



**MODULE- 10**  
**HIBERNATE**

# Course Topics

→ Module 1

» Introduction to Java

→ Module 2

» Data Handling and Functions

→ Module 3

» Object Oriented Programming in Java

→ Module 4

» Packages and Multi-threading

→ Module 5

» Collections

→ Module 6

» XML

→ Module 7

» JDBC

→ Module 8

» Servlets

→ Module 9

» JSP

→ Module 10

» **Hibernate**

→ Module 11

» Spring

→ Module 12

» Spring, Ajax and Design Patterns

→ Module 13

» SOA

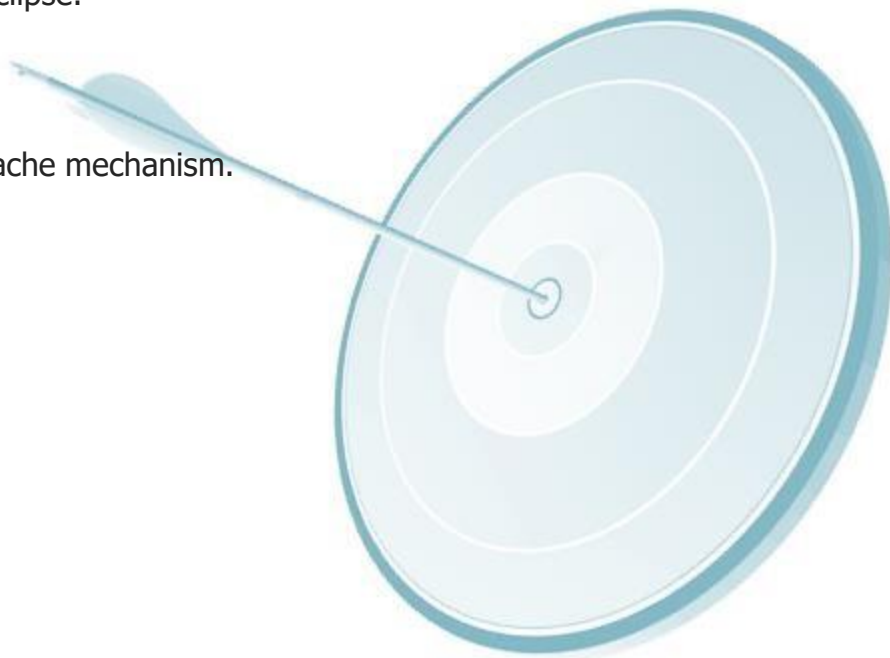
→ Module 14

» Web Services and Project

# Objectives

At the end of this module, you will be able to

- Understand Hibernate and how to use hibernate in eclipse.
- Create an application using Hibernate
- Map tables using Hibernate
- Understand inheritance in hibernate and hibernate cache mechanism.

