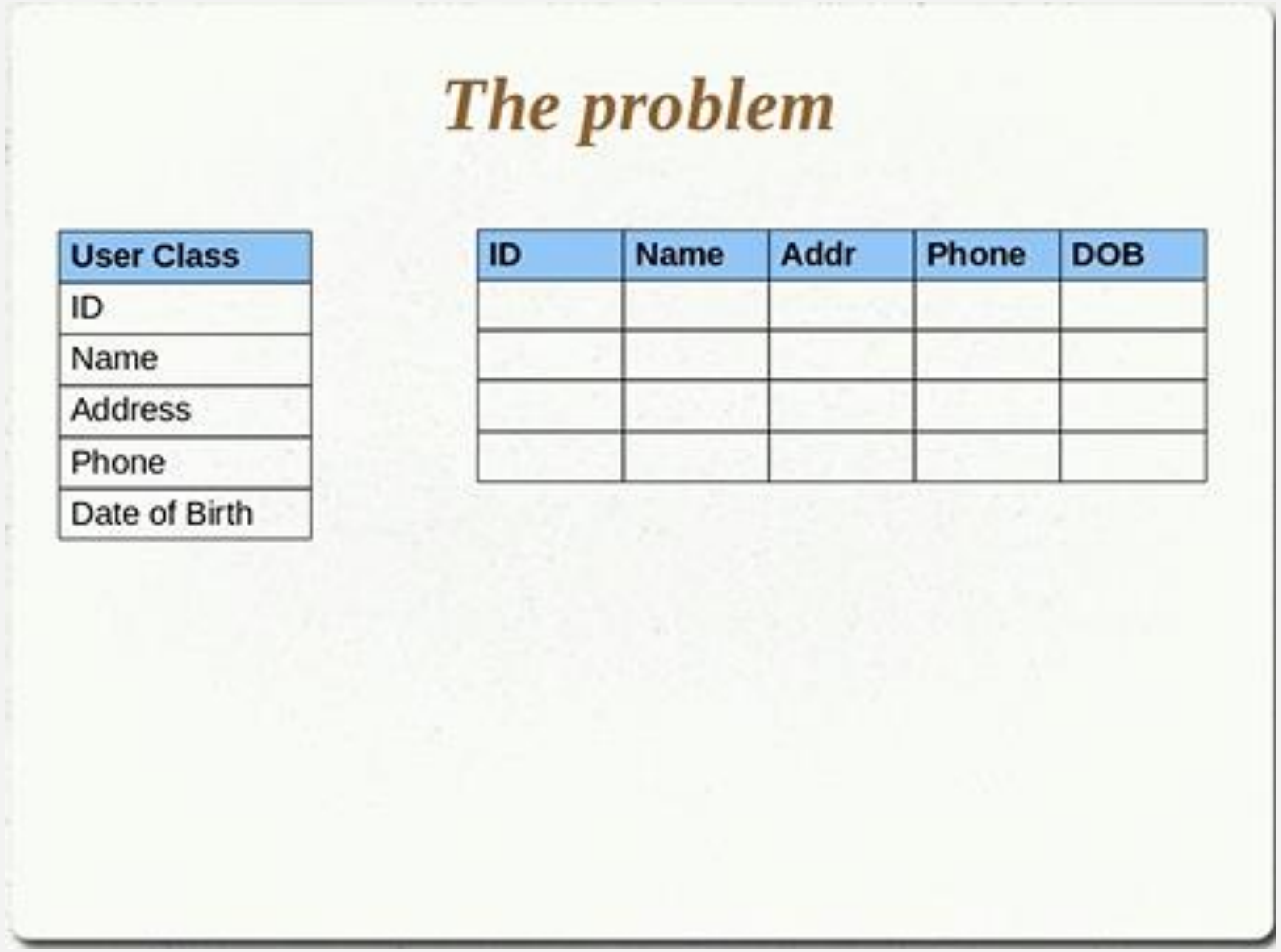
HIBERNATE INTRODUCTION

- An ORM tool
- Used in the data layer of applications
- Implements JPA

THE PROBLEM



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 xml file
- The Model object – Annotations or Xml
- 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!

SETTING UP HIBERNATE

- To Download Hibernate Jar files
  - www.hibernate.org/downloads
- Install plugin Eclipse – **Jboss Hibernate tools**
- Create Project in Eclipse
- Create User Library and add all jar files into it and set that library for your project

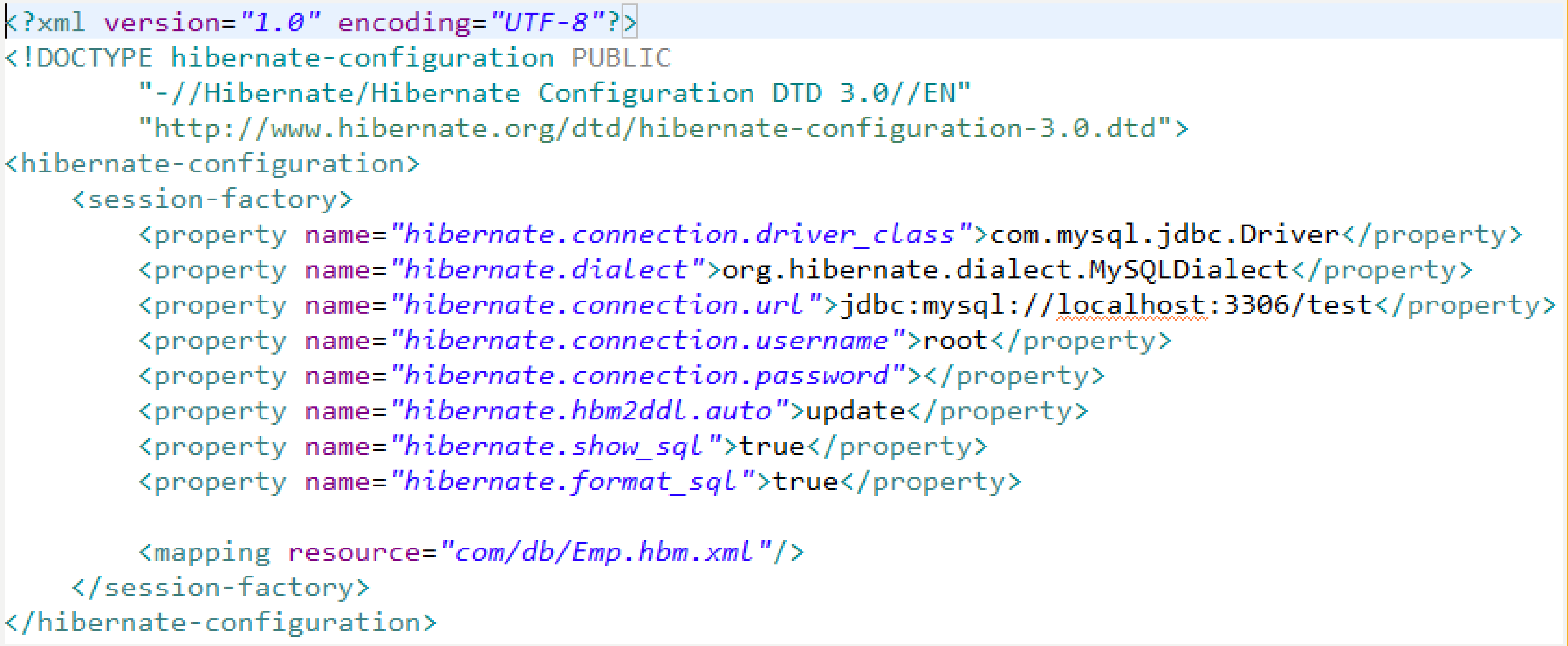
BACK TO YOUR PROJECT

- Add Hibernate configuration file - hibernate.cfg.xml
- Add Following property

Property	Purpose
hibernate.connection.driver_class	The JDBC driver class
hibernate.dialect	This property makes Hibernate generate the appropriate SQL for the chosen database
hibernate.connection.url	The JDBC URL to the database instance
hibernate.connection.username	The database username
hibernate.connection.password	The database password
hibernate.hbm2ddl.auto	Automatically validates or exports schema DDL to the database when the SessionFactory is created. With create-drop, the database schema will be dropped when the SessionFactory is closed explicitly **e.g.** validate | update | create | create-drop

