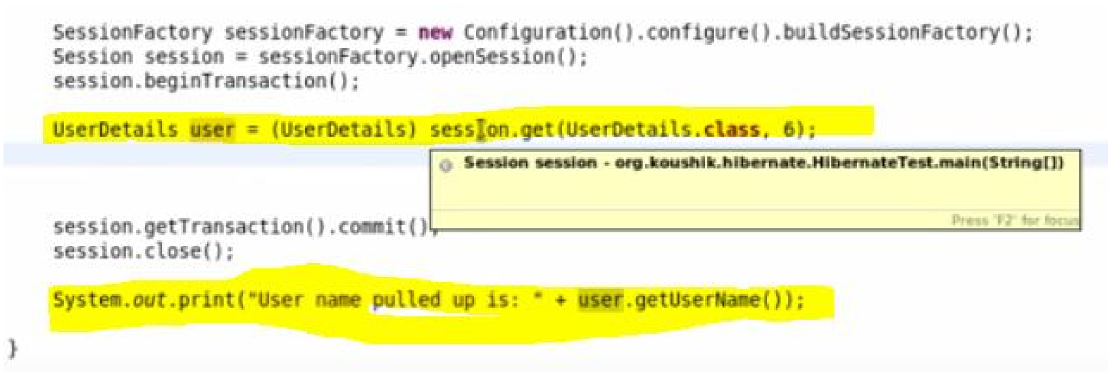
	vehicleid integer	vehiclename character varying(255)
1	1	Car
2	2	Bike
3	3	Porsche

CURD Operations:

Read:

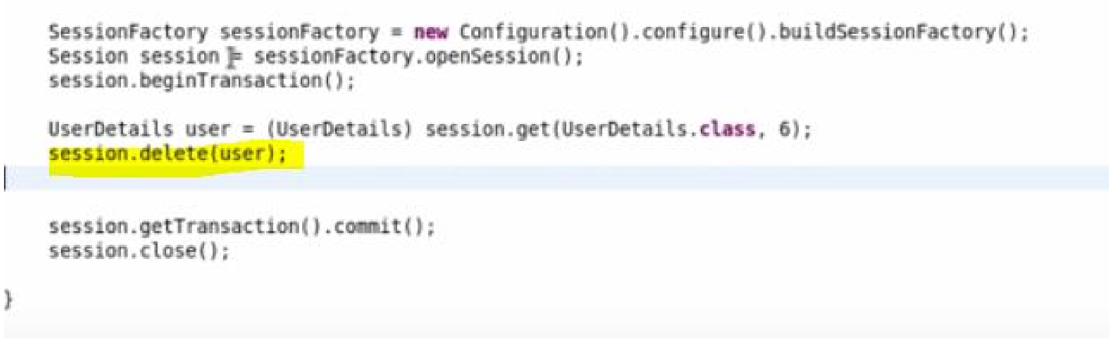

```
        SessionFactory sessionFactory = new Configuration().configure().buildSessionFac
        Session session = sessionFactory.openSession();
        session.beginTransaction();

        UserDetails user = (UserDetails) session.get(UserDetails.class, 6);
        System.out.print("User name pulled up is: " + user.getUserName());


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

```
<terminated> HibernateTest [Java Application] /usr/java/jdk1.6.0_25/bin/java (May 18, 2011 7:18:49 AM)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Hibernate: select userdetail0_.userId as userId0_0_, userdetail0_.userName as userName0_0_ from
User name pulled up is: User 5
```


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

    UserDetails user = (UserDetails) session.get(UserDetails.class, 6);


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

    System.out.print("User name pulled up is: " + user.getUserName());

}
```
Session session - org.koushik.hibernate.HibernateTest.main(String[])
Press 'F2' for focus

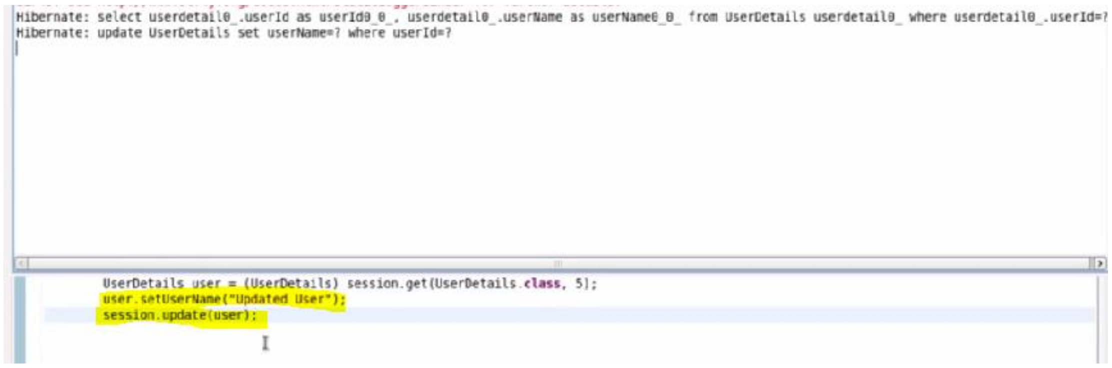
As soon as we do **session.get**, select query is fired(Eager Fetch) and proxy object is fetched, so we can call getter methods after closing the session as well.

Delete:



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

    UserDetails user = (UserDetails) session.get(UserDetails.class, 6);
    session.delete(user);


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

}
```

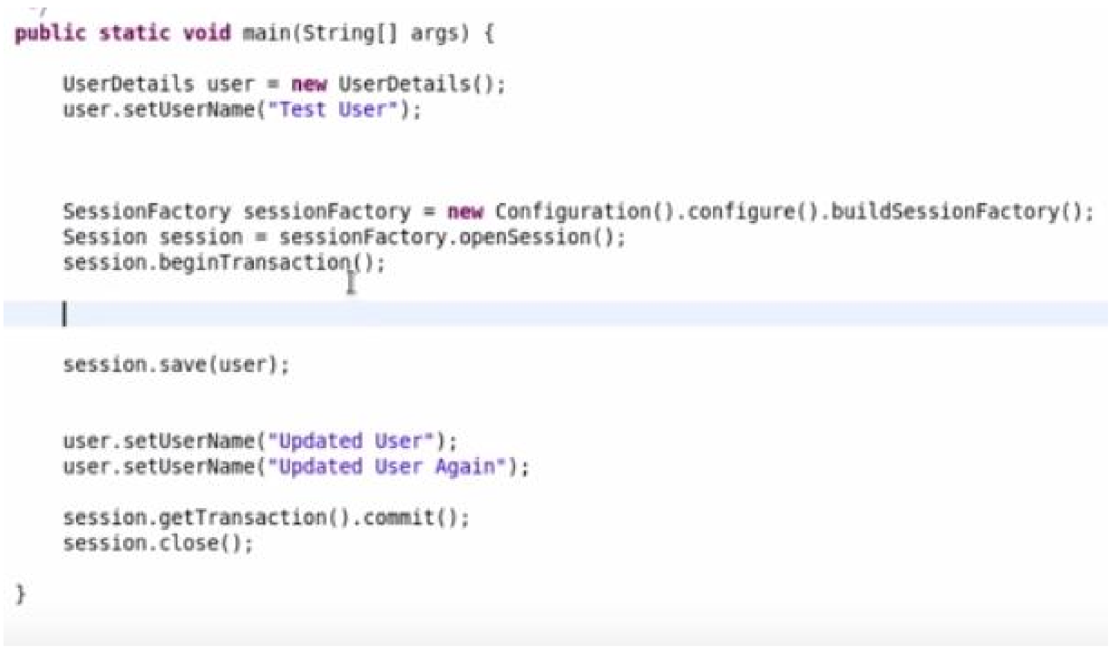
Update:



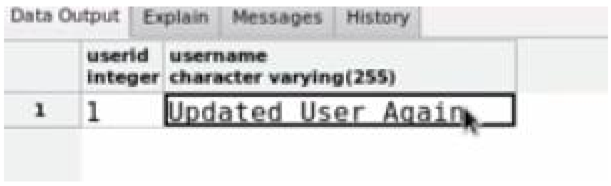
```
Hibernate: select userdetail0_.userId as userId0_0_, userdetail0_.userName as userName0_0_ from UserDetails userdetail0_ where userdetail0_.userId=?
Hibernate: update UserDetails set userName=? where userId=?
```

```
        UserDetails user = (UserDetails) session.get(UserDetails.class, 5);
        user.setUserName("Updated User");
        session.update(user);
```

Transient, Persistent and Detached Objects:



```
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.setUserName("Updated User");
    user.setUserName("Updated User Again");