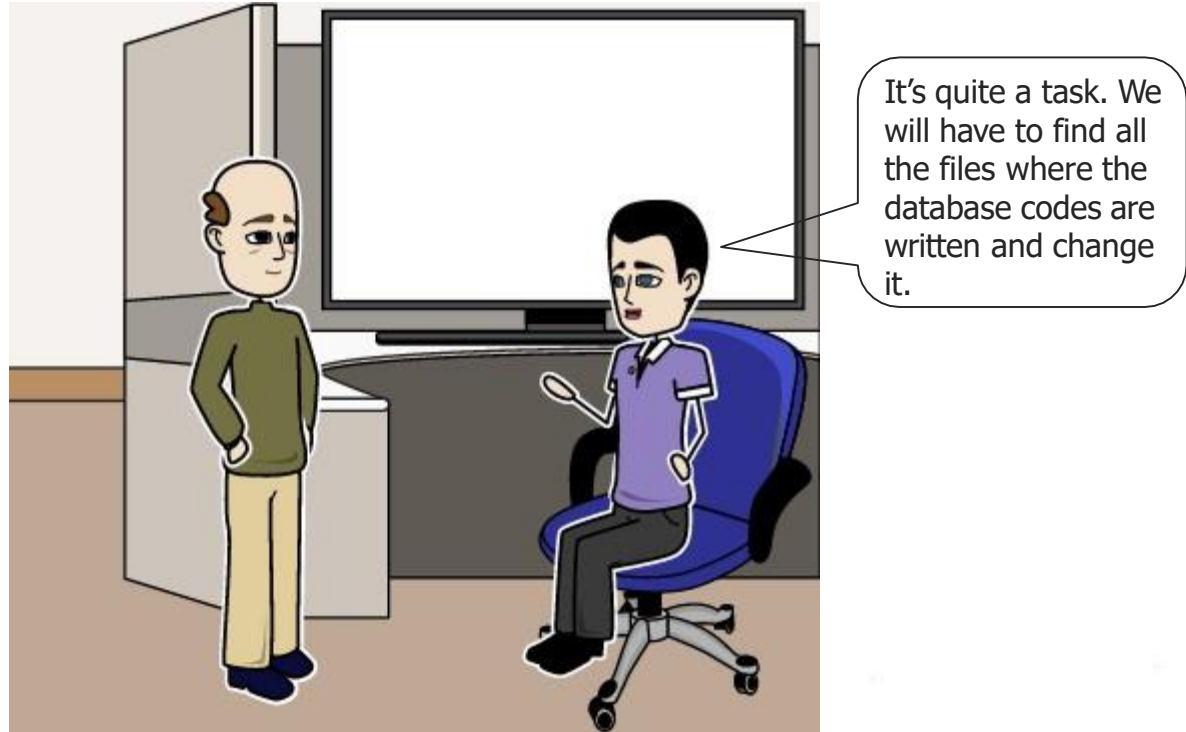


# John gets a Project

John we want to shift our database from Oracle to MySQL. How can we go about it?



# Switching Database!

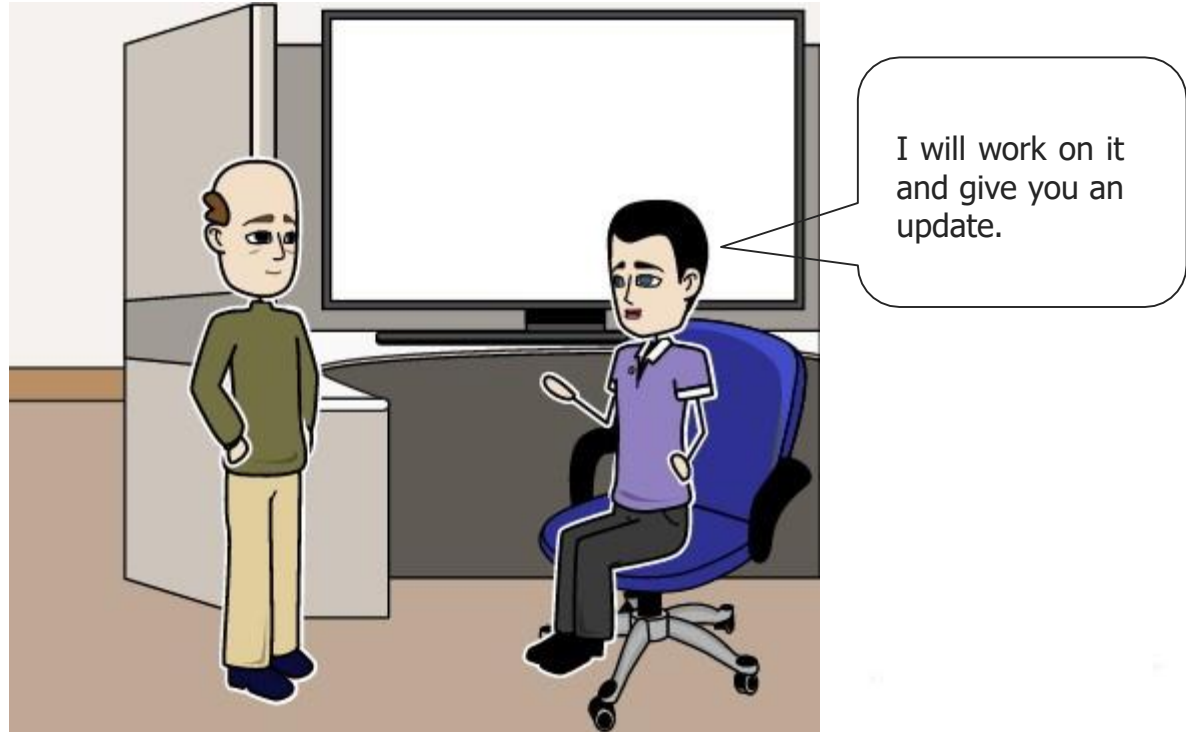


# Switching Database!

That is a very long process...  
Do we have a good and easy way of dealing with this?