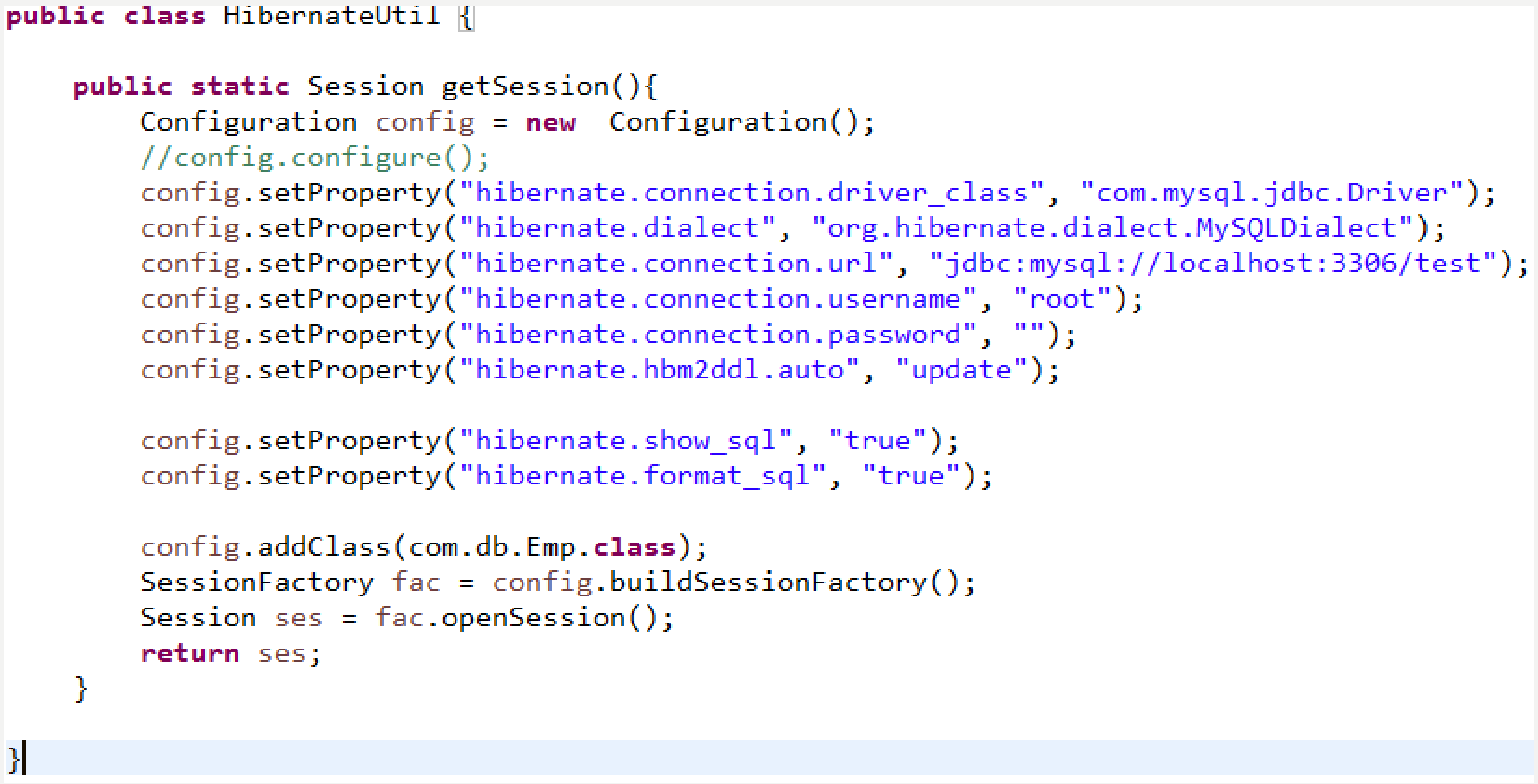
Property	Purpose
hibernate.show_sql	Write all SQL statements to console. **e.g.** true | false
hibernate.format_sql	Pretty print the SQL in the log and console. **e.g.** true | false
hibernate.connection.pool_size	Limits the number of connections waiting in the Hibernate database connection pool.
hibernate.connection.autocommit	Allows auto commit mode to be used for the JDBC connection

HIBERNATE.CFG.XML



```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/test</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password"></property>
        <property name="hibernate.hbm2ddl.auto">update</property>
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>

        <mapping resource="com/db/Emp.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

PROGRAMMATIC CONFIGURATION

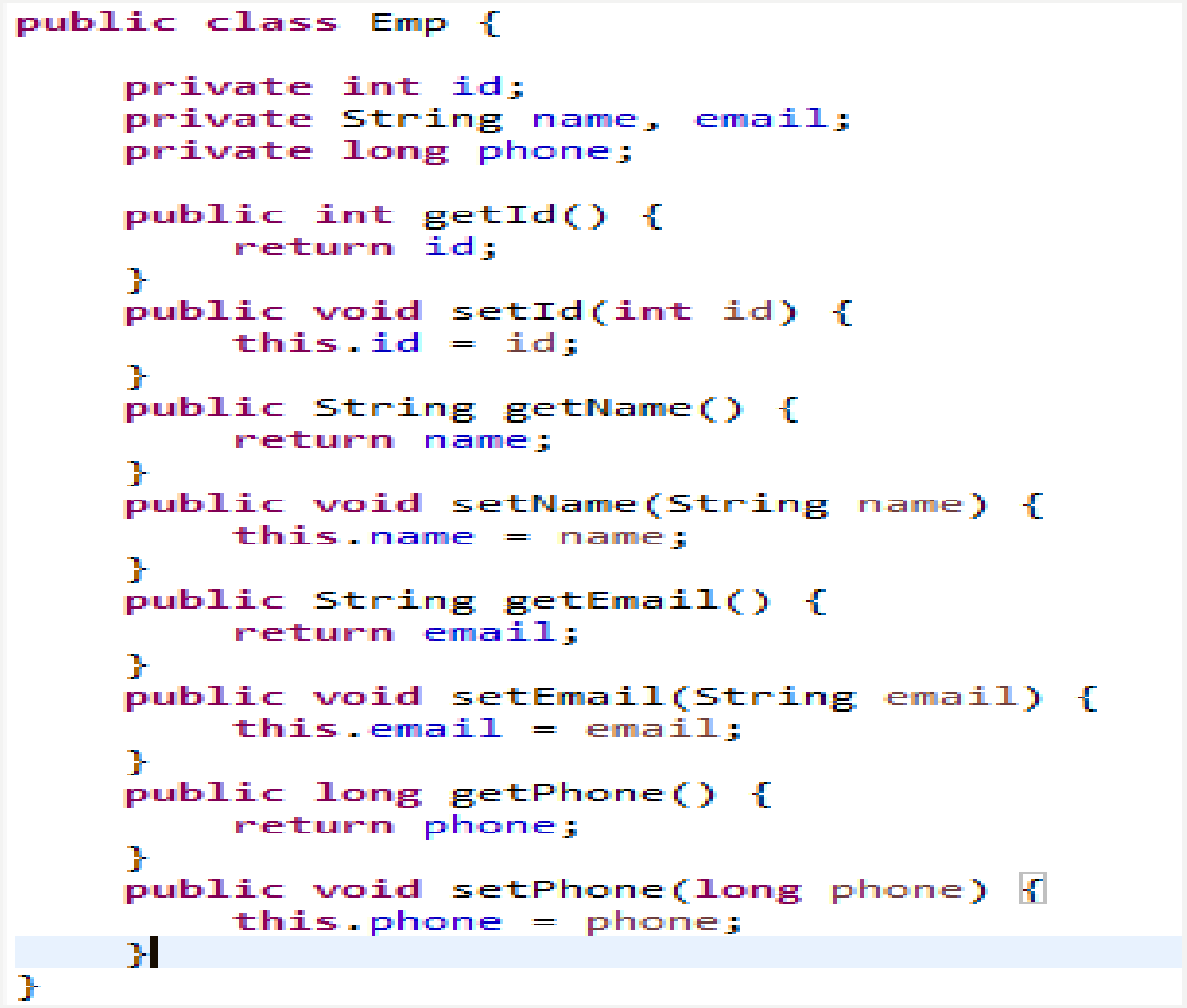


```java
public class HibernateUtil {

    public static Session getSession(){
        Configuration config = new  Configuration();
        //config.configure();
        config.setProperty("hibernate.connection.driver_class", "com.mysql.jdbc.Driver");
        config.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLDialect");
        config.setProperty("hibernate.connection.url", "jdbc:mysql://localhost:3306/test");
        config.setProperty("hibernate.connection.username", "root");
        config.setProperty("hibernate.connection.password", "");
        config.setProperty("hibernate.hbm2ddl.auto", "update");

        config.setProperty("hibernate.show_sql", "true");
        config.setProperty("hibernate.format_sql", "true");

        config.addClass(com.db.Emp.class);
        SessionFactory fac = config.buildSessionFactory();
        Session ses = fac.openSession();
        return ses;
    }

}
```

USING HIBERNATE API

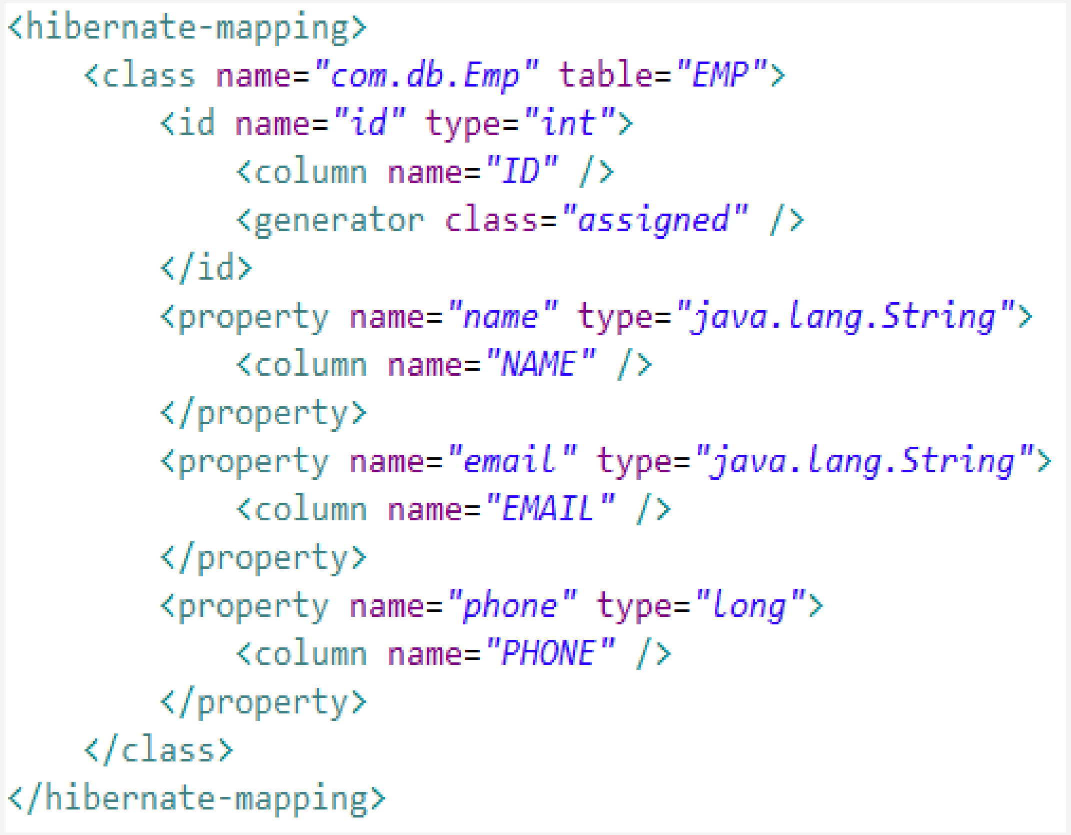
- Create a session factory
  - One object per application
  - It is created out of the configuration of hibernate file (hibernate.cfg.xml)
- Create a session from the session factory
- Use the session to save model objects

CREATE MODAL CLASS