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

}
```

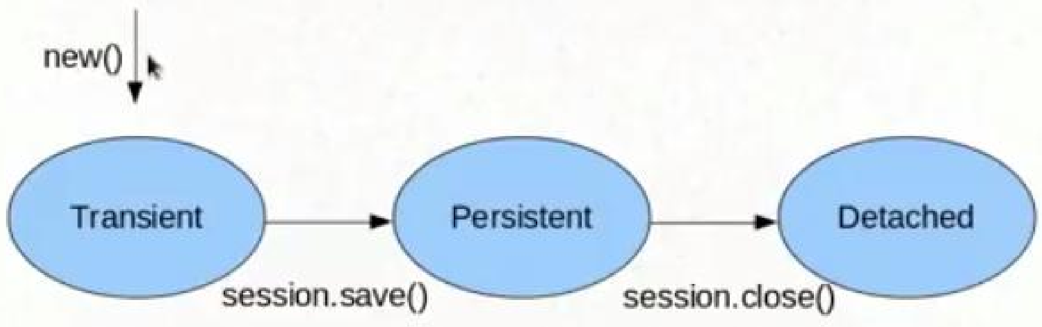


	userid integer	username character varying(255)
1	1	Updated User Again

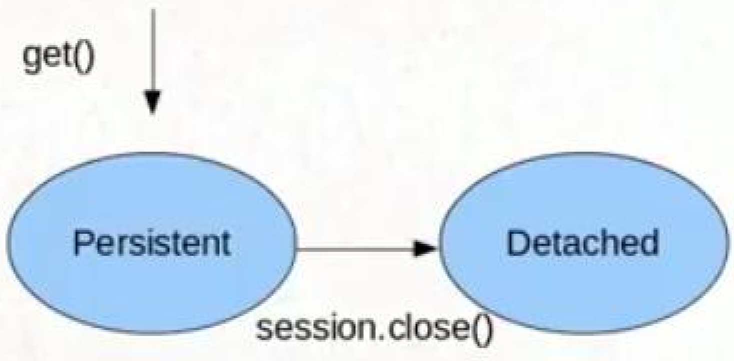
At the beginning the user object is in Transient state, after session.save(user), it will be in persistance state. After we do session.close(), it will be in detached state.

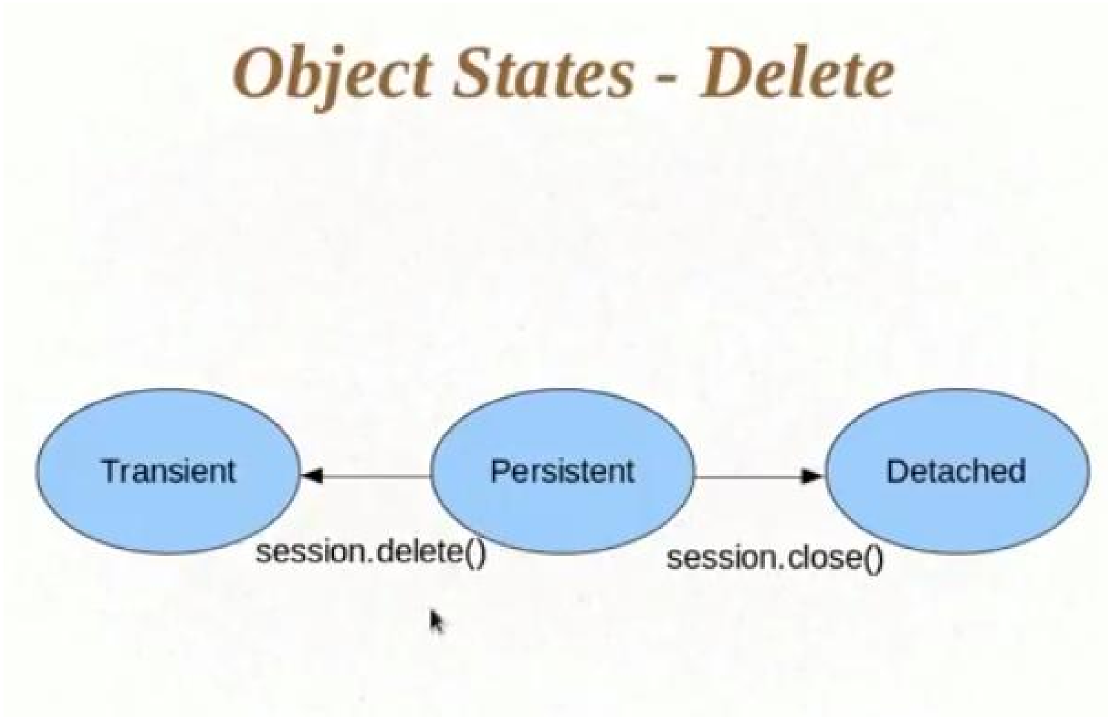
Understanding State Changes:

Object States - Create

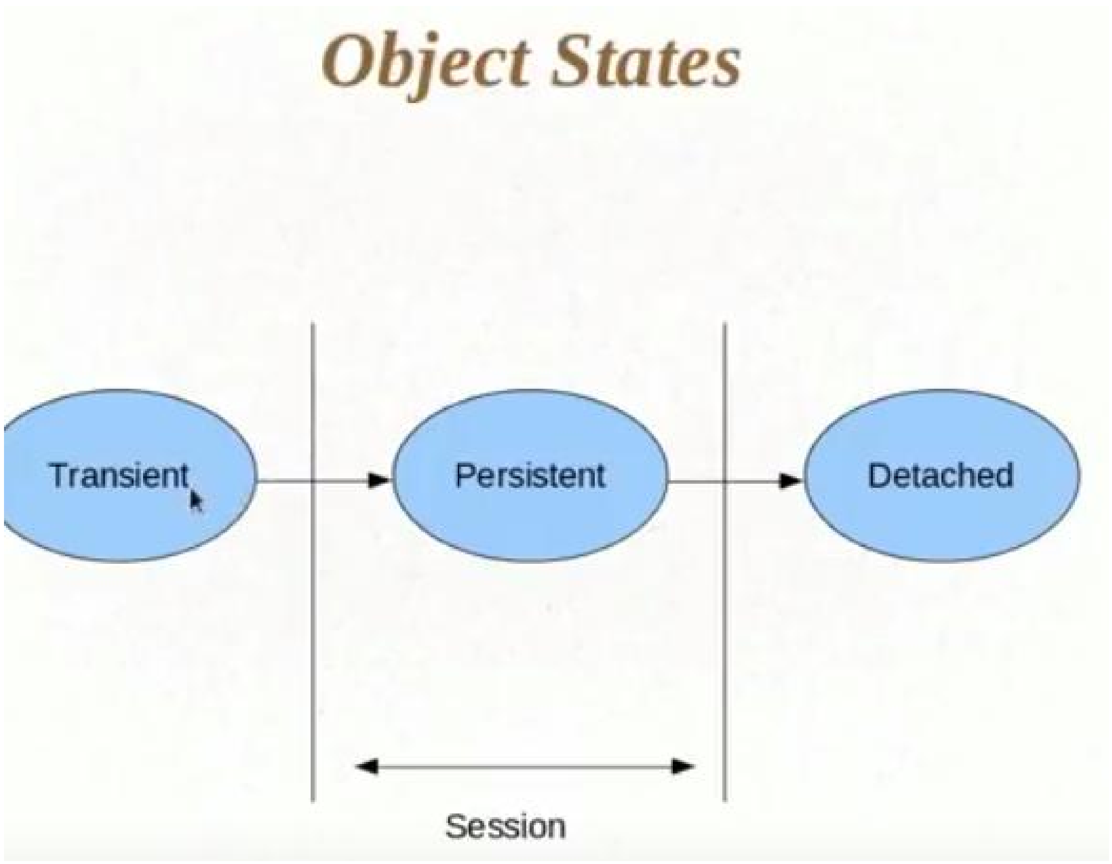


Object States - Read



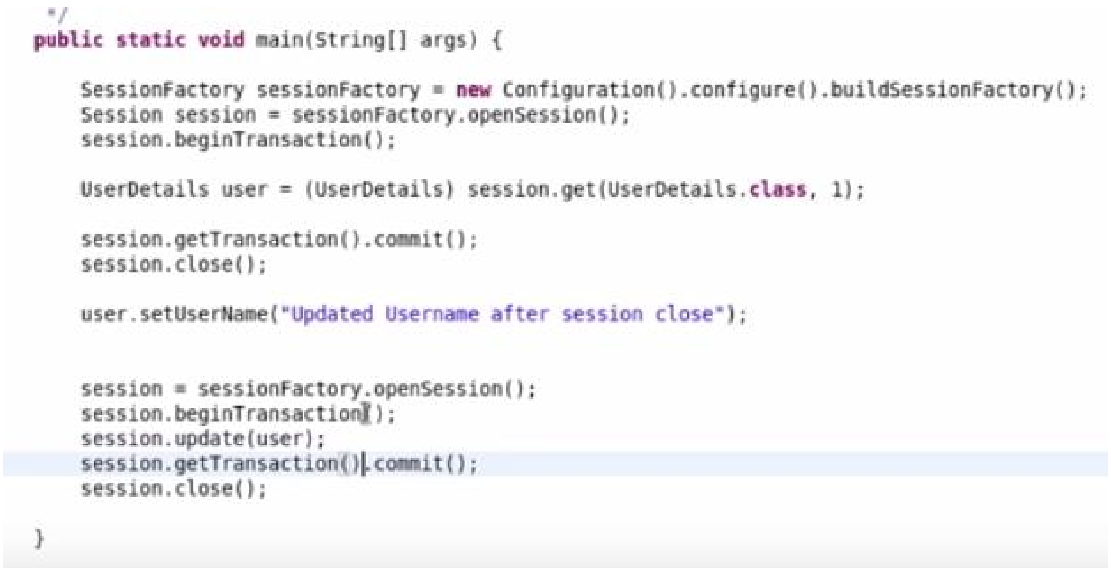


Object States - Delete

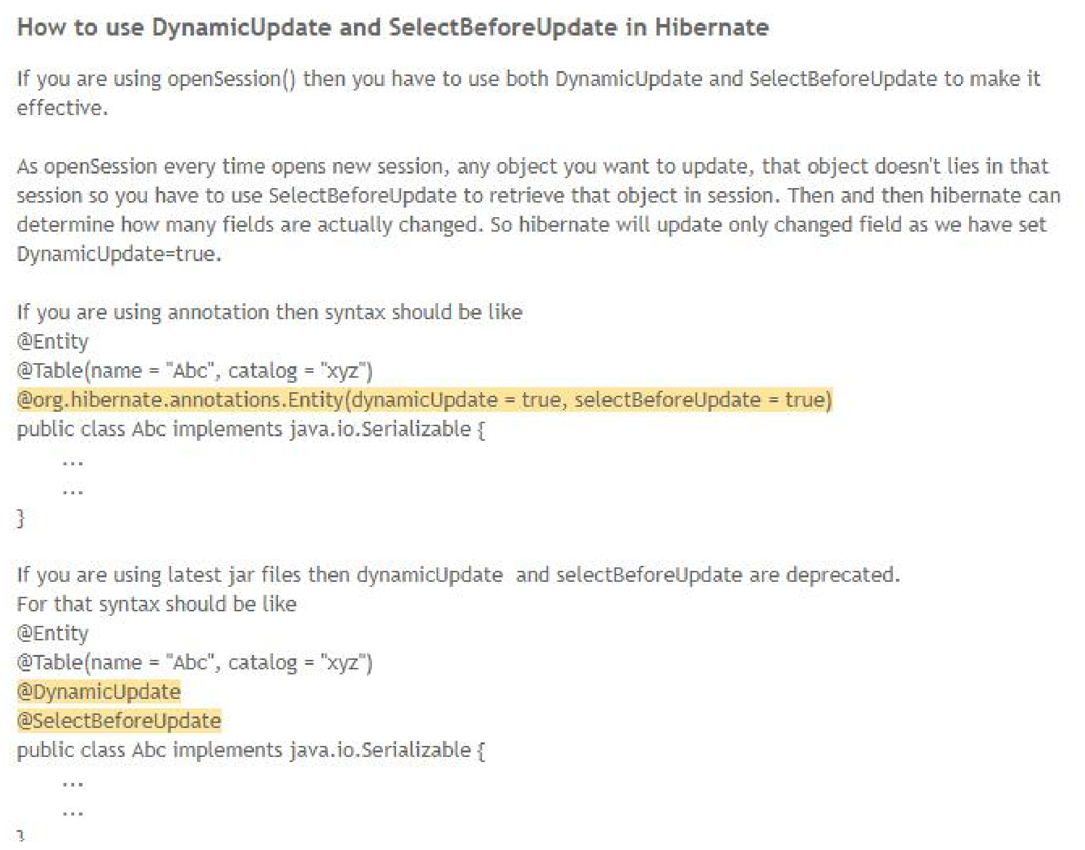


Object States

Persisting Detached Objects:



```
 */
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.setUserName("Updated Username after session close");


    session = sessionFactory.openSession();
    session.beginTransaction();
    session.update(user);
    session.getTransaction().commit();
    session.close();

}
```



How to use DynamicUpdate and SelectBeforeUpdate in Hibernate

If you are using openSession() then you have to use both DynamicUpdate and SelectBeforeUpdate to make it effective.

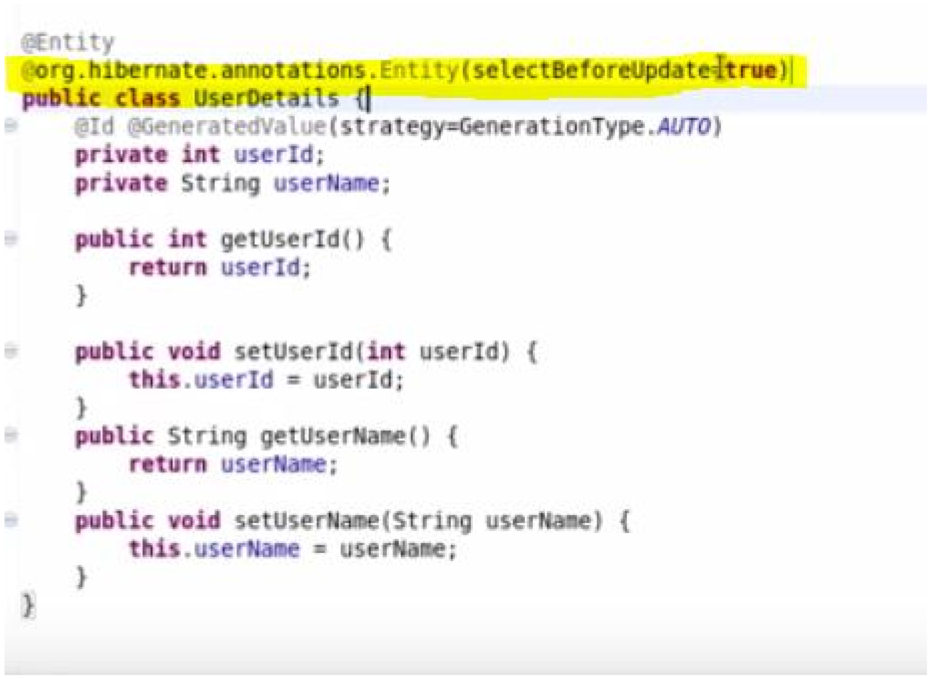
As openSession every time opens new session, any object you want to update, that object doesn't lies in that session so you have to use SelectBeforeUpdate to retrieve that object in session. Then and then hibernate can determine how many fields are actually changed. So hibernate will update only changed field as we have set DynamicUpdate=true.

If you are using annotation then syntax should be like