# Switching Database!



# Switching Database!

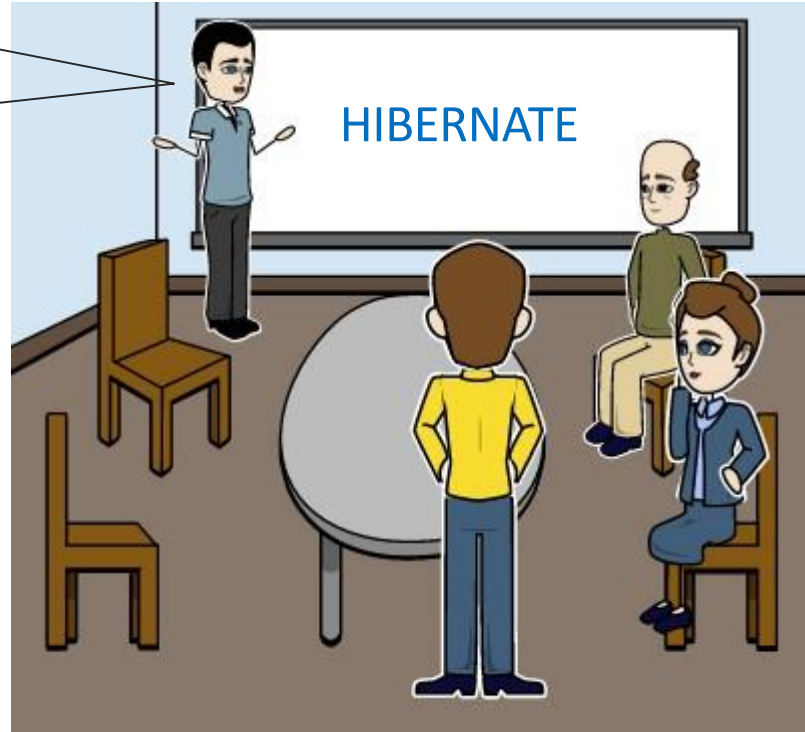
Let us have a brief discussion with Mark and Daisy on this. Let's come up with a good solution.



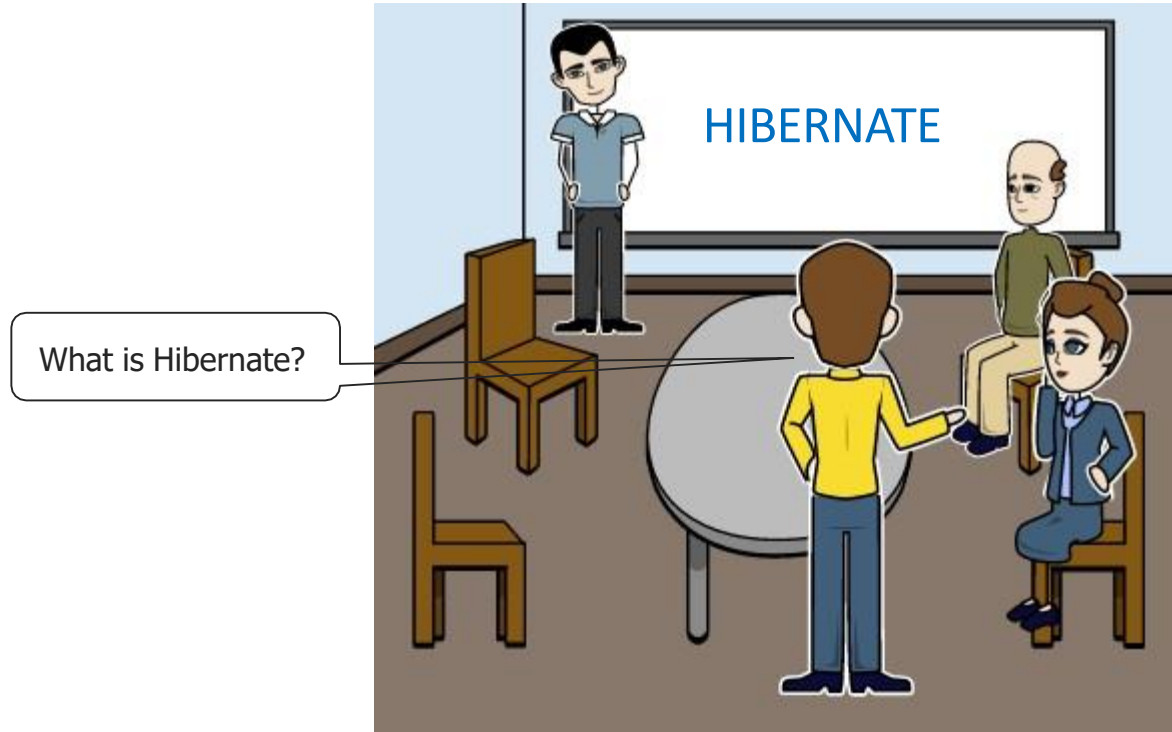


# Simply Hibernate...

We could use  
"Hibernate" to  
change our  
database from  
Oracle to MySQL.