```
public class Emp {

	private int id;
	private String name, email;
	private long phone;

	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public String getEmail() {
		return email;
	}
	public void setEmail(String email) {
		this.email = email;
	}
	public long getPhone() {
		return phone;
	}
	public void setPhone(long phone) {
		this.phone = phone;
	}
}
```

HIBERNATE MODEL MAPPING – E.G. EMP.HBM.XML



```xml
<hibernate-mapping>
    <class name="com.db.Emp" table="EMP">
        <id name="id" type="int">
            <column name="ID" />
            <generator class="assigned" />
        </id>
        <property name="name" type="java.lang.String">
            <column name="NAME" />
        </property>
        <property name="email" type="java.lang.String">
            <column name="EMAIL" />
        </property>
        <property name="phone" type="long">
            <column name="PHONE" />
        </property>
    </class>
</hibernate-mapping>
```

Generator

Assigned – lets the application assign an identifier to the object before save() is called

Increment – generates identifiers of type long, short or int that are unique only when no other process is inserting data into the same table

Identity - supports identity columns in DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL

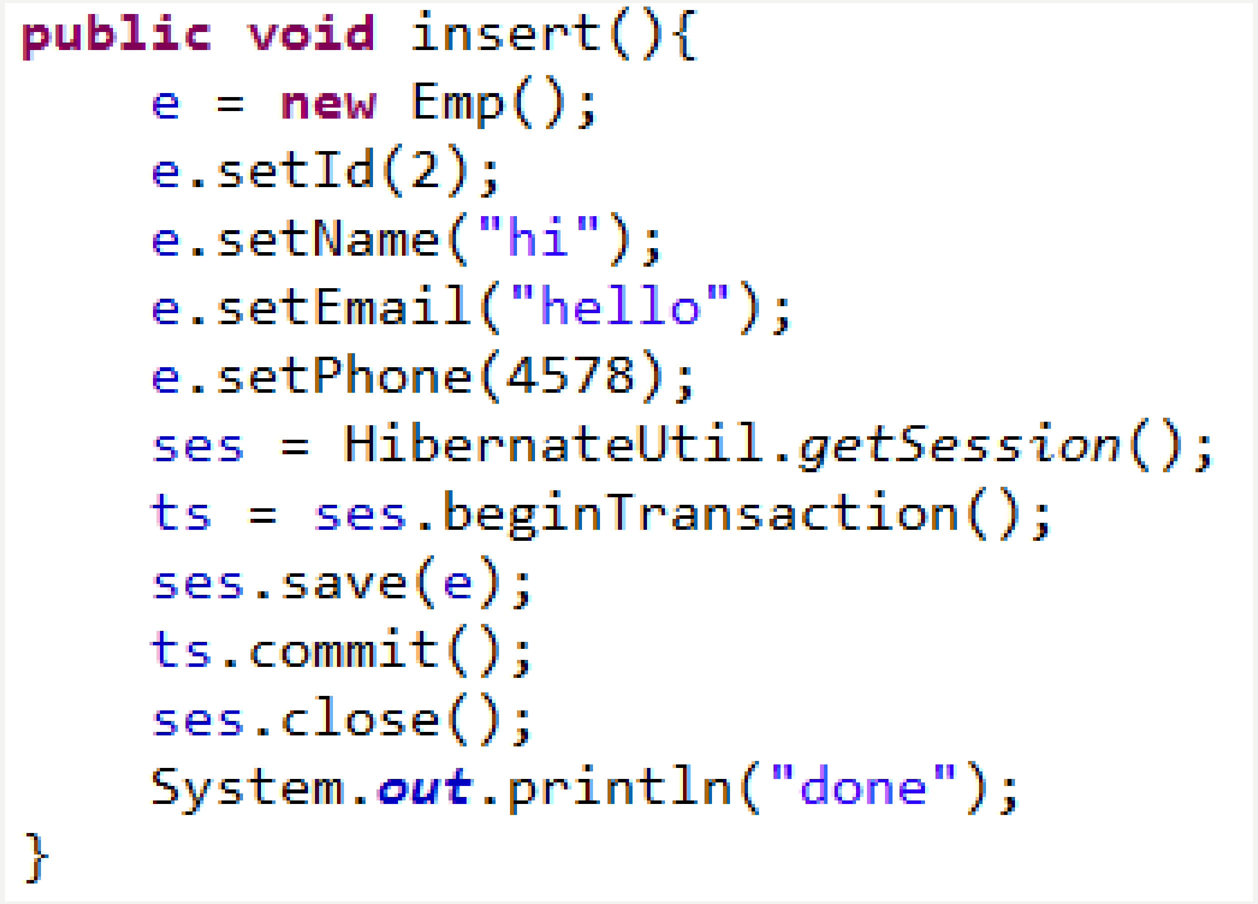
And many more : https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/mapping.html#mapping-declaration-id

MAPPING WITH JPA {JAVA PERSISTENCE ANNOTATIONS}

- @Entity declares the class as an entity (i.e. a persistent POJO class)
- @Id declares the identifier property of this entity
- @Table is set at the class level; it allows you to define the table, catalog, and schema names for your entity mapping
- @column is used for a property mapping can be defined

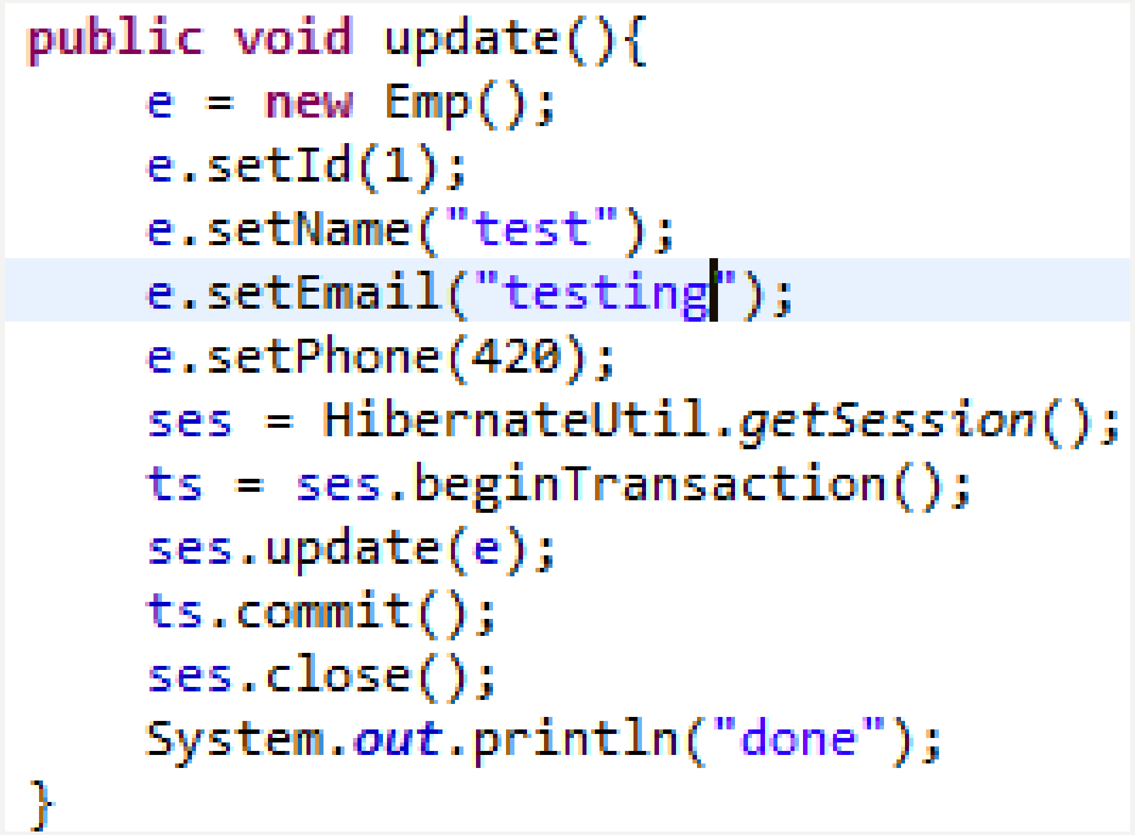
```
@Entity
public class Employee
{
                long id;