```
@Entity
@Table(name = "Abc", catalog = "xyz")
@org.hibernate.annotations.Entity(dynamicUpdate = true, selectBeforeUpdate = true)
public class Abc implements java.io.Serializable {
    ...
    ...
}
```

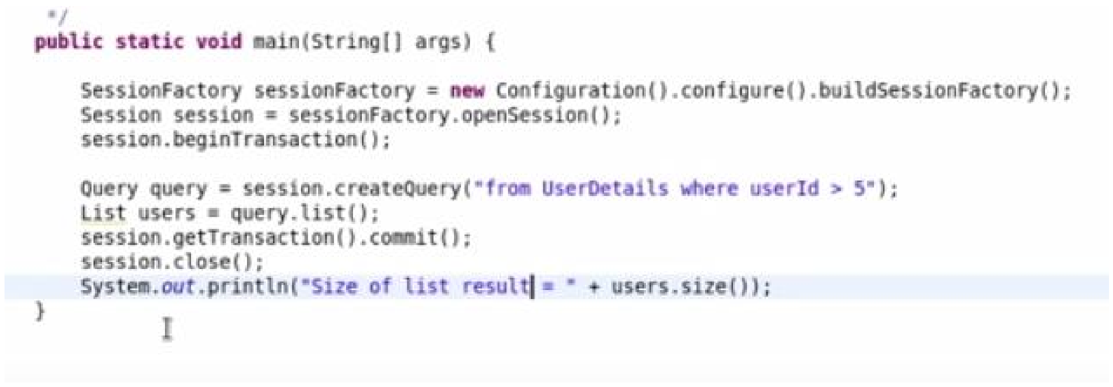
If you are using latest jar files then dynamicUpdate and selectBeforeUpdate are deprecated.
For that syntax should be like

```
@Entity
@Table(name = "Abc", catalog = "xyz")
@DynamicUpdate
@SelectBeforeUpdate
public class Abc implements java.io.Serializable {
    ...
    ...
}
```



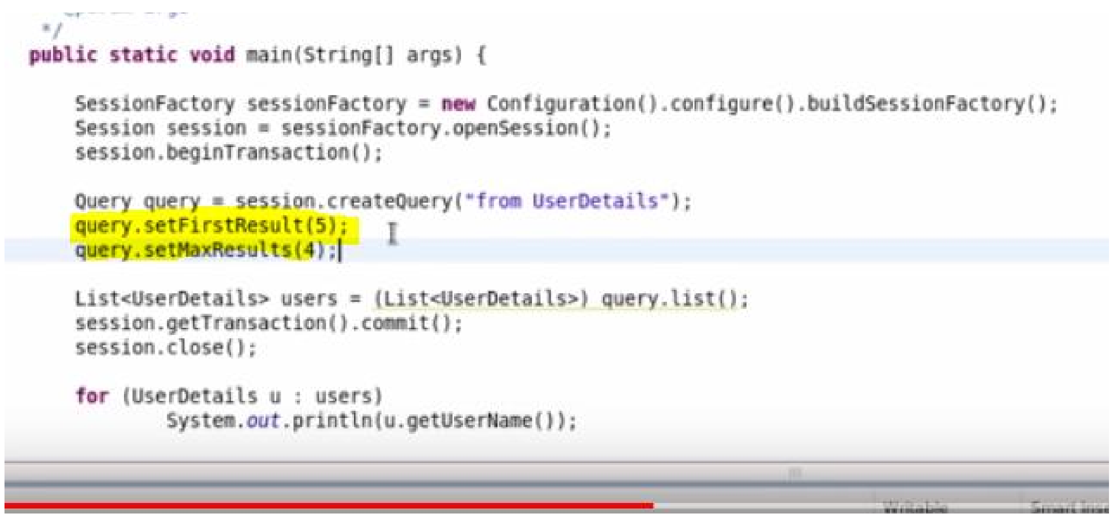
```
@Entity
@org.hibernate.annotations.Entity(selectBeforeUpdate=true)
public class UserDetails {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int userId;
    private String userName;

    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;
    }
}
```

Introducing HQL and the Query Object:



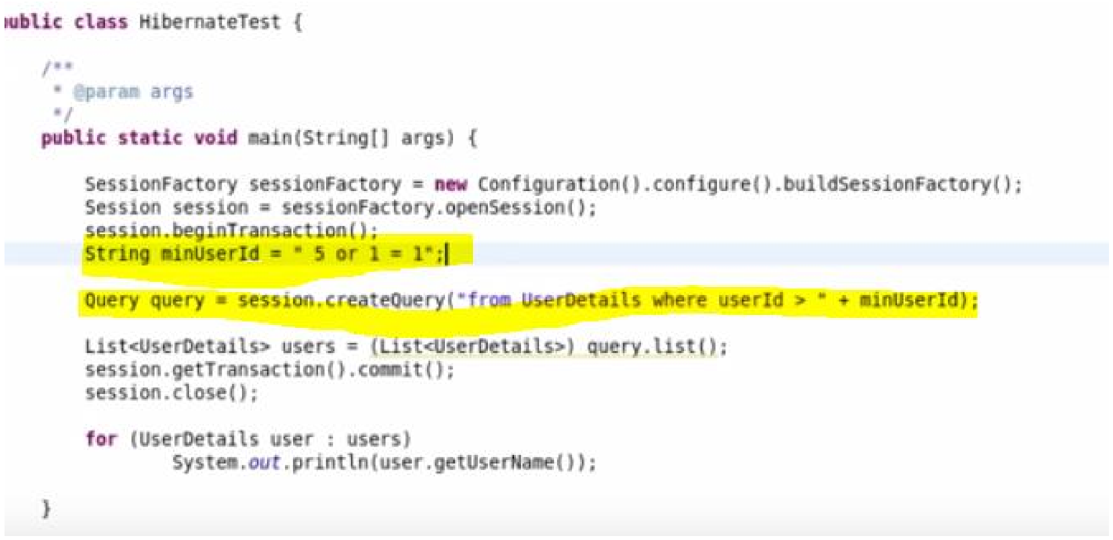
```
 */
public static void main(String[] args) {

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

    Query query = session.createQuery("from UserDetails where userId > 5");
    List users = query.list();
    session.getTransaction().commit();
    session.close();
    System.out.println("Size of list result = " + users.size());
}
```

Pagination:

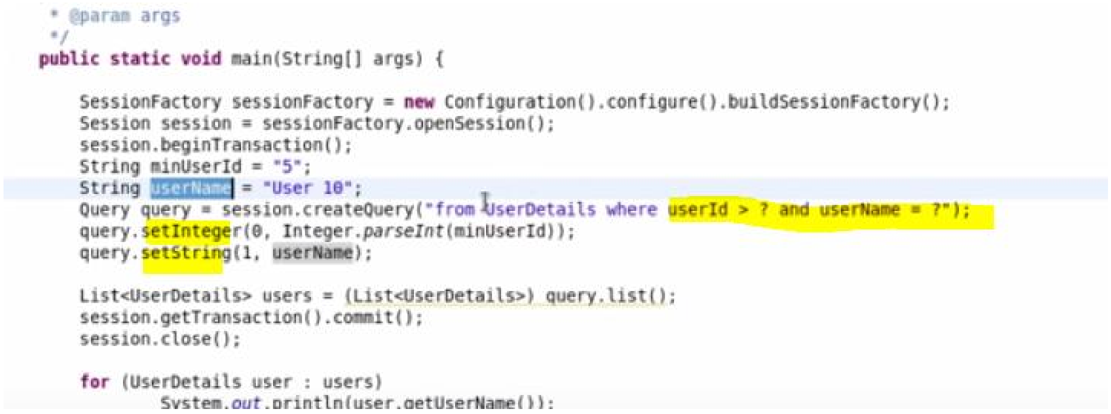


```
 */
public static void main(String[] args) {

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

    Query query = session.createQuery("from UserDetails");
    query.setFirstResult(5);
    query.setMaxResults(4);

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

    for (UserDetails u : users)
            System.out.println(u.getUserName());
```

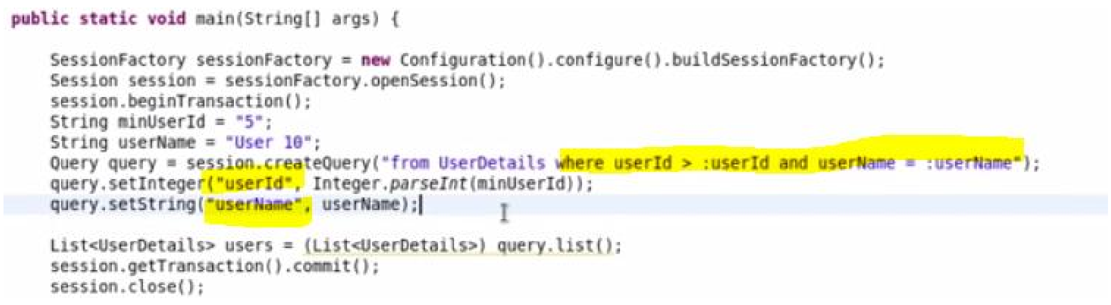
Understanding Parameter Binding and SQL Injection



```
ublic class HibernateTest {

    /**
     * @param args
     */
    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 user : users)
                System.out.println(user.getUserName());

    }
```

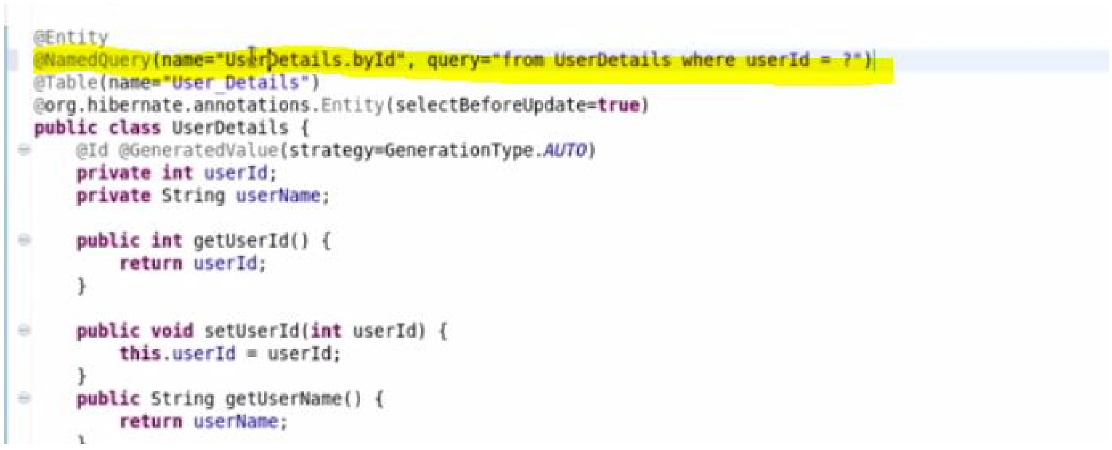


```
 * @param args
 */
public static void main(String[] args) {

    SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
    Session session = sessionFactory.openSession();
    session.beginTransaction();
    String minUserId = "5";
    String userName = "User 10";
    Query query = session.createQuery("from UserDetails where userId > ? and userName = ?");
    query.setInteger(0, Integer.parseInt(minUserId));
    query.setString(1, userName);

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

    for (UserDetails user : users)
        System.out.println(user.getUserName());
```

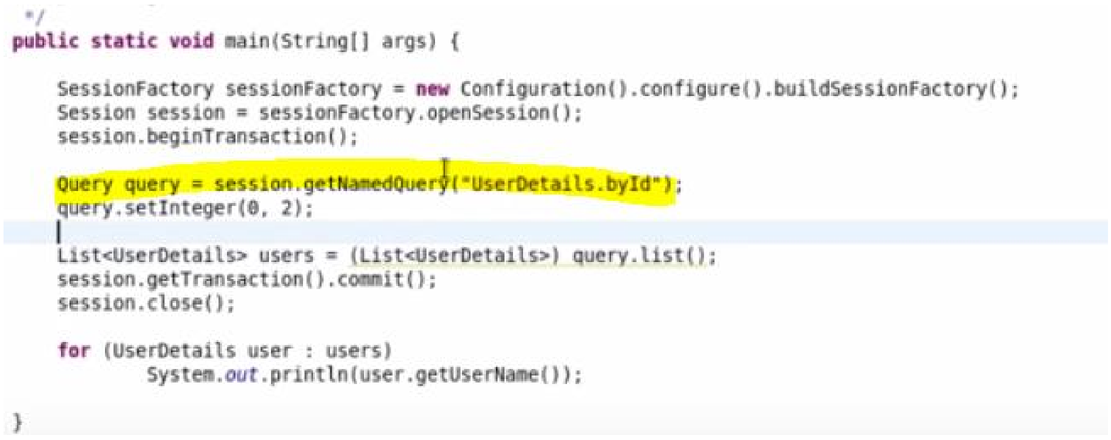