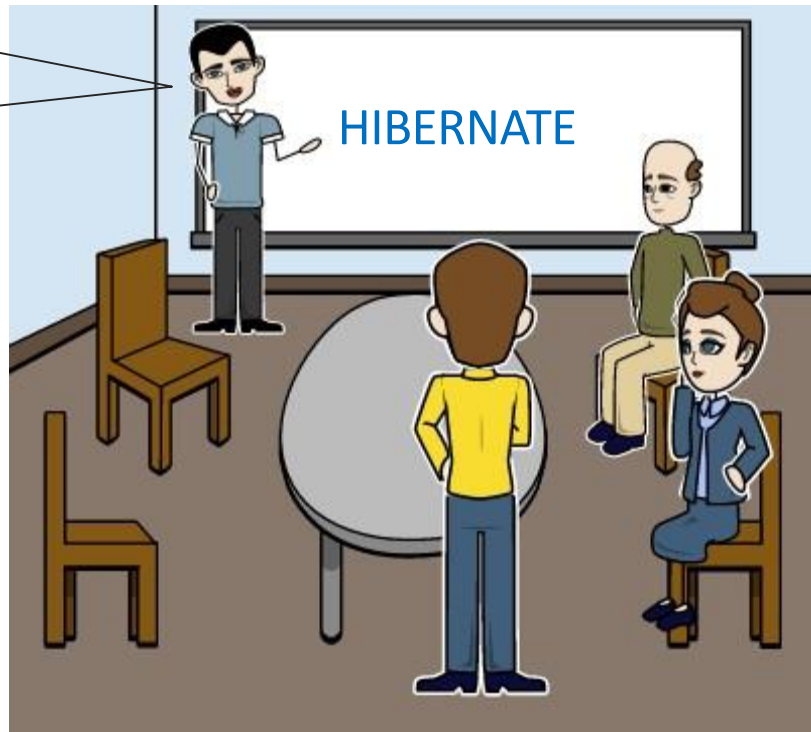


# Simply Hibernate...



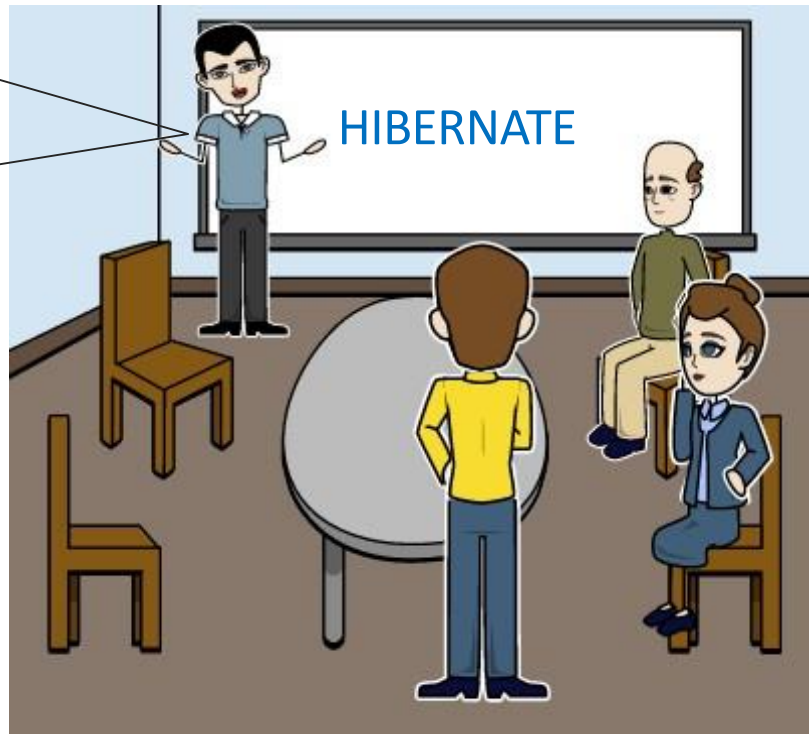
# Simply Hibernate...