                @Id
                public Long getId() { return id; }

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

SAVING OBJECTS USING HIBERNATE APIS



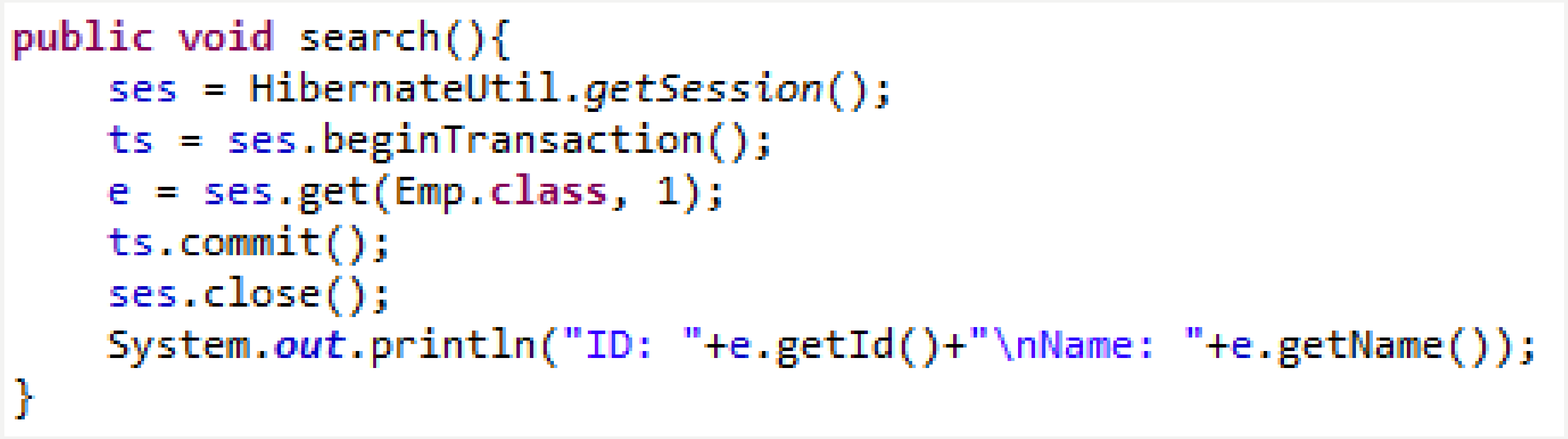
```
public void insert(){
    e = new Emp();
    e.setId(2);
    e.setName("hi");
    e.setEmail("hello");
    e.setPhone(4578);
    ses = HibernateUtil.getSession();
    ts = ses.beginTransaction();
    ses.save(e);
    ts.commit();
    ses.close();
    System.out.println("done");
}
```

UPDATING OBJECTS USING HIBERNATE APIS

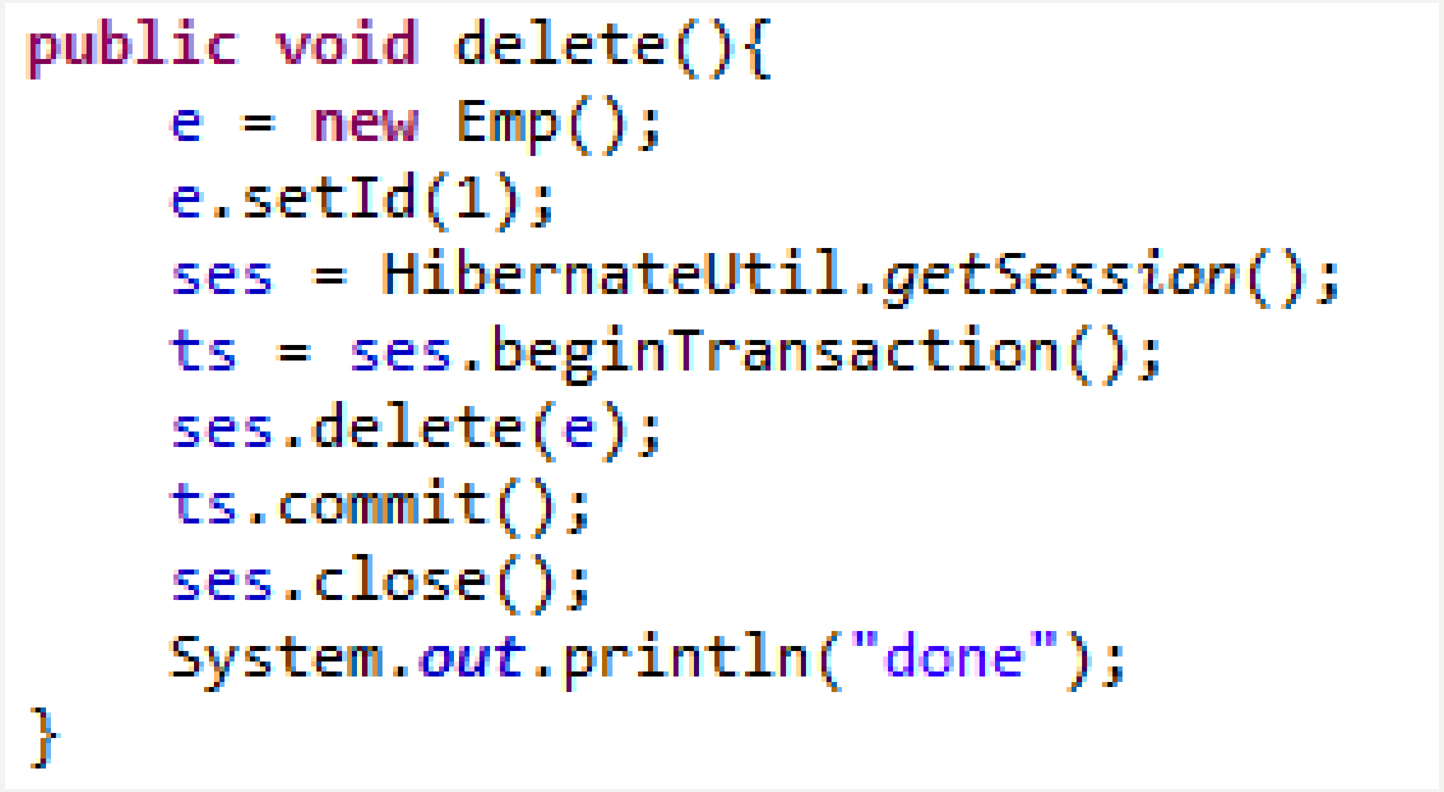


```
public void update(){
    e = new Emp();
    e.setId(1);
    e.setName("test");
    e.setEmail("testing");
    e.setPhone(420);
    ses = HibernateUtil.getSession();
    ts = ses.beginTransaction();
    ses.update(e);
    ts.commit();
    ses.close();
    System.out.println("done");
}
```

SEARCHING OBJECT USING HIBERNATE APIS


```
public void search(){
    ses = HibernateUtil.getSession();
    ts = ses.beginTransaction();
    e = ses.get(Emp.class, 1);
    ts.commit();
    ses.close();
    System.out.println("ID: "+e.getId()+"\nName: "+e.getName());
}
```

DELETING OBJECT USING HIBERNATE APIS

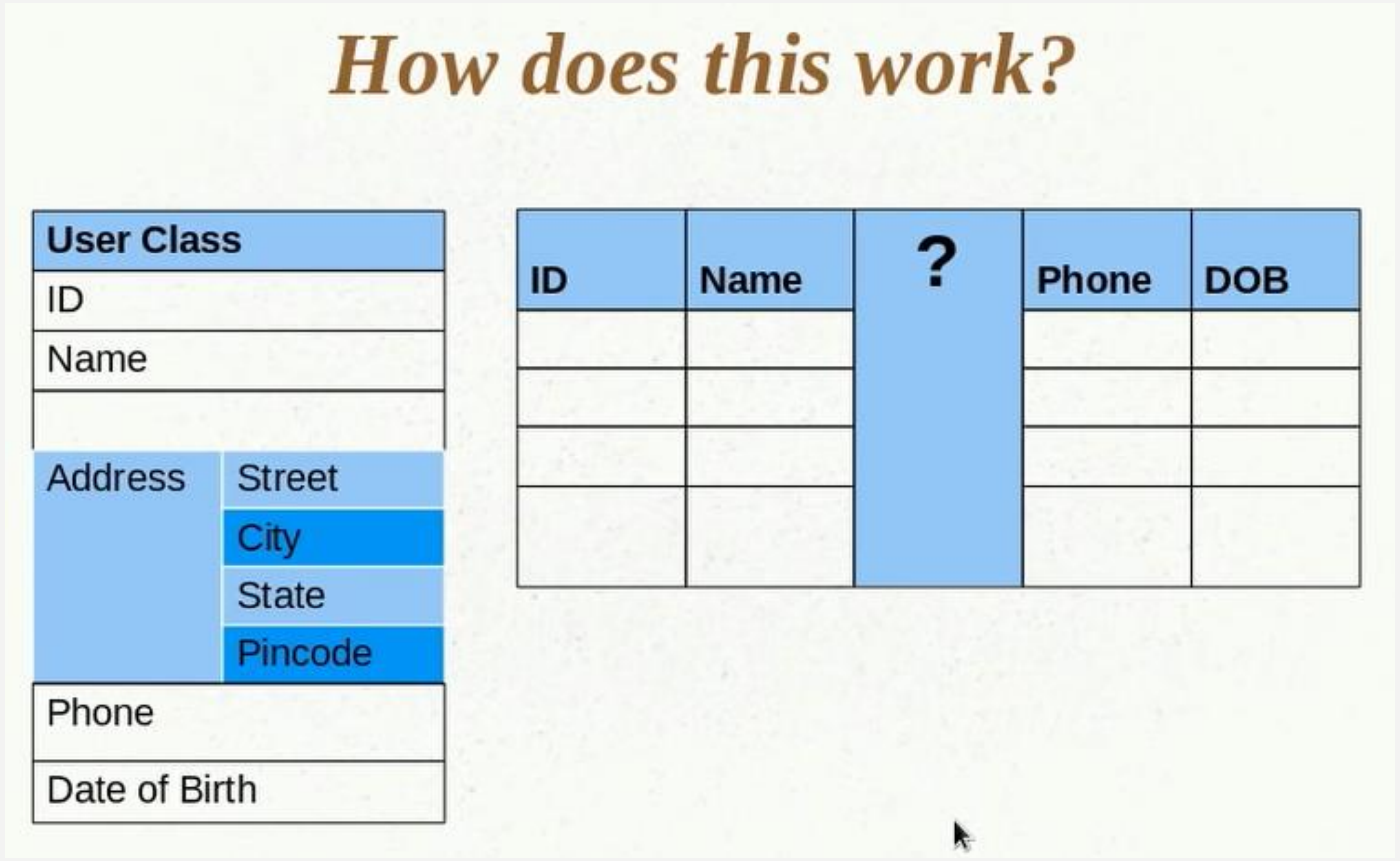


```
public void delete(){
    e = new Emp();
    e.setId(1);
    ses = HibernateUtil.getSession();
    ts = ses.beginTransaction();
    ses.delete(e);
    ts.commit();
    ses.close();
    System.out.println("done");
}
```