```
public static void main(String[] args) {

    SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
    Session session = sessionFactory.openSession();
    session.beginTransaction();
    String minUserId = "5";
    String userName = "User 10";
    Query query = session.createQuery("from UserDetails where userId > :userId and userName = :userName");
    query.setInteger("userId", Integer.parseInt(minUserId));
    query.setString("userName", userName);

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

Named Queries:

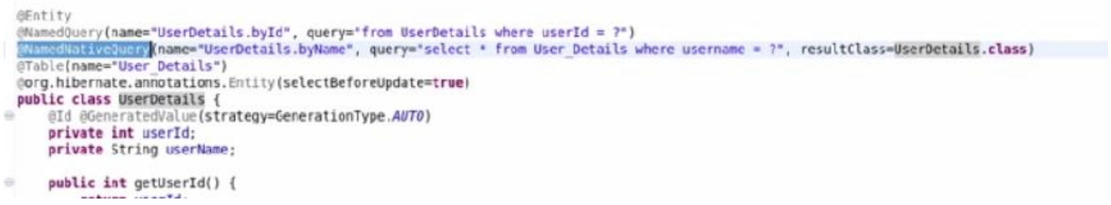


```
@Entity
@NamedQuery(name="UserDetails.byId", query="from UserDetails where userId = ?")
@Table(name="User_Details")
@org.hibernate.annotations.Entity(selectBeforeUpdate=true)
public class UserDetails {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int userId;
    private String userName;

    public int getUserId() {
        return userId;
    }

    public void setUserId(int userId) {
        this.userId = userId;
    }
    public String getUserName() {
        return userName;
```



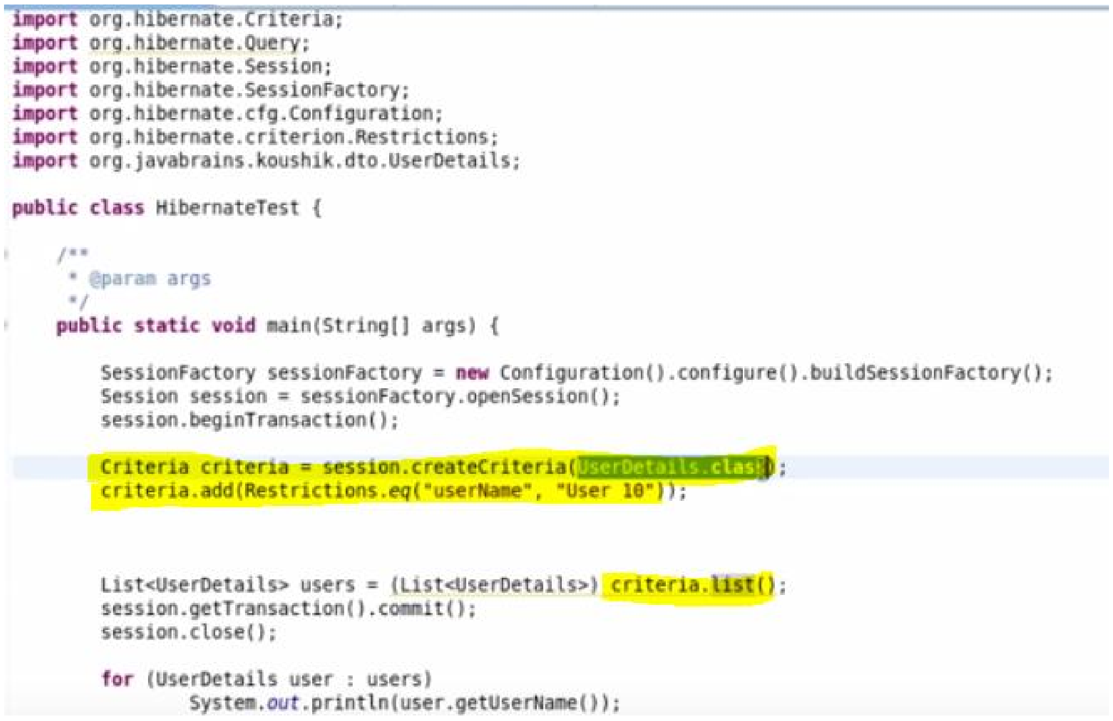
```
 */
public static void main(String[] args) {

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

    Query query = session.getNamedQuery("UserDetails.byId");
    query.setInteger(0, 2);

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

    for (UserDetails user : users)
            System.out.println(user.getUserName());

}
```

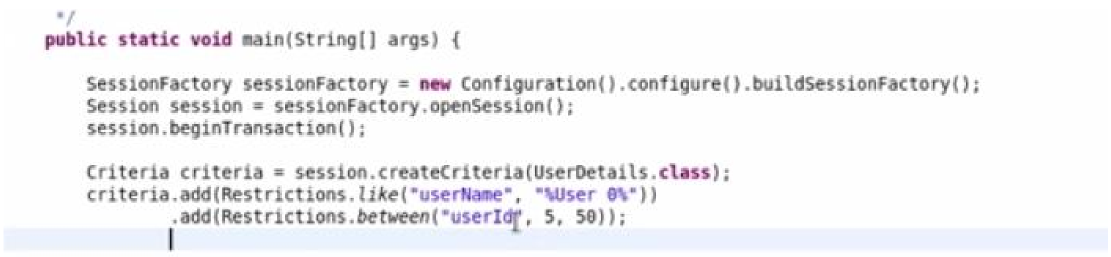
To execute native query,

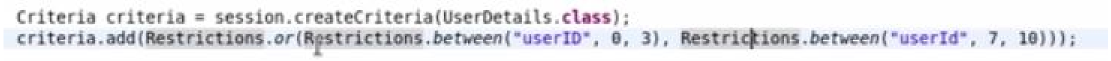


```
@Entity
@NamedQuery(name="UserDetails.byId", query="from UserDetails where userId = ?")
@NamedNativeQuery(name="UserDetails.byName", query="select * from User_Details where username = ?", resultClass=UserDetails.class)
@Table(name="User_Details")
@org.hibernate.annotations.Entity(selectBeforeUpdate=true)
public class UserDetails {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int userId;
    private String userName;

    public int getUserId() {
```

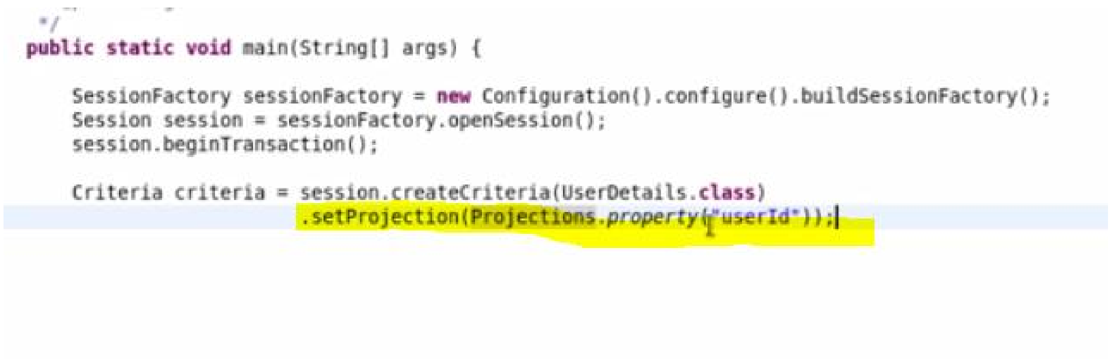
Introduction to Criteria API:


```java
import org.hibernate.Criteria;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import org.hibernate.criterion.Restrictions;
import org.javabrains.koushik.dto.UserDetails;

public class HibernateTest {

    /**
     * @param args
     */
    public static void main(String[] args) {

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

        Criteria criteria = session.createCriteria(UserDetails.class);
        criteria.add(Restrictions.eq("userName", "User 10"));


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

        for (UserDetails user : users)
            System.out.println(user.getUserName());
```

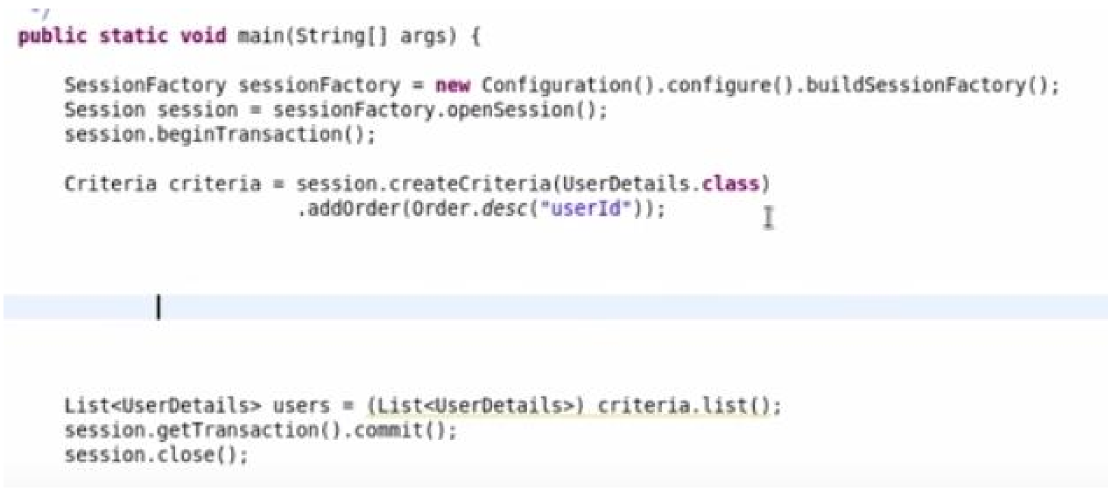

```java
     */
    public static void main(String[] args) {

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

        Criteria criteria = session.createCriteria(UserDetails.class);
        criteria.add(Restrictions.like("userName", "%User 0%"))
                .add(Restrictions.between("userId", 5, 50));
```


```java
Criteria criteria = session.createCriteria(UserDetails.class);
criteria.add(Restrictions.or(Restrictions.between("userID", 0, 3), Restrictions.between("userId", 7, 10)));
```