Hibernate is a framework to map your Java classes into database tables.



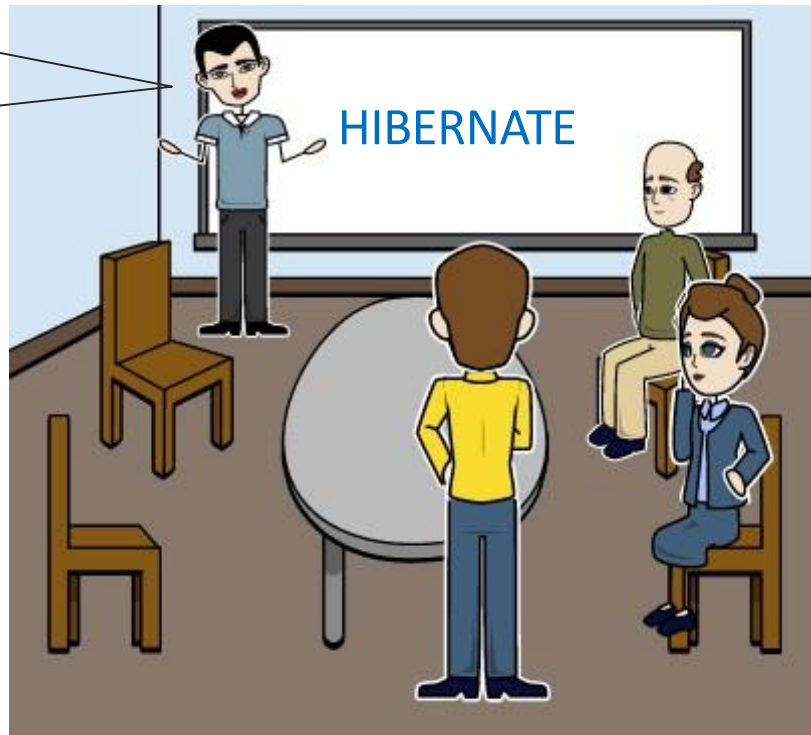
# Simply Hibernate...

Hibernate provides xml configuration file named hibernate.cfg.xml, so if we have to make any changes related to database, we just need to edit a single file.



# Simply Hibernate...

We can use hibernate for functions such as insert, update, delete and retrieve.



Let's Learn More.....

# Hibernate - Introduction

- Hibernate is an ORM (Object Relationship Mapping) framework.
- Hibernate ORM (Hibernate in short) is an object-relational mapping library for the Java language, providing a framework for mapping an object-oriented domain model to a traditional relational database.
- Hibernate is free.
- Hibernate generates the SQL automatically. User need not write the SQL.
- Hibernate reduces the development time.

# Difference between JDBC and HIBERNATE

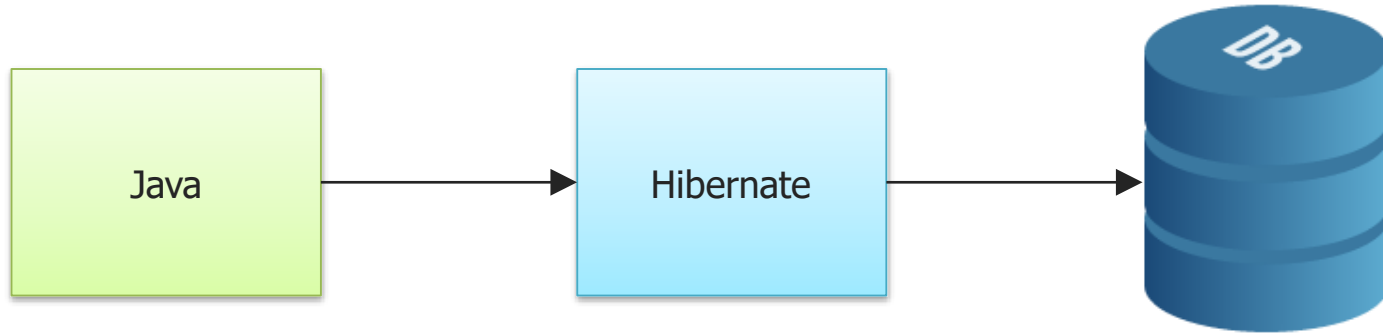
JDBC	HIBERNATE
SQL queries are database specific.	Hibernate is database independent.
implementing Java cache is required.	Hibernate provides three level of caches.
If we use JDBC , then switching databases is tough.	Change in database can be done by editing a single file.
Developers have to write codes to map tables and Java objects.	Hibernate provides an easy way to map our Java objects into database tables.