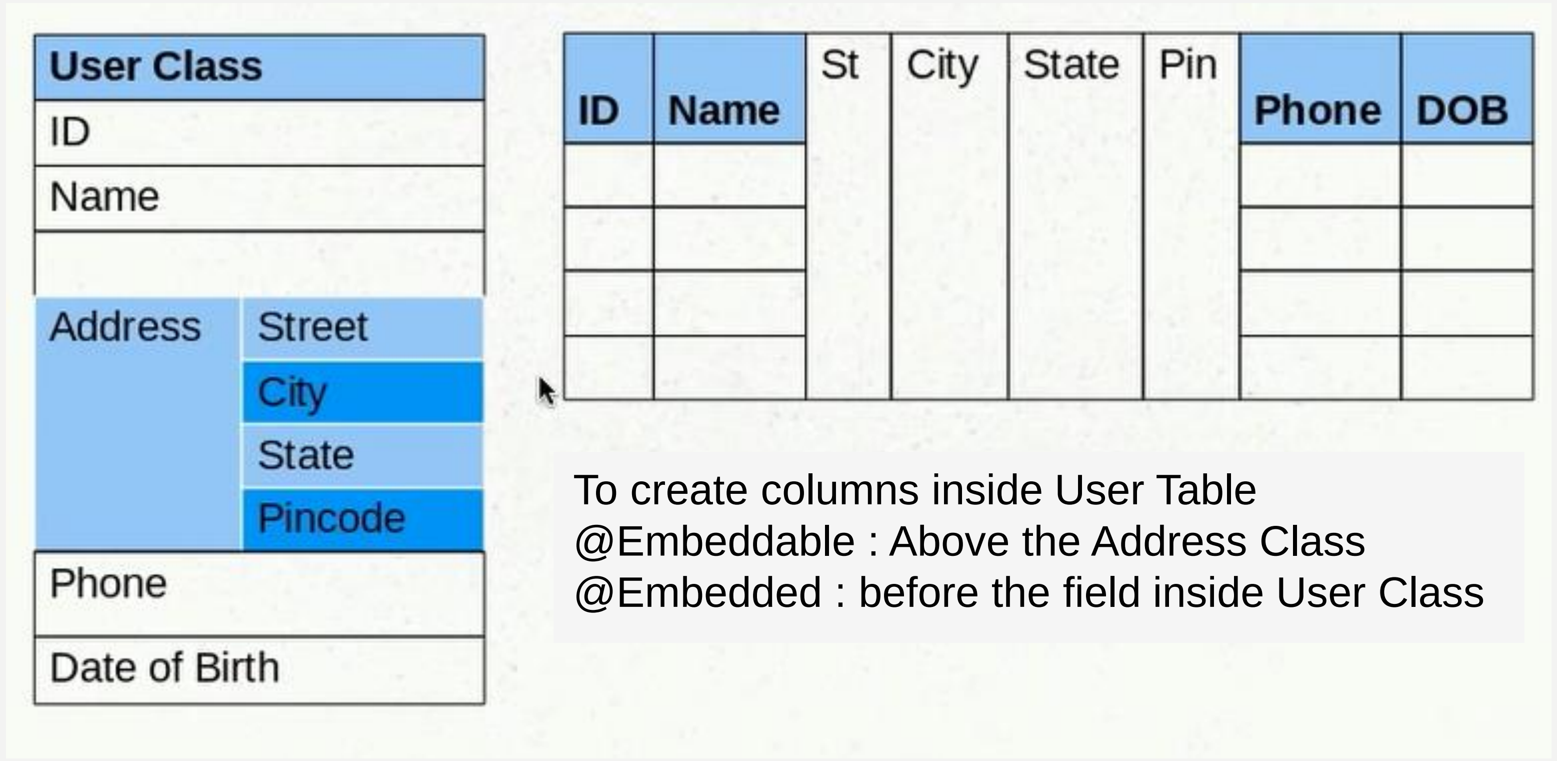
MORE ANNOTATIONS

- @Entity (name="TableName")
- @Column(name="ColumnName") – we can use @column annotation before private field or getter method
- @Table – This annotation will help to differentiate TableName and Entity and it will beneficial in HQL (Hibernate Query Language)
- @Transient – if don't want to store some field or declare that field static
- @Temporal(Date) – if want to store date type field and don't want time
- @Temporal(Time) – if want to store time only not date
- @Lob – if want to store large objects
- @GeneratedValue – hibernate generate value like auto increment
- @GeneratedValue (strategy={auto, identity, sequence, table,…})

EMBEDDING OBJECTS



ONE APPROACH-SEPARATE COLUMNS



To create columns inside User Table
@Embeddable : Above the Address Class
@Embedded : before the field inside User Class

- In order to mark value tag not create separate table

```
@Embeddable
Class Address{ ….}

class UserDetail
{
        @Embedded
        Address address;
}
```

MAPPING COLUMN OF EMBEDDED OBJECT

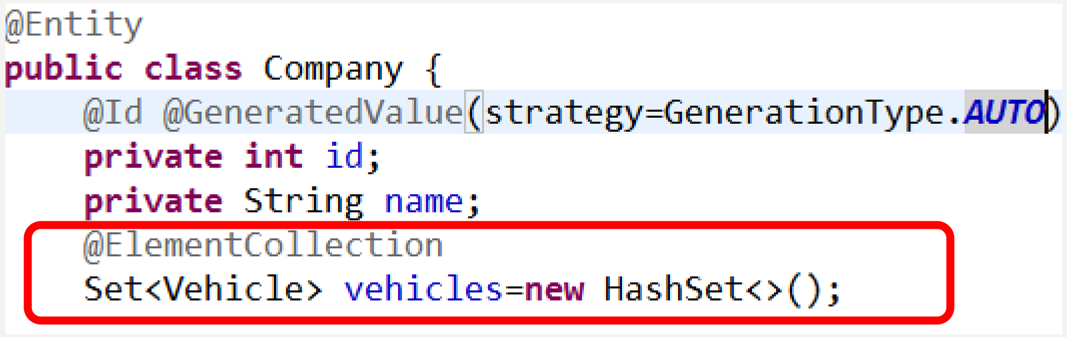
- Inside address class @column before the fields
- What happens if we have two address (1) Home Address (2) Office Address
- Solution – **Override Attribute**

```
@attributeOverrides({
        @attributeOverride(name="street", column=@column(name="h_street")),
        @attributeOverride(name="city", column=@column(name="h_city")),
        @attributeOverride(name="state", column=@column(name="h_state")),
        @attributeOverride(name="pincode", column=@column(name="h_pincode")),
})
```

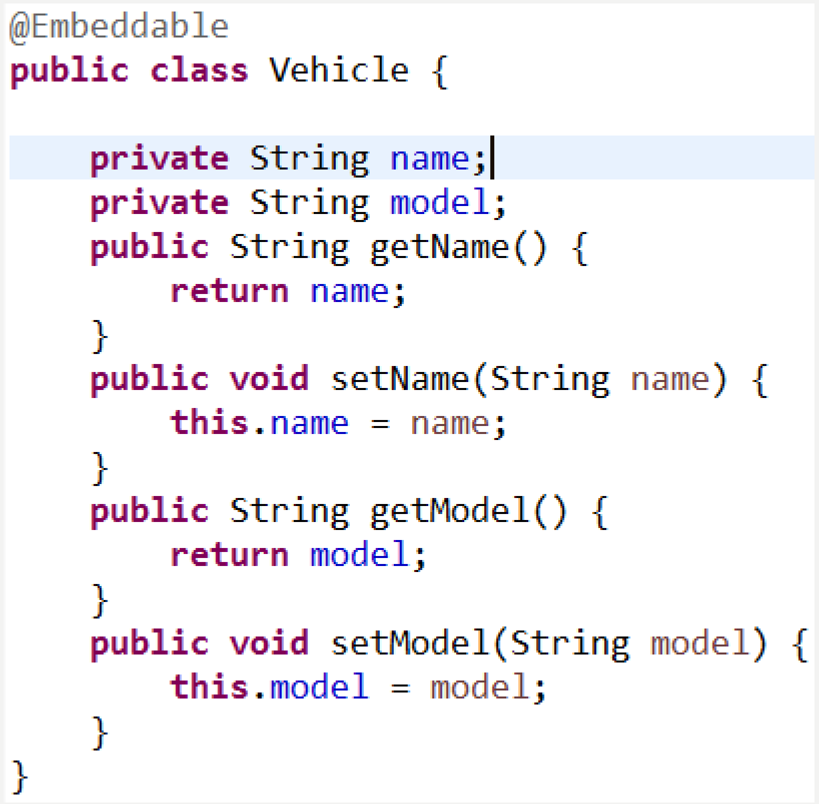
COLLECTION MAPPING

- if I don't know how many address user is going to have then
- Type of collection in Persistent class available are :
  1. java.util.List
  2. java.util.Set
  3. java.util.SortedSet
  4. java.util.Map
  5. java.util.SortedMap
  6. java.util.Collection
  7. or write the implementation of org.hibernate.usertype.UserCollectionType