Projections and Query By Example



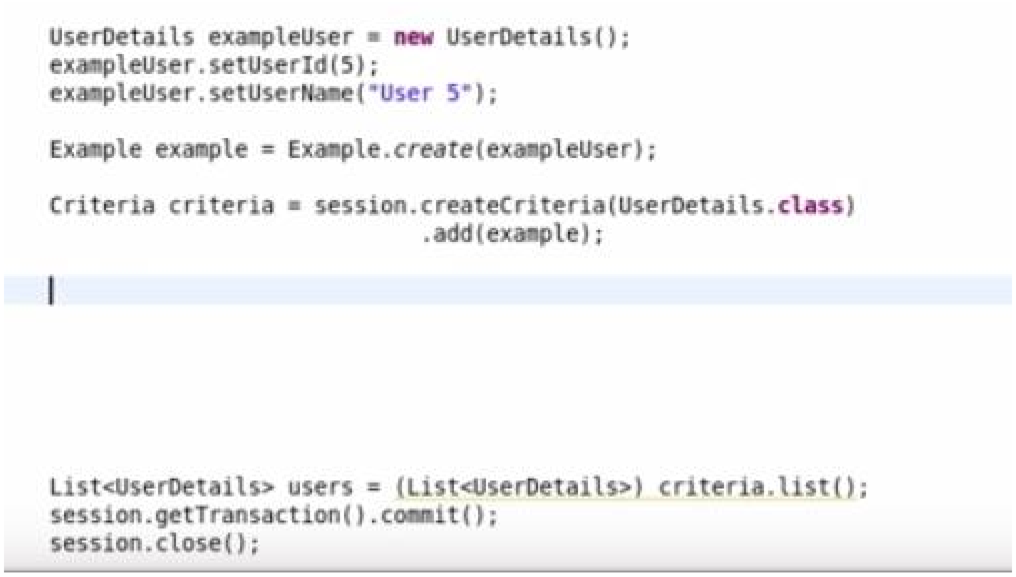
```
*/
public static void main(String[] args) {

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

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



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

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

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



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

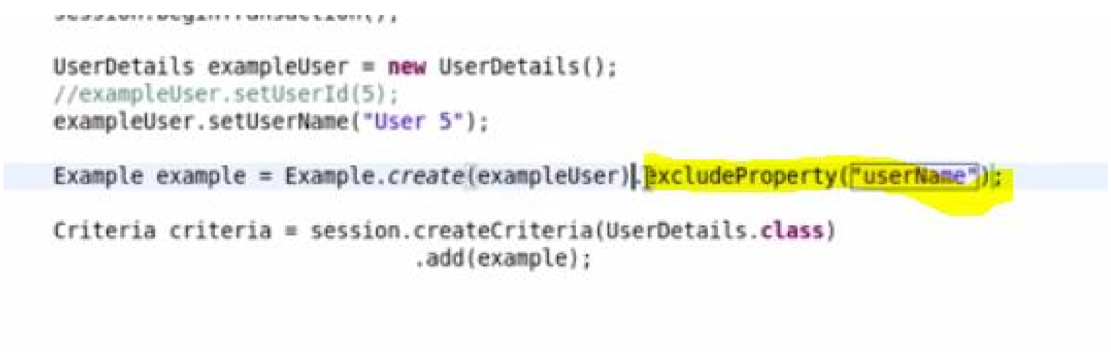
Query by Example:

Hibernate will ignore null properties and primary key in case of example query.



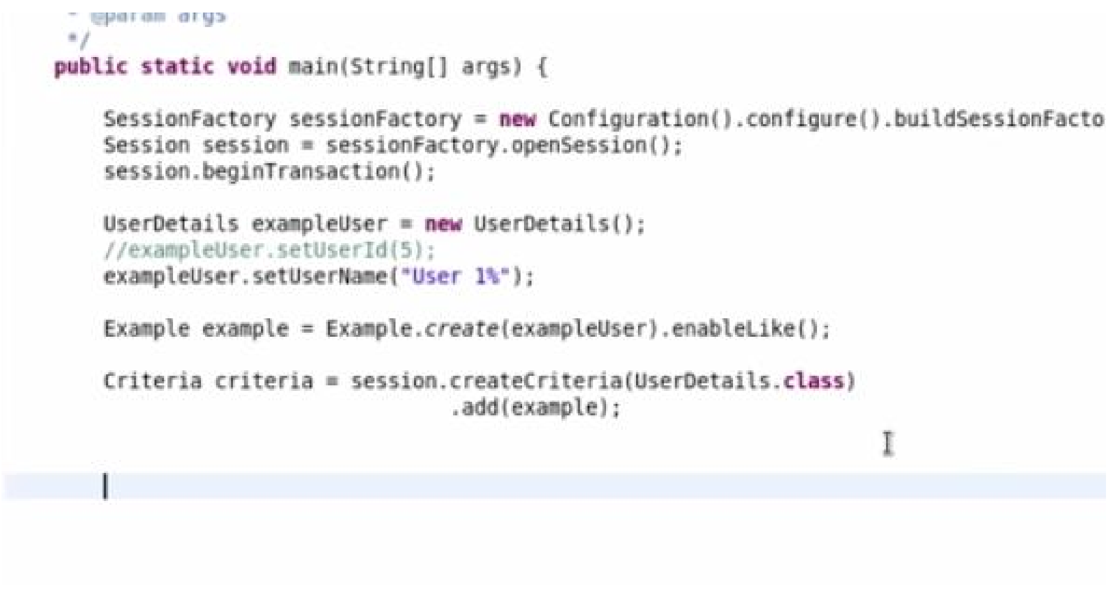
```
UserDetails exampleUser = new UserDetails();
exampleUser.setUserId(5);
exampleUser.setUserName("User 5");

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();
```

Excluding a property,

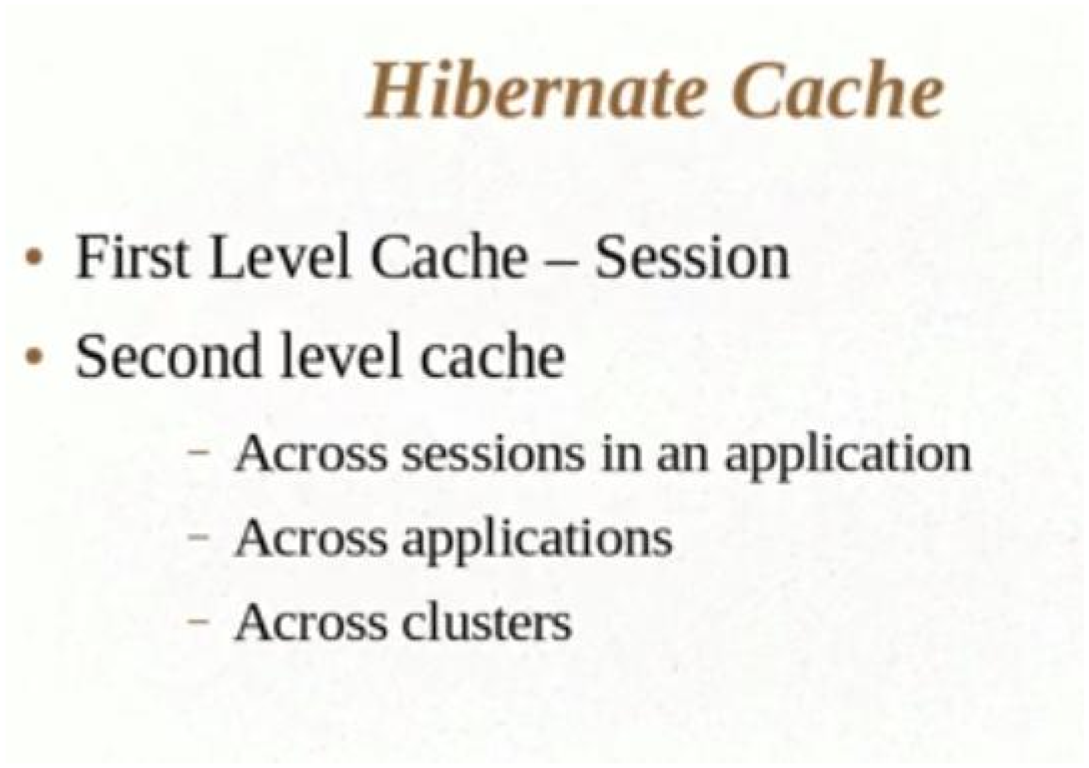


```
UserDetails exampleUser = new UserDetails();
//exampleUser.setUserId(5);
exampleUser.setUserName("User 5");

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

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

Like Operation,

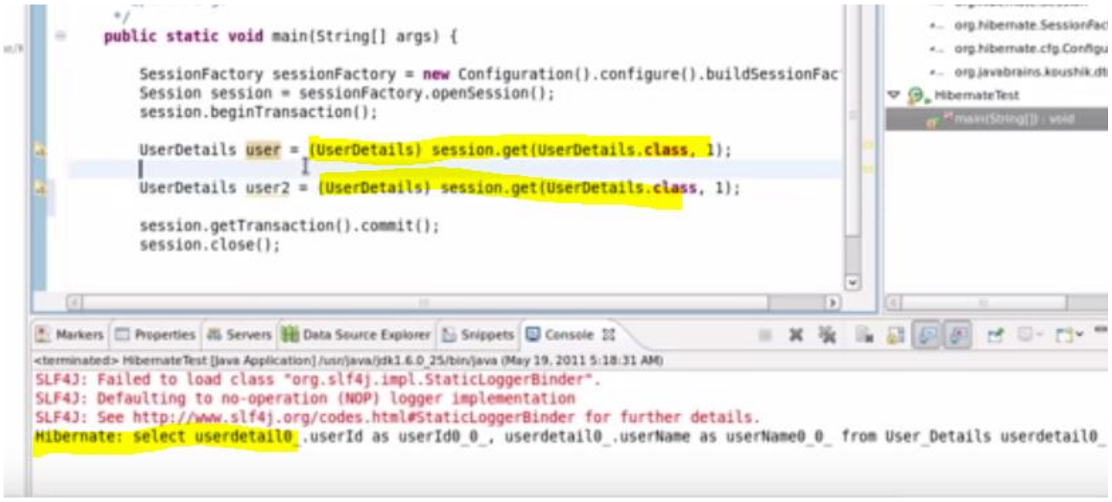


```
     * @param args
     */
    public static void main(String[] args) {

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

        UserDetails exampleUser = new UserDetails();
        //exampleUser.setUserId(5);
        exampleUser.setUserName("User 1%");

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

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

```



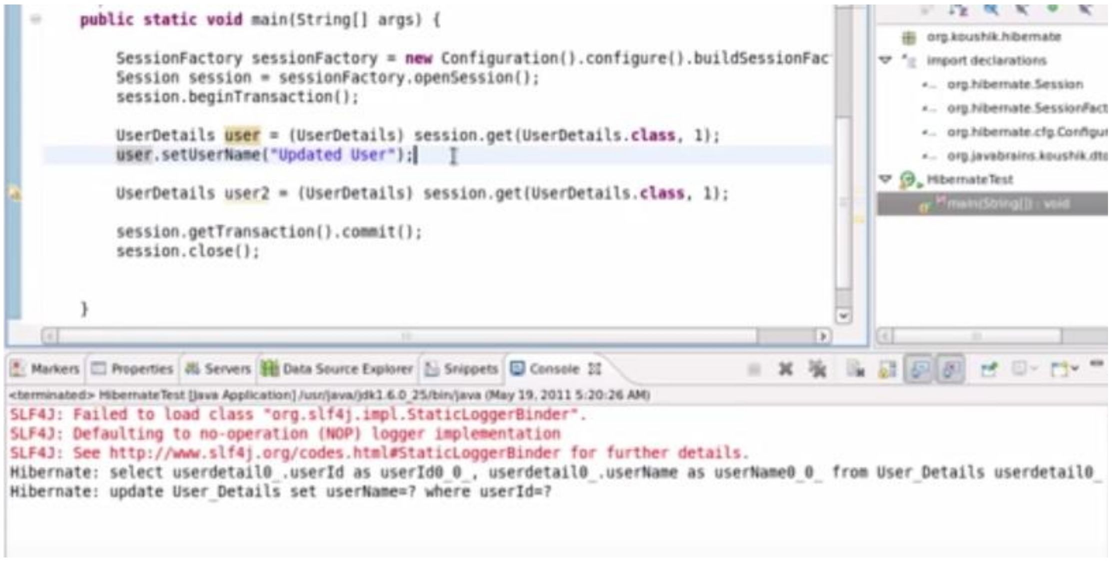
Hibernate Cache

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



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

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

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

        UserDetails user2 = (UserDetails) session.get(UserDetails.class, 1);

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

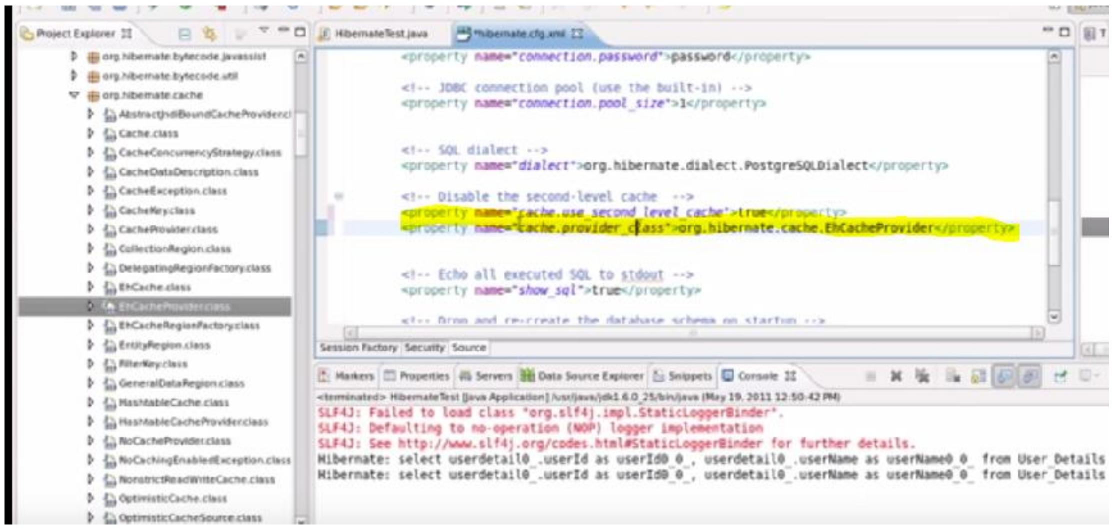
```
<terminated> HibernateTest [Java Application] /usr/java/jdk1.6.0_25/bin/java (May 19, 2011 5:18:31 AM)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Hibernate: select userdetail0_.userId as userId0_0_, userdetail0_.userName as userName0_0_ from User_Details userdetail0_
```