# Hibernate - ORM

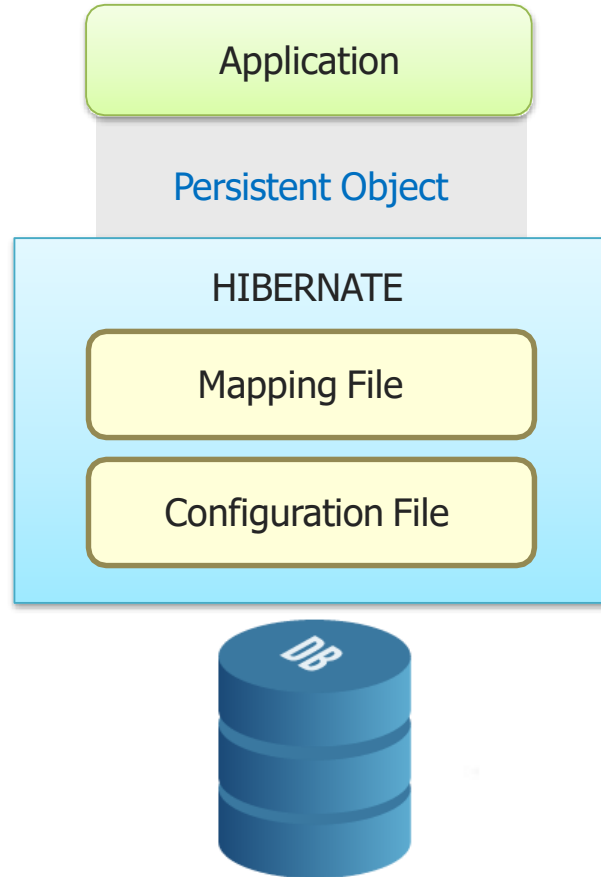
- Business objects can directly access data objects instead of tables. Helpful in DAO and DTO patterns.
- SQL query is generated by ORM, hence SQL Query is hidden.
- Hibernate is developed on top of JDBC.
- Need not write DB specific queries.
- Transaction Management like `Commit()` and `Rollback()` are automatically taken care.
- The time required for development is less when compared to JDBC.