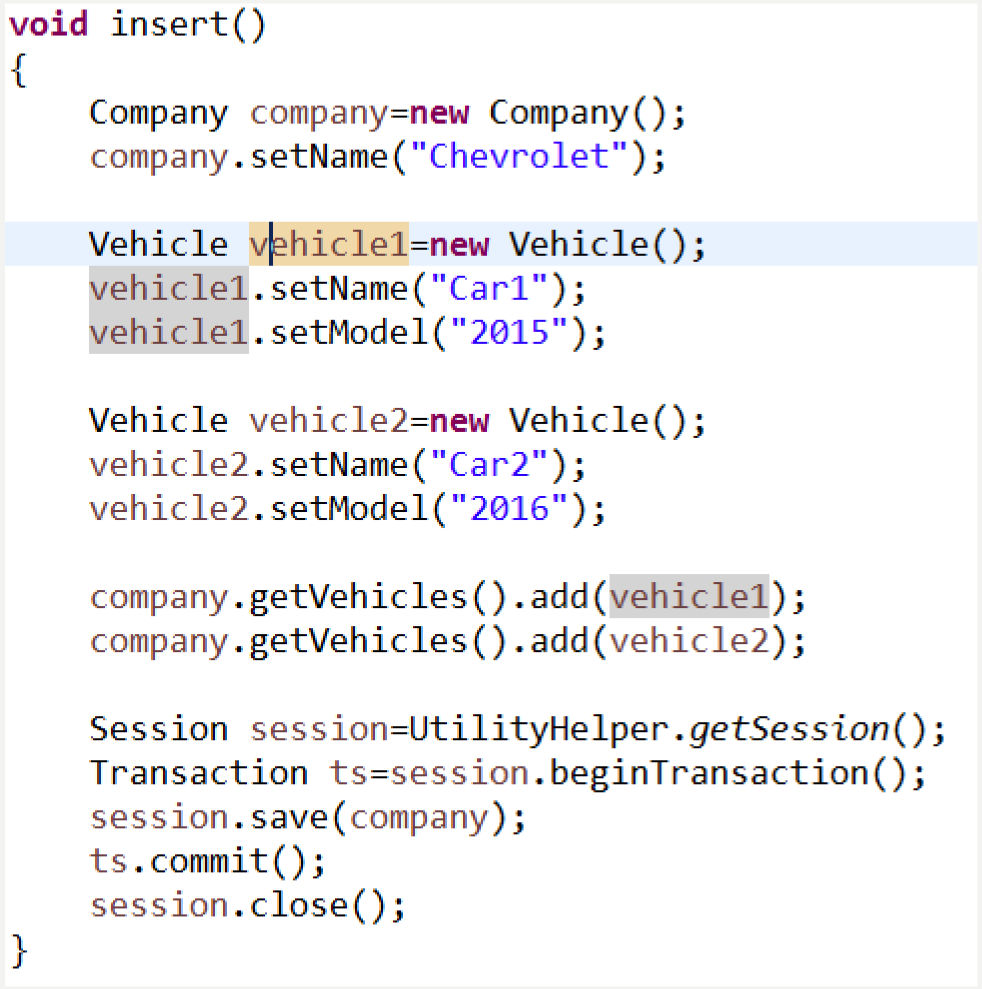
EXAMPLE



```java
@Entity
public class Company {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    private String name;
    @ElementCollection
    Set<Vehicle> vehicles=new HashSet<>();
```

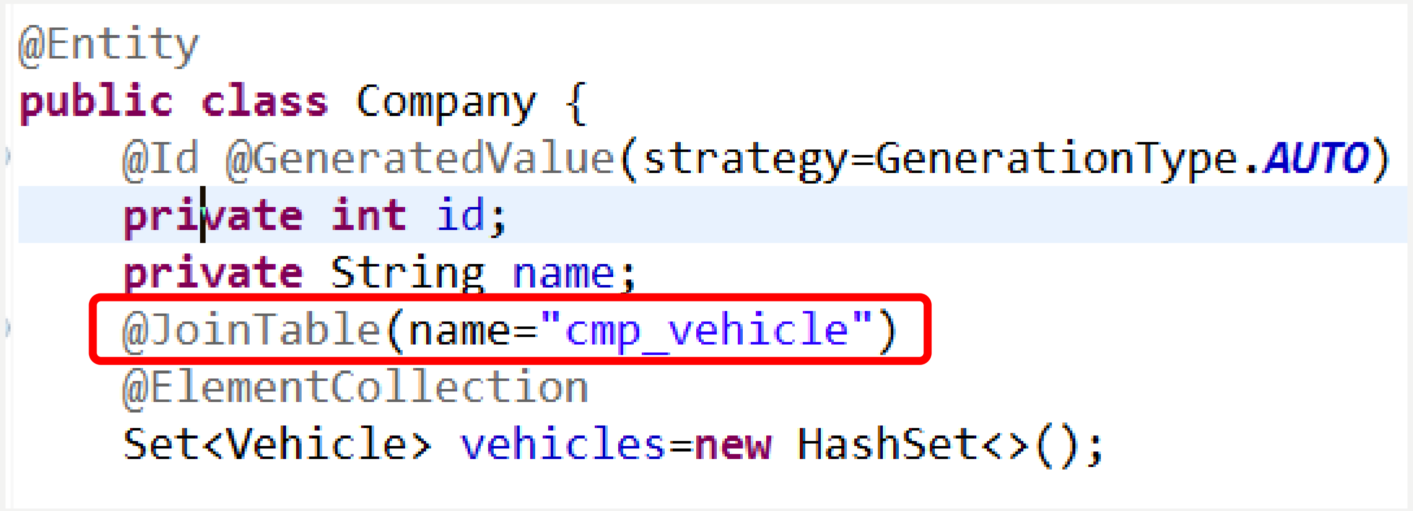


```java
@Embeddable
public class Vehicle {

    private String name;
    private String model;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getModel() {
        return model;
    }
    public void setModel(String model) {
        this.model = model;
    }
}
```



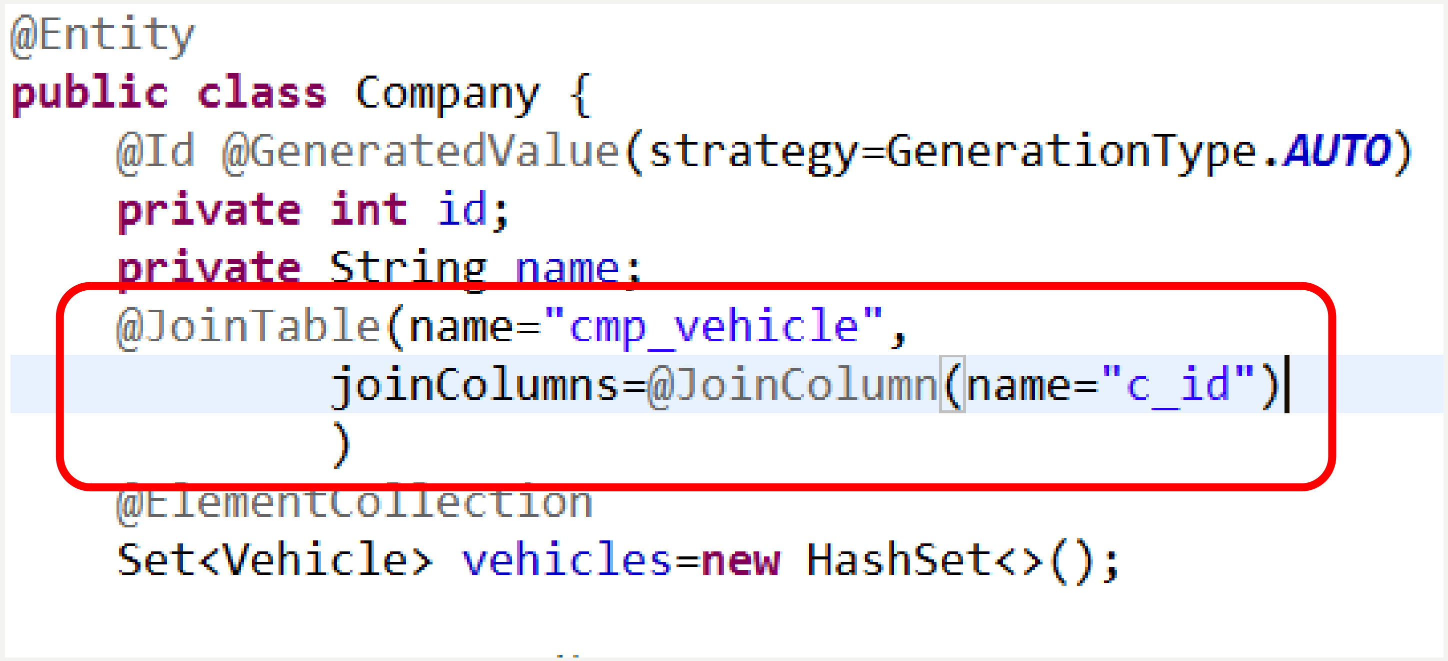
```java
void insert()
{
    Company company=new Company();
    company.setName("Chevrolet");

    Vehicle vehicle1=new Vehicle();
    vehicle1.setName("Car1");
    vehicle1.setModel("2015");

    Vehicle vehicle2=new Vehicle();
    vehicle2.setName("Car2");
    vehicle2.setModel("2016");

    company.getVehicles().add(vehicle1);
    company.getVehicles().add(vehicle2);

    Session session=UtilityHelper.getSession();
    Transaction ts=session.beginTransaction();
    session.save(company);
    ts.commit();
    session.close();
}
```