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

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

        UserDetails user = (UserDetails) session.get(UserDetails.class, 1);
        user.setUserName("Updated User");

        UserDetails user2 = (UserDetails) session.get(UserDetails.class, 1);

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

    }
```

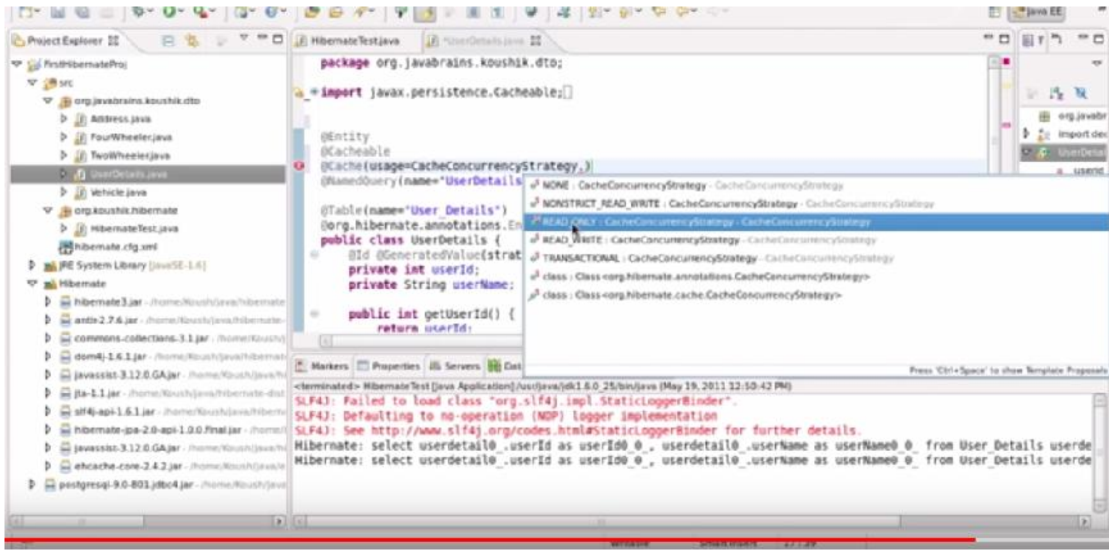
```
<terminated> HibernateTest [Java Application] /usr/java/jdk1.6.0_25/bin/java (May 19, 2011 5:20:26 AM)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Hibernate: select userdetail0_.userId as userId0_0_, userdetail0_.userName as userName0_0_ from User_Details userdetail0_
Hibernate: update User_Details set userName=? where userId=?
```

Configuring Second Level Cache



```
<property name="connection.password">password</property>

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


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

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


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

<!-- Drop and re-create the database schema on startup -->
```

```
<terminated> HibernateTest [Java Application] /usr/java/jdk1.6.0_25/bin/java (May 19, 2011 12:50:42 PM)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Hibernate: select userdetail0_.userId as userId0_0_, userdetail0_.userName as userName0_0_ from User_Details
Hibernate: select userdetail0_.userId as userId0_0_, userdetail0_.userName as userName0_0_ from User_Details
```

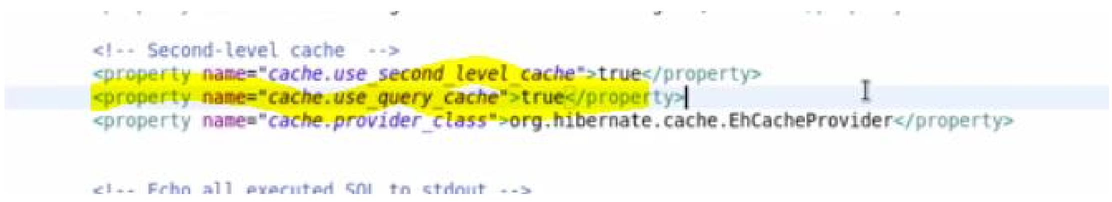


```
package org.javabrains.koushik.dto;

import javax.persistence.Cacheable;


@Entity
@Cacheable
@Cache(usage=CacheConcurrencyStrategy.)
@NamedQuery(name="UserDetails

@Table(name="User_Details")
@org.hibernate.annotations.En
public class UserDetails {
    @Id @GeneratedValue(strat
    private int userId;
    private String userName;

    public int getUserId() {
        return userId;
```

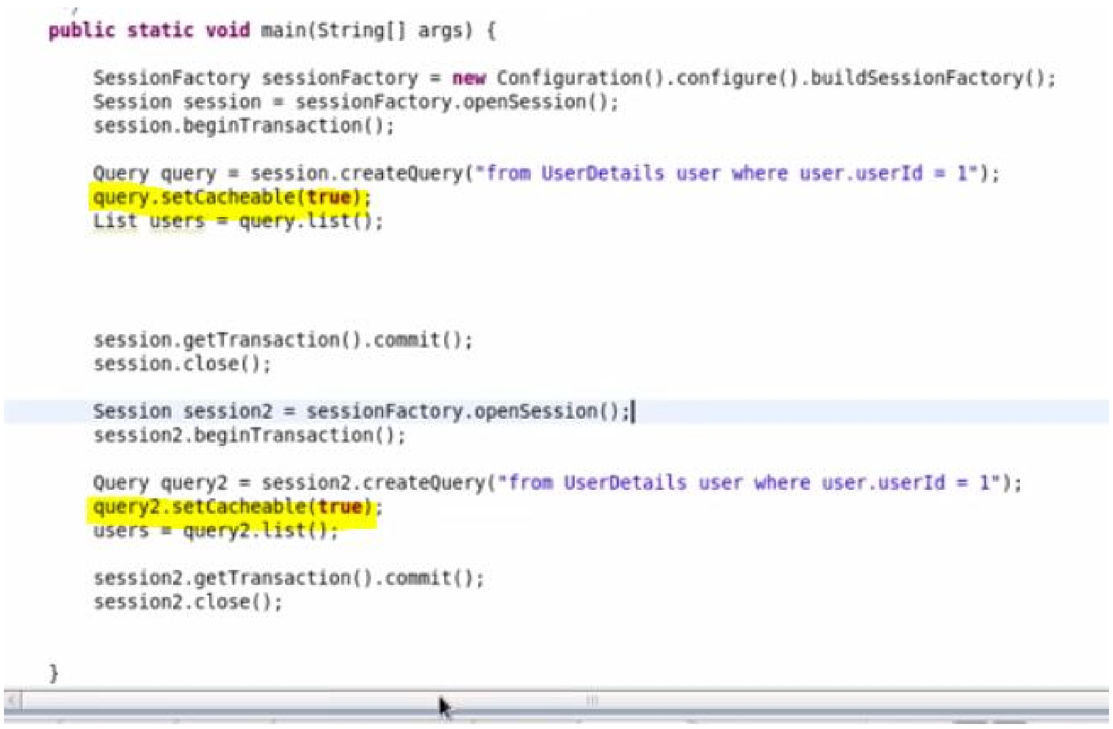
```
NONE : CacheConcurrencyStrategy - CacheConcurrencyStrategy
NONSTRICT_READ_WRITE : CacheConcurrencyStrategy - CacheConcurrencyStrategy
READ_ONLY : CacheConcurrencyStrategy - CacheConcurrencyStrategy
READ_WRITE : CacheConcurrencyStrategy - CacheConcurrencyStrategy
TRANSACTIONAL : CacheConcurrencyStrategy - CacheConcurrencyStrategy
class : Class<org.hibernate.annotations.CacheConcurrencyStrategy>
class : Class<org.hibernate.cache.CacheConcurrencyStrategy>
```

```
<terminated> HibernateTest [Java Application] /usr/java/jdk1.6.0_25/bin/java (May 19, 2011 12:50:42 PM)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Hibernate: select userdetail0_.userId as userId0_0_, userdetail0_.userName as userName0_0_ from User_Details userde
Hibernate: select userdetail0_.userId as userId0_0_, userdetail0_.userName as userName0_0_ from User_Details userde
```

Using Query Cache:



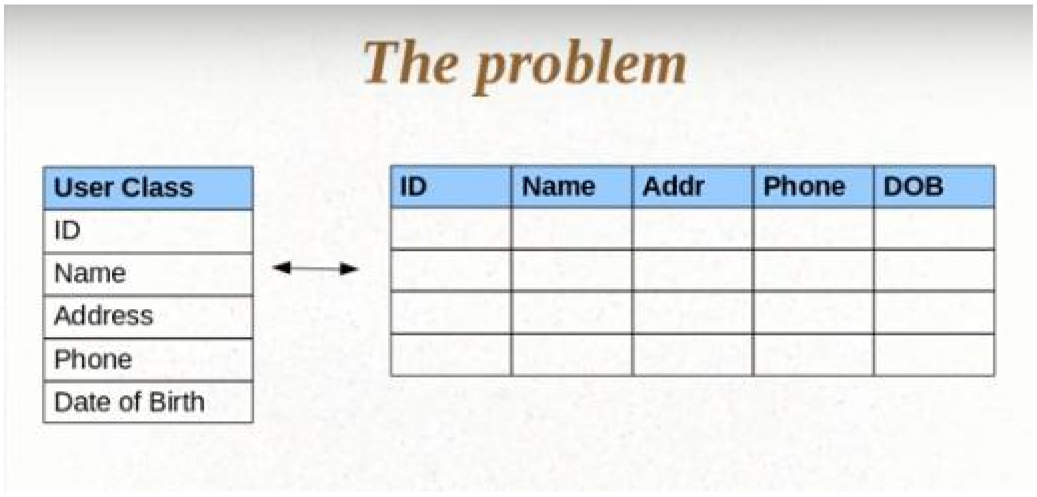
```
<!-- Second-level cache  -->
<property name="cache.use_second_level_cache">true</property>
<property name="cache.use_query_cache">true</property>
<property name="cache.provider_class">org.hibernate.cache.EhCacheProvider</property>


<!-- Echo all executed SQL to stdout -->
```



```
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 = 1");
    query.setCacheable(true);
    List users = query.list();



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

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

    Query query2 = session2.createQuery("from UserDetails user where user.userId = 1");
    query2.setCacheable(true);
    users = query2.list();