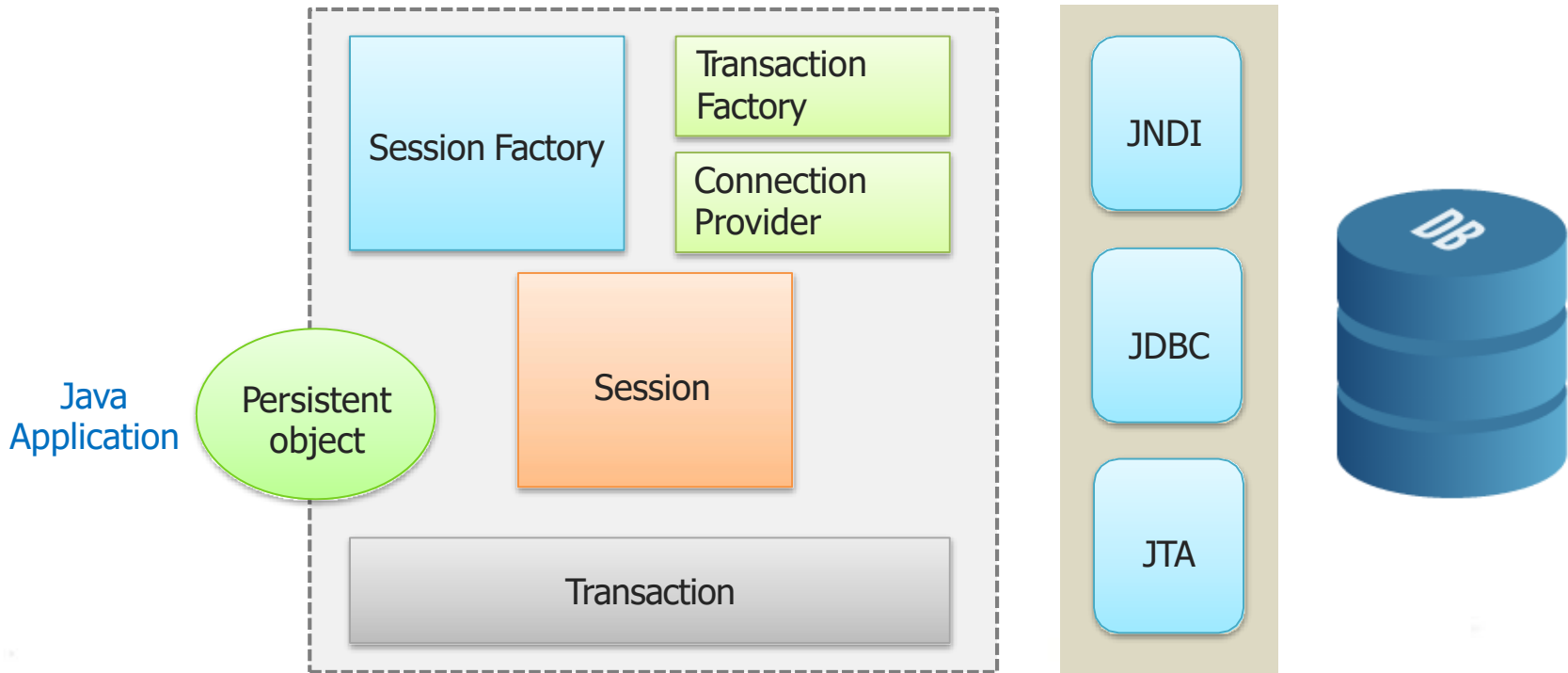
# Connecting to DB using Hibernate



# Hibernate Architecture



# Hibernate Architecture (contd.)



# Hibernate Architecture (contd.)

- [SessionFactory](#) creates session.
- Session is an interface which connects the application and stored data in the database.
- Session provides methods to insert, update and delete data in database.
- Transaction allows the application to define the units of work.
- [ConnectionProvider](#) is a strategy for obtaining JDBC connections.
- The [ConnectionProvider](#) interface is not intended to be exposed to the application. It is used internally by Hibernate to obtain connections.

# Files required in Hibernate

Hibernate configuration file → [hibernate.cfg.xml](#) → This file will have database connection parameter.

Java object which needs to be written in the db. This is a bean class.

This is a Hibernate mapping file → [.hbm file](#). → This file will map the bean class with database.

Client class → This class is to test the hibernate application.

# Hibernate.cfg.xml

This is hibernate configuration file.

This file will have Database configuration parameters like:

- (a) Driver class
- (b) Connection URL
- (c) User name, Password

Dialect is the one which converts the hibernate statements into its corresponding SQL queries of the database.

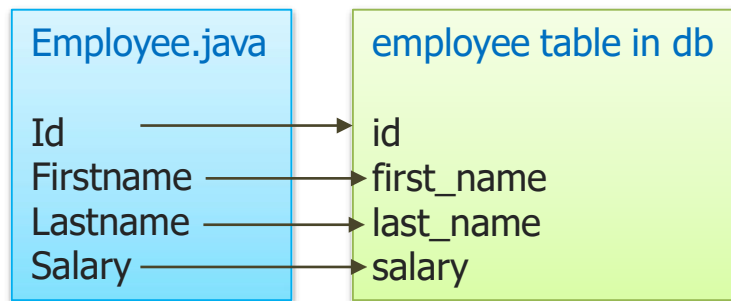
[show\\_sql](#) → will display the generated sql in output console window.

[Mapping resource](#) → will specify the mapping file which maps between class and db table.

You will see how it is written when we do an example in later slides..

# Persistent Class

- A class which has the fields to be mapped with the database.
- This class has constructor with arguments.
- Getter and Setter methods for all the attributes of the class.



# Hbm file

- Hbm file stands for Hibernate mapping.
- This file maps between persistent class and table. All the fields are mapped one to one.

```
<hibernate-mapping>
  <class name="Employee" table="EMPLOYEE">
    <id name="id" type="int" column="id"> </id>
    <property name="firstName" column="first_name" type="string"/>
    <property name="salary" column="salary" type="int"/>
  </class>
</hibernate-mapping>
```

- This file maps employee class with EMPLOYEE table.
- Class attributes id, firstname and salary are mapped with its columns of the EMPLOYEE table respectively.