- To Create table with meaningful name



```
@Entity
public class Company {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    private String name;
    @JoinTable(name="cmp_vehicle")
    @ElementCollection
    Set<Vehicle> vehicles=new HashSet<>();
```

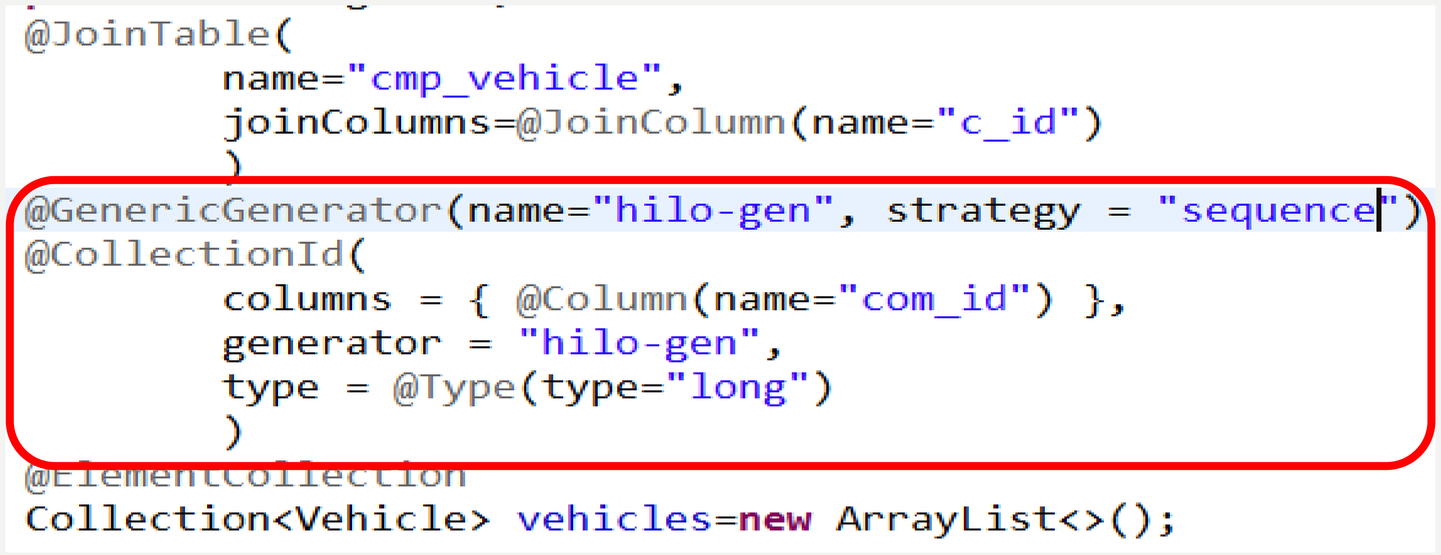
- To Create Column of Company with meaningful name



```
@Entity
public class Company {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    private String name;
    @JoinTable(name="cmp_vehicle",
            joinColumns=@JoinColumn(name="c_id")
            )
    @ElementCollection
    Set<Vehicle> vehicles=new HashSet<>();
```

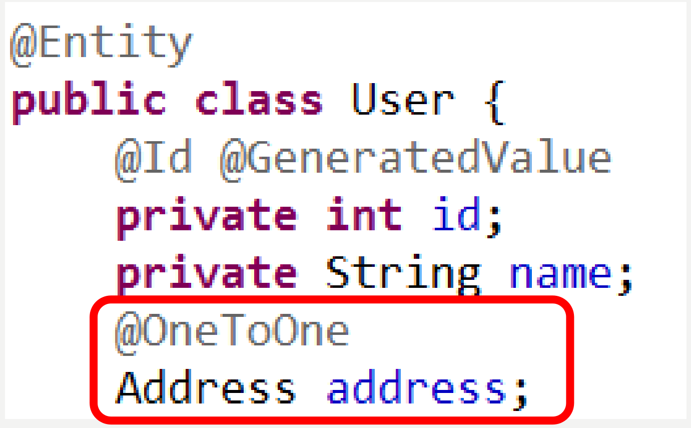
ID PRIMARY KEY IN NEW TABLE

- @CollectionId → org.hibernate.annotations.collectionid
- collection id provided by hibernate alone not persistance
- hibernate also implement JPA so no need to worry
- ArrayList support indexes so we have to use it if we want primary key inside **Vehicle**

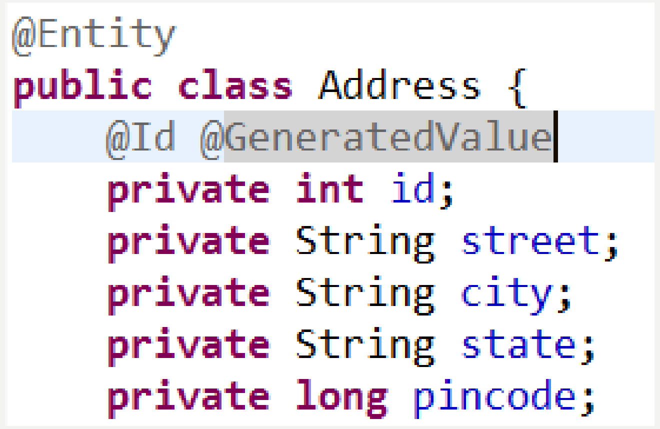


```
@JoinTable(
        name="cmp_vehicle",
        joinColumns=@JoinColumn(name="c_id")
        )
@GenericGenerator(name="hilo-gen", strategy = "sequence")
@CollectionId(
        columns = { @Column(name="com_id") },
        generator = "hilo-gen",
        type = @Type(type="long")
        )
@ElementCollection
Collection<Vehicle> vehicles=new ArrayList<>();
```

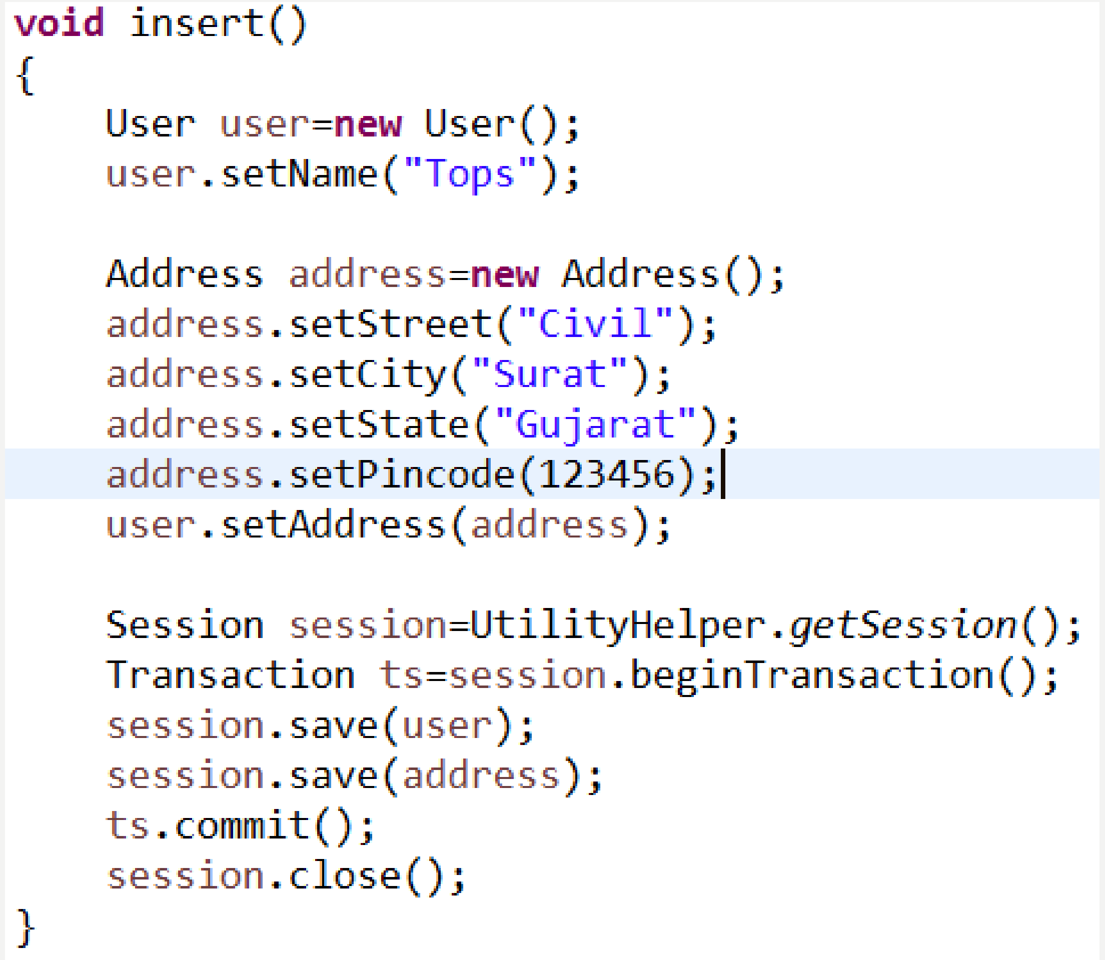
ONE TO ONE MAPPING



```
@Entity
public class User {
    @Id @GeneratedValue
    private int id;
    private String name;
    @OneToOne
    Address address;
```