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

}
```

==

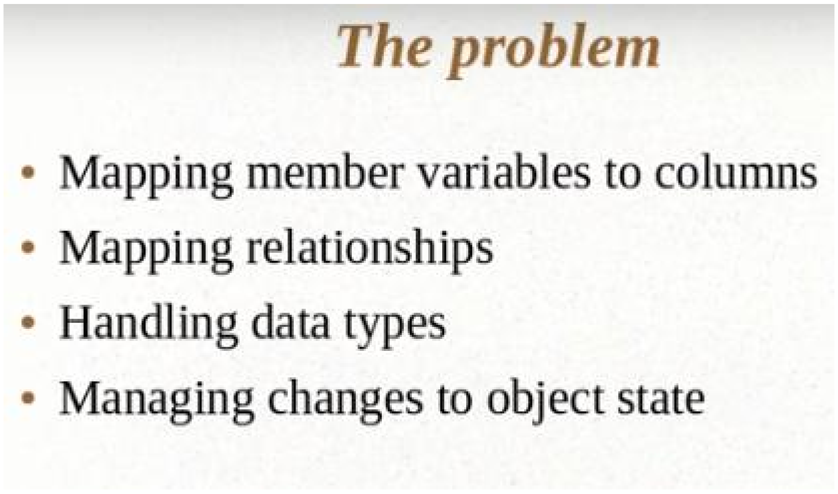
JavaBrains:



The problem

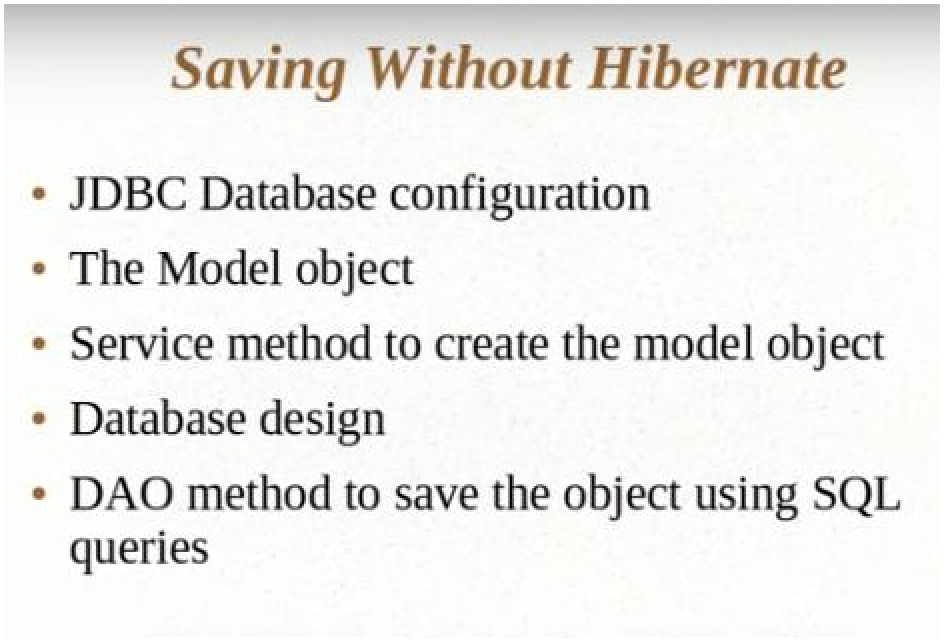
User Class
ID
Name
Address
Phone
Date of Birth

ID	Name	Addr	Phone	DOB



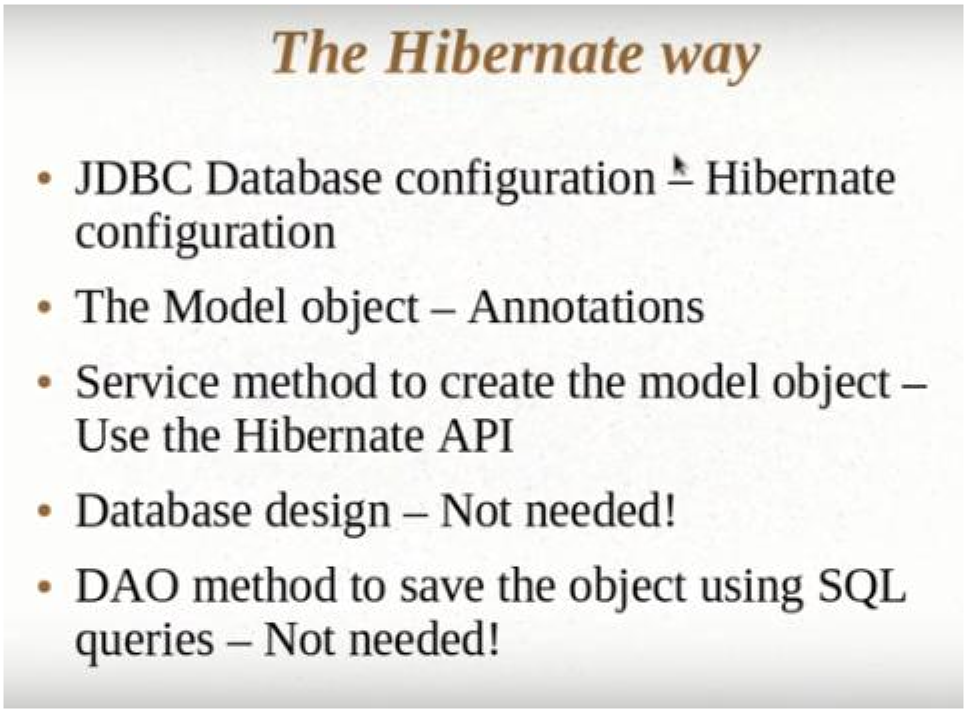
The problem

- Mapping member variables to columns
- Mapping relationships
- Handling data types
- Managing changes to object state



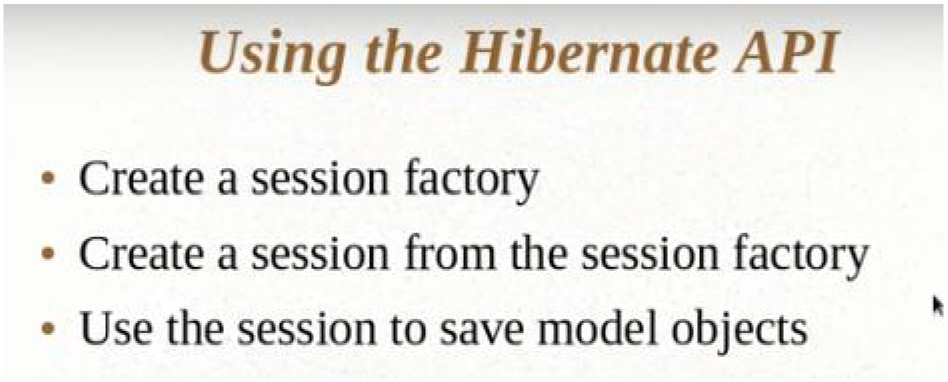
Saving Without Hibernate

- 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

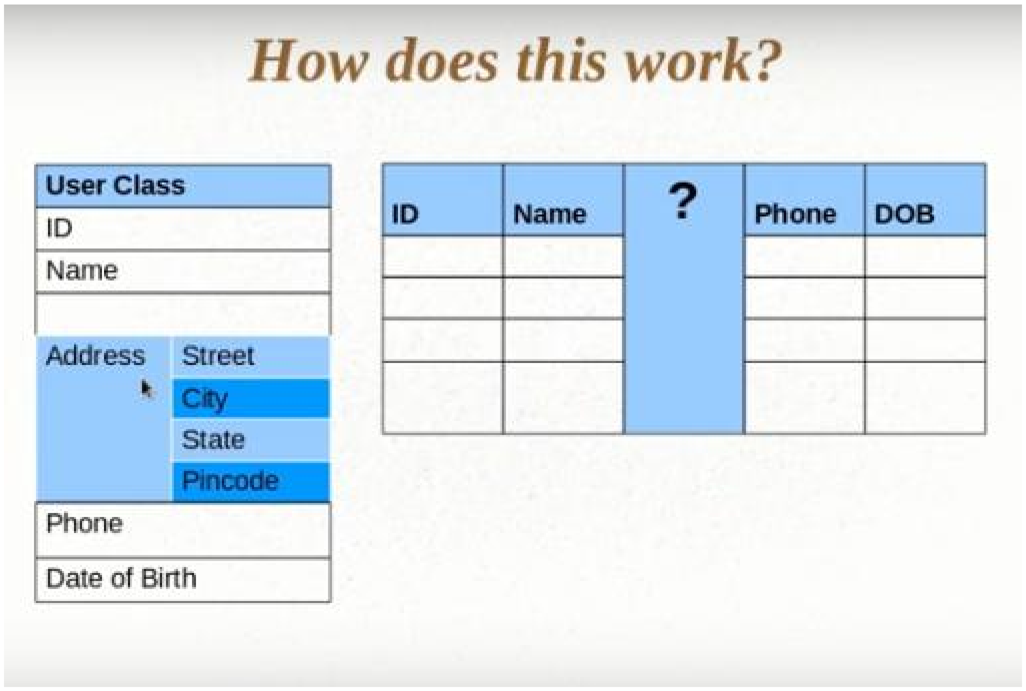
- JDBC Database configuration – Hibernate configuration
- The Model object – Annotations
- Service method to create the model object – Use the Hibernate API
- Database design – Not needed!
- DAO method to save the object using SQL queries – Not needed!



Using the Hibernate API

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

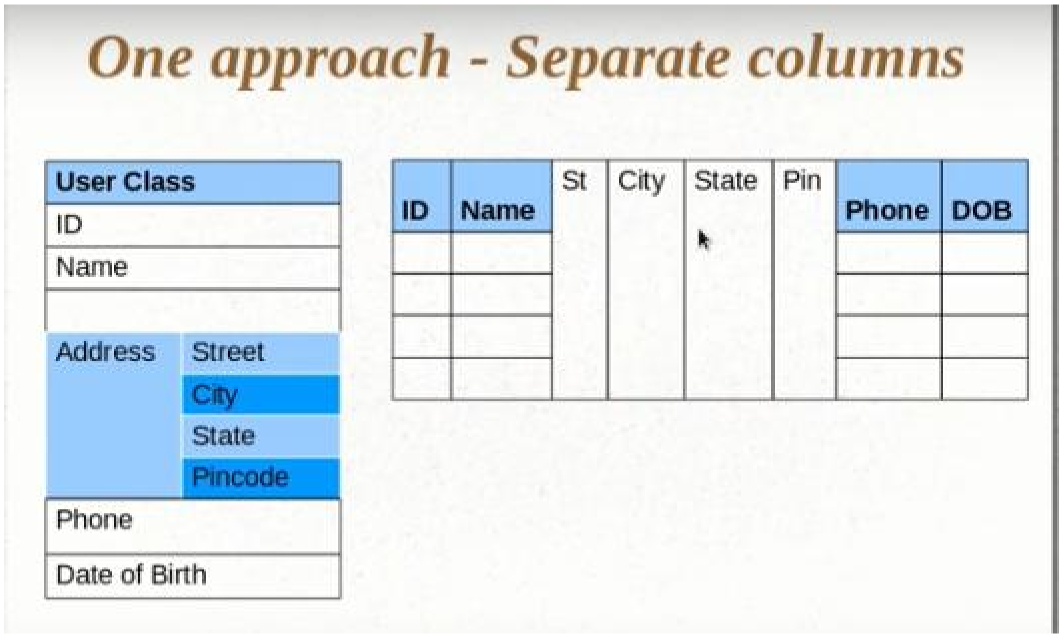
If we do not want to save a property we can mark that as transient or static.



How does this work?

User Class	
ID	
Name	
Address	Street
	City
	State
	Pincode
Phone	
Date of Birth	

ID	Name	?	Phone	DOB



One approach - Separate columns

User Class	
ID	
Name	
Address	Street
	City
	State
	Pincode
Phone	
Date of Birth	

ID	Name	St	City	State	Pin	Phone	DOB

An Entity has meaning of it's own

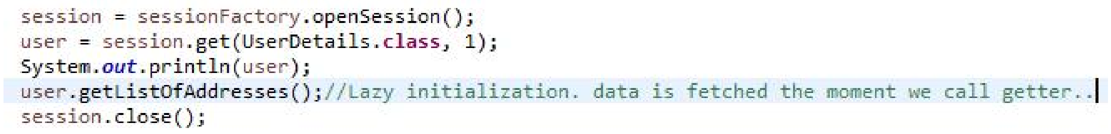
A value object does not have meaning of it's own.

In the above case Adress will act as a value object not

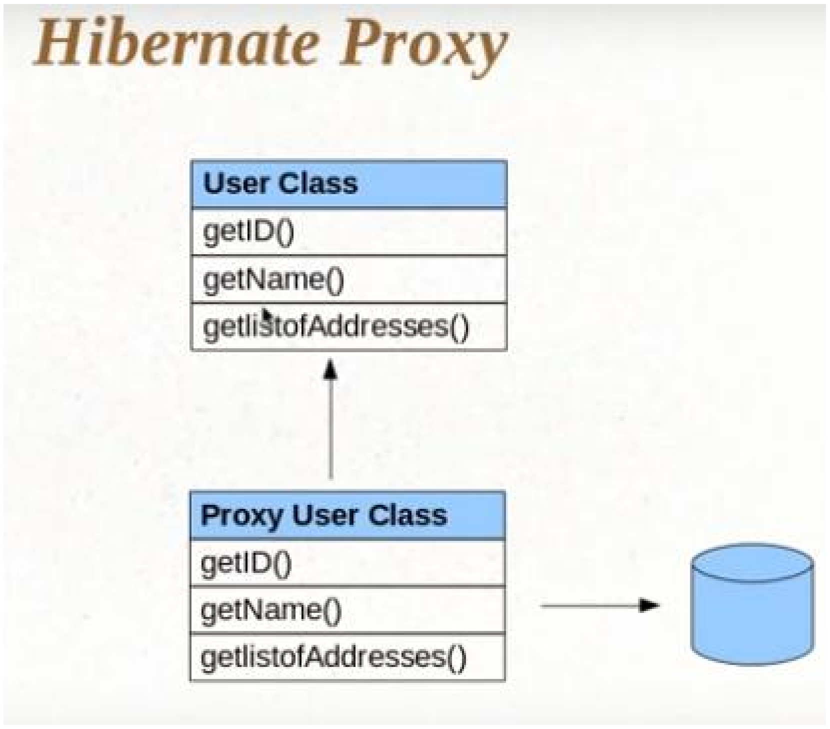
an entity object.

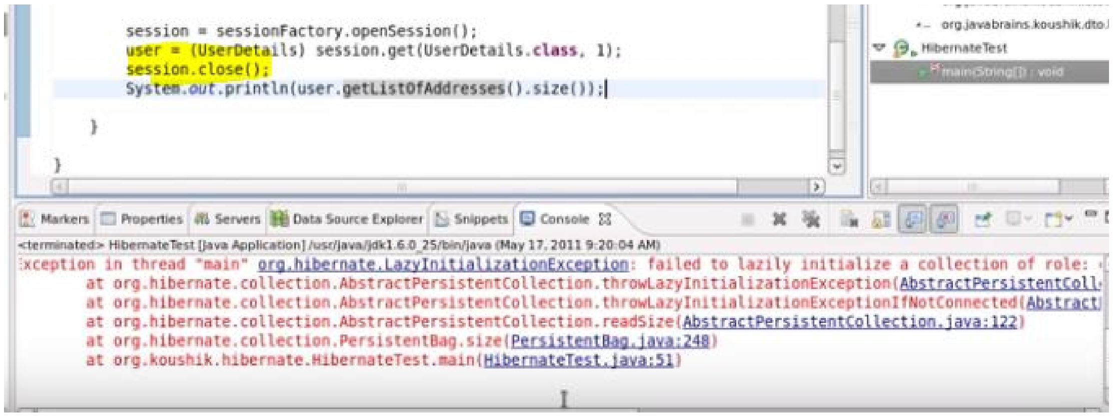
if emeded object is primary key use @EmbededId

Proxy Objects and Eager and Lazy Fetch Types:



```
session = sessionFactory.openSession();
user = session.get(UserDetails.class, 1);
System.out.println(user);
user.getListOfAddresses();//Lazy initialization. data is fetched the moment we call getter..
session.close();
```

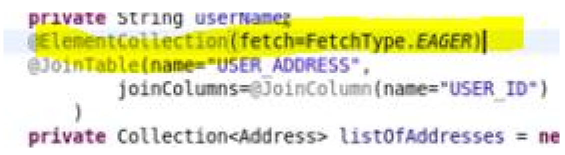




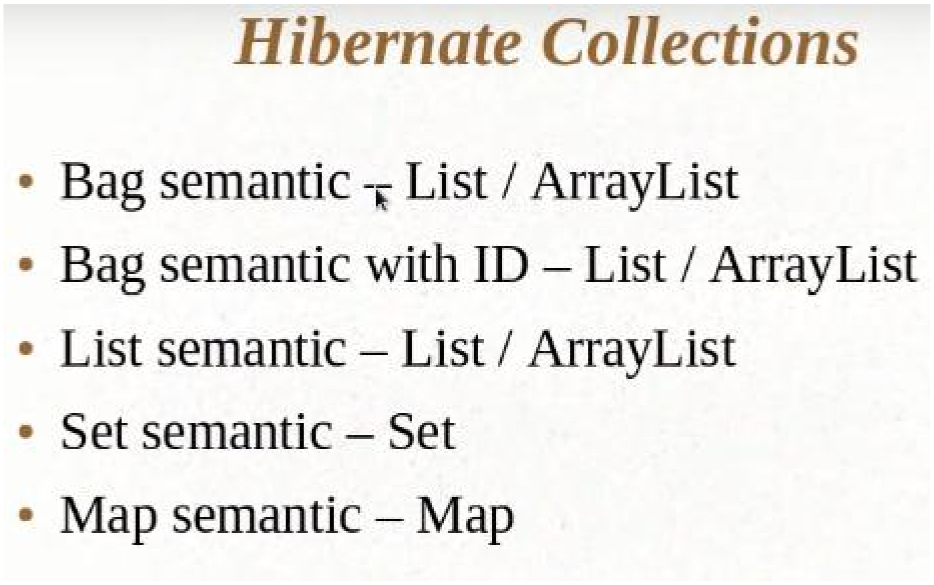
```
        session = sessionFactory.openSession();
        user = (UserDetails) session.get(UserDetails.class, 1);
        session.close();
        System.out.println(user.getListOfAddresses().size());

    }

}
```

```
<terminated> HibernateTest [Java Application] /usr/java/jdk1.6.0_25/bin/java (May 17, 2011 9:20:04 AM)
xception in thread "main" org.hibernate.LazyInitializationException: failed to lazily initialize a collection of role:
	at org.hibernate.collection.AbstractPersistentCollection.throwLazyInitializationException(AbstractPersistentColl
	at org.hibernate.collection.AbstractPersistentCollection.throwLazyInitializationExceptionIfNotConnected(Abstract
	at org.hibernate.collection.AbstractPersistentCollection.readSize(AbstractPersistentCollection.java:122)
	at org.hibernate.collection.PersistentBag.size(PersistentBag.java:248)
	at org.koushik.hibernate.HibernateTest.main(HibernateTest.java:51)
```

To fetch eagerly

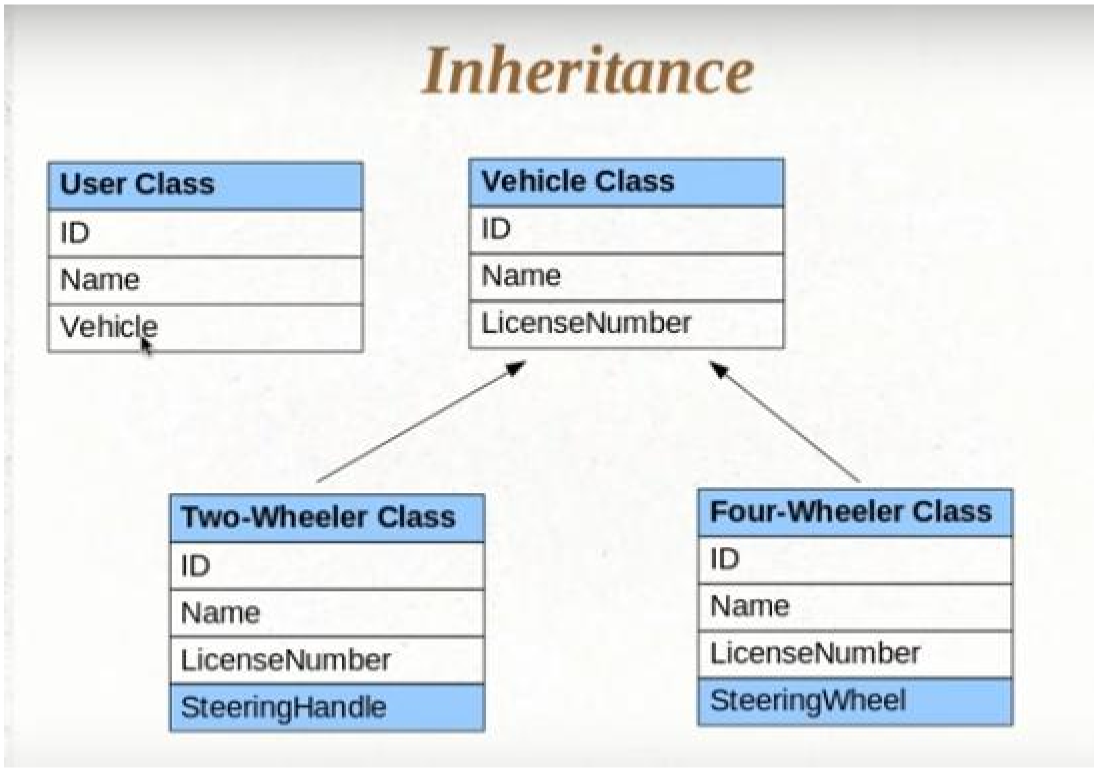


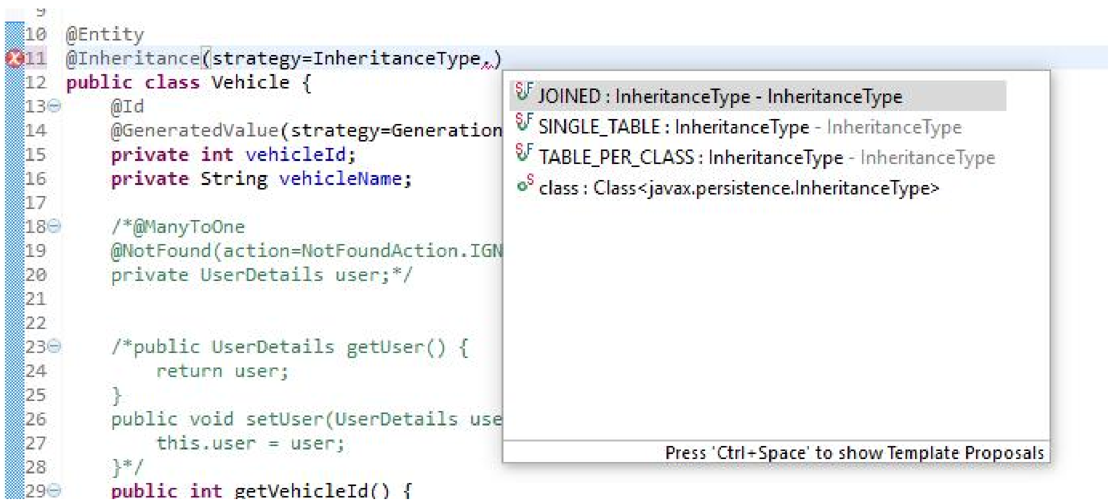
```
private String userName;
@ElementCollection(fetch=FetchType.EAGER)
@JoinTable(name="USER_ADDRESS",
        joinColumns=@JoinColumn(name="USER_ID")
    )
private Collection<Address> listOfAddresses = ne
```



Hibernate Collections

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

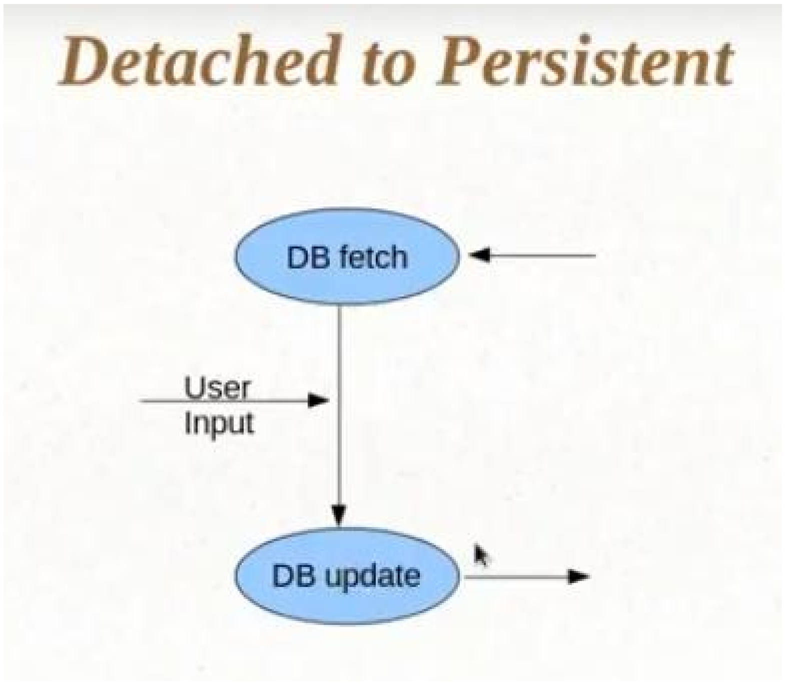




```
@Entity
@Inheritance(strategy=InheritanceType.)
public class Vehicle {
    @Id
    @GeneratedValue(strategy=Generation
    private int vehicleId;
    private String vehicleName;

    /*@ManyToOne
    @NotFound(action=NotFoundAction.IGN
    private UserDetails user;*/


    /*public UserDetails getUser() {
        return user;
    }
    public void setUser(UserDetails use
        this.user = user;
    }*/
    public int getVehicleId() {
```

JOINED : InheritanceType - InheritanceType
SINGLE_TABLE : InheritanceType - InheritanceType
TABLE_PER_CLASS : InheritanceType - InheritanceType
class : Class<javax.persistence.InheritanceType>
Press 'Ctrl+Space' to show Template Proposals

A Discriminator column is added which keeps track of the class names.



Projections:

Can use aggregate functions by using projections,

e.g max Id record etc..