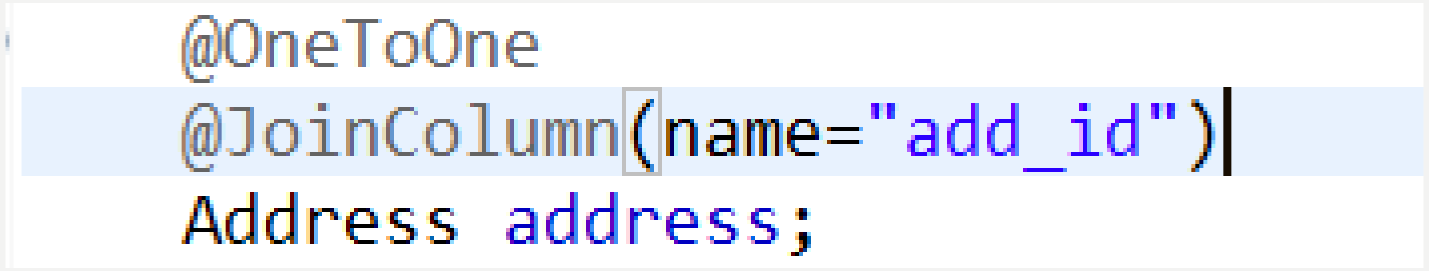


```
@Entity
public class Address {
    @Id @GeneratedValue
    private int id;
    private String street;
    private String city;
    private String state;
    private long pincode;
```



```
void insert()
{
    User user=new User();
    user.setName("Tops");

    Address address=new Address();
    address.setStreet("Civil");
    address.setCity("Surat");
    address.setState("Gujarat");
    address.setPincode(123456);
    user.setAddress(address);

    Session session=UtilityHelper.getSession();
    Transaction ts=session.beginTransaction();
    session.save(user);
    session.save(address);
    ts.commit();
    session.close();
}
```

CHANGE JOIN COLUMN NAME



```
@OneToOne
@JoinColumn(name="add_id")
Address address;
```

HQL

- Hibernate uses a powerful query language (HQL) that is similar in appearance to SQL.
- Compared with SQL, however, HQL is fully object-oriented and understands notions like inheritance, polymorphism and association.
- With the exception of names of Java classes and properties, queries are case-insensitive.

QUERY INTERFACE

- The object of Query can be obtained by calling the createQuery() method Session interface.
- query interface provides many methods
  1. **public int executeUpdate()** is used to execute the update or delete query.
  2. **public List list()** returns the result of the ralation as a list.
  3. **public Query setFirstResult(int rowno)** specifies the row number from where record will be retrieved.
  4. **public Query setMaxResult(int rowno)** specifies the no. of records to be retrieved from the relation (table).
  5. **public Query setParameter(int position, Object value)** it sets the value to the JDBC style query parameter.
  6. **public Query setParameter(String name, Object value)** it sets the value to a named query parameter.

THE FROM CLAUSE

- The simplest possible form :
  - List<Emp> ls = ses.createQuery("From Emp").list();
- In order to refer to the Cat in other parts of the query, you will need to assign an alias
  - from Cat as cat
- Multiple classes can appear, resulting in a cartesian product or "cross" join.
  - from Formula, Parameter
  - from Formula as form, Parameter as param

GET RECORDS WITH PAGINATION

```
Query query=session.createQuery("from Emp");
query.setFirstResult(5);
query.setMaxResult(10);
List list=query.list();//will return the records from 5 to 10th number
```

UPDATE QUERY

```
Transaction tx=session.beginTransaction();
Query q=session.createQuery("update User set name=:n where id=:i");
q.setParameter("n","Udit Kumar");
q.setParameter("i",111);
int status=q.executeUpdate();
System.out.println(status);
tx.commit();
```

DELETE QUERY

```
Query query=session.createQuery("delete from Emp where id=100");
//specifying class name (Emp) not tablename
query.executeUpdate();
```

AGGREGATE FUNCTIONS

- You may call avg(), min(), max() etc. aggregate functions by HQL.

```
Query q=session.createQuery("select sum(salary) from Emp");
List<Integer> list=q.list();
System.out.println(list.get(0));


Query q=session.createQuery("select max(salary) from Emp");
```

HIBERNATE CRITERIA QUERY LANGUAGE

- The Criteria interface provides methods to apply criteria such as retreiving all the records of table whose salary is greater than 50000 etc
- The HCQL provides methods to add criteria, so it is **easy** for the java programmer to add criteria. The java programmer is able to add many criteria on a query
- The object of Criteria can be obtained by calling the **createCriteria()** method of Session interface.

RESTRICTIONS CLASS

- **public static SimpleExpression lt(String propertyName,Object value)** sets the **less than** constraint to the given property.
- **public static SimpleExpression le(String propertyName,Object value)** sets the **less than or equal** constraint to the given property.
- **public static SimpleExpression gt(String propertyName,Object value)** sets the **greater than** constraint to the given property.
- **public static SimpleExpression ge(String propertyName,Object value)** sets the **greater than or equal** than constraint to the given property.
- **public static SimpleExpression ne(String propertyName,Object value)** sets the **not equal** constraint to the given property.
- **public static SimpleExpression eq(String propertyName,Object value)** sets the **equal** constraint to the given property.
- **public static Criterion between(String propertyName, Object low, Object high)** sets the **between** constraint.
- **public static SimpleExpression like(String propertyName, Object value)** sets the **like** constraint to the given property.

ORDER CLASS

- **public static Order asc(String propertyName)** applies the ascending order on the basis of given property.
- **public static Order desc(String propertyName)** applies the descending order on the basis of given property.

GET ALL THE RECORDS

```
Crietria c=session.createCriteria(Emp.class);//passing Class class argument
List list=c.list();

Crietria c=session.createCriteria(Emp.class);
c.setFirstResult(10);
c.setMaxResult(20);
List list=c.list();
```

GET THE RECORDS WHOSE SALARY IS GREATER THAN 10000

```
Crietria c=session.createCriteria(Emp.class);
c.add(Restrictions.gt("salary",10000));//salary is the propertyname
List list=c.list();
```

GET THE RECORDS IN ASCENDING ORDER ON THE BASIS OF SALARY

```
Crietria c=session.createCriteria(Emp.class);
c.addOrder(Order.asc("salary"));
List list=c.list();
```

HCQL WITH PROJECTION

- We can fetch data of a particular column by projection such as name etc. Let's see the simple example of projection that prints data of NAME column of the table only.

```
Criteria c=session.createCriteria(Emp.class);
c.setProjection(Projections.property("name"));
List list=c.list();
```

ASSOCIATIONS AND JOINS

- You can also assign aliases to associated entities or to elements of a collection of values using a join. For example:
  - from Cat as cat inner join cat.mate as mate left outer join cat.kittens as kitten
  - from Cat as cat left join cat.mate.kittens as kittens
  - from Formula form full join form.parameter param

FORMS OF JOIN SYNTAX

- HQL supports two forms of association joining: implicit and explicit.

- The queries shown in the previous section all use the explicit form, that is, where the join keyword is explicitly used in the from clause. This is the recommended form.

- The implicit form does not use the join keyword. Instead, the associations are "dereferenced" using dot-notation. implicit joins can appear in any of the HQL clauses. implicit join result in inner joins in the resulting SQL statement.
  - from Cat as cat where cat.mate.name like '%s%'

THE SELECT CLAUSE

- The select clause picks which objects and properties to return in the query result set. Consider the following
  - select mate from Cat as cat inner join cat.mate as mate

AGGREGATE FUNCTIONS

- HQL queries can even return the results of aggregate functions on properties:
  - select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat)

    from Cat cat

THE GROUP BY CLAUSE

- A query that returns aggregate values can be grouped by any property of a returned class or components:
  - **select cat.color, sum(cat.weight), count(cat)**

    from Cat cat

    group by cat.color
- A having clause is also allowed.
  - **select cat.color, sum(cat.weight), count(cat)**

    from Cat cat

    group by cat.color

    having cat.color in (eg.Color.TABBY, eg.Color.BLACK)

SUBQUERIES

- For databases that support subselects, Hibernate supports subqueries within queries. A subquery must be surrounded by parentheses (often by an SQL aggregate function call). Even correlated subqueries (subqueries that refer to an alias in the outer query) are allowed
  - **from Cat as fatcat**

    where fatcat.weight > (

    select avg(cat.weight) from DomesticCat cat

)

    - OR -
  - **select cat.id, (select max(kit.weight) from cat.kitten kit)**

    from Cat as cat

TIPS & TRICKS

- You can count the number of query results without returning them
  - **((Integer) session.createQuery("select count(*) from ....").iterate().next()).intValue()**