# Hibernate Annotations

Hibernate annotations are newest way to map between class object and the database table without XML file. For greater flexibility, XML based mapping is preferred.

```
@Entity
@Table(name = "EMPLOYEE")
public class Employee {
    @Id @GeneratedValue
    @Column(name = "id")
    private int id;

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

    @Column(name = "last_name")
    private String lastName;
}
```

Here, rather than xml file mapping, table name can be specified with the annotation @Table and @Id @GeneratedValue, generates the Unique id automatically and column firstName and lastName is mapped with the table columns first\_name and last\_name.

# Hibernate Mapping Types

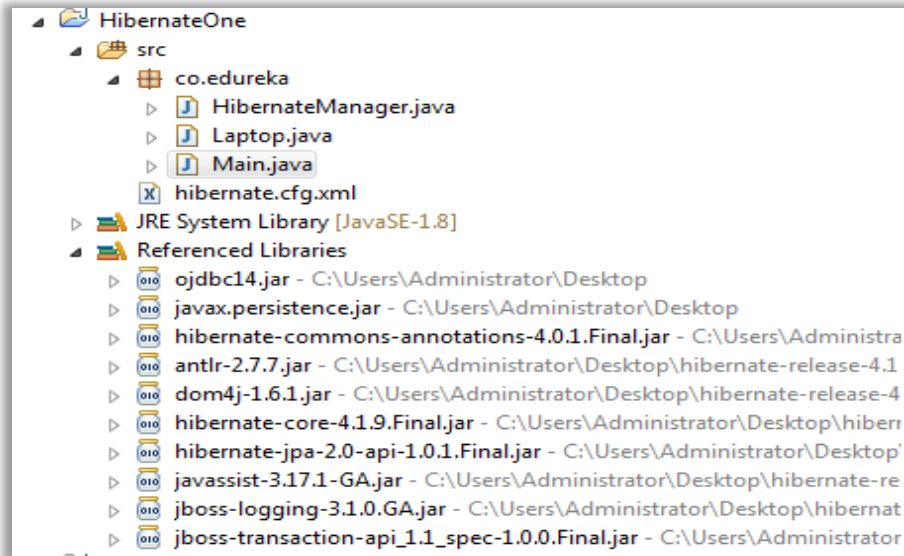
Hibernate mapping types are used to translate between Java and SQL data types.

The Hibernate mapping types are what we use in mapping files neither Java data types nor SQL data types. Following are the few Hibernate mapping type.

Mapping type in hbm file	Data type in java	Table column data type
integer	int or java.lang.Integer	INTEGER
long	long or java.lang.Long	BIGINT
short	short or java.lang.Short	SMALLINT
float	float or java.lang.Float	FLOAT
double	double or java.lang.Double	DOUBLE
string	java.lang.String	VARCHAR

# Configuring Hibernate With Eclipse

- You can download the hibernate related jars from [sourceforge](#) at [Hibernate Jars](#).
- You also have to download [javax.persistence.jar](#) and the driver jar file for the database you are working with. Like If you are working with Oracle database you have to include `ojdbc.jar` file. Your Project structure will look like below snapshot.



# Creating a SessionFactory

In HibernateManager class we are creating `SessionFactory` object that we will need later.

```
package co.edureka;

import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;
import org.hibernate.SessionFactory;
import org.hibernate.service.ServiceRegistryBuilder;

public class HibernateManager {

    private static ServiceRegistry serviceRegistry;
    private static final SessionFactory sessionFactory = buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        try {
            Configuration configuration = new Configuration();
            configuration.configure();

            serviceRegistry = new ServiceRegistryBuilder().applySettings(
                configuration.getProperties()).buildServiceRegistry();

            return configuration.buildSessionFactory(serviceRegistry);

        } catch (Throwable ex) {
            System.err.println("Failed to create sessionFactory object." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

# Creating Java Model Class

Laptop is our Java Model class which we are mapping to a database table LAPTOP by using @Table annotation on the Laptop class.

```
package co.edureka;

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

@Entity
@Table(name="LAPTOP")
public class Laptop {

    @Id
    private int price;
    private String model;
    public Laptop() {}
    public Laptop(String model, int price) {
        this.model = model;
        this.price = price;
    }
    public int getPrice() {
        return price;
    }
    public void setPrice(int price) {
        this.price = price;
    }
    public String getModel() {
        return model;
    }
    public void setModel(String model) {
        this.model = model;
    }
}
```

# Test Class

```
package co.edureka;

import org.hibernate.Session;
import org.hibernate.SessionFactory;

public class Test {

    public static void main(String[] args) {

        Laptop lappy = new Laptop();
        lappy.setModel("Sony Vaio SVF14M16SNS");
        lappy.setPrice(45000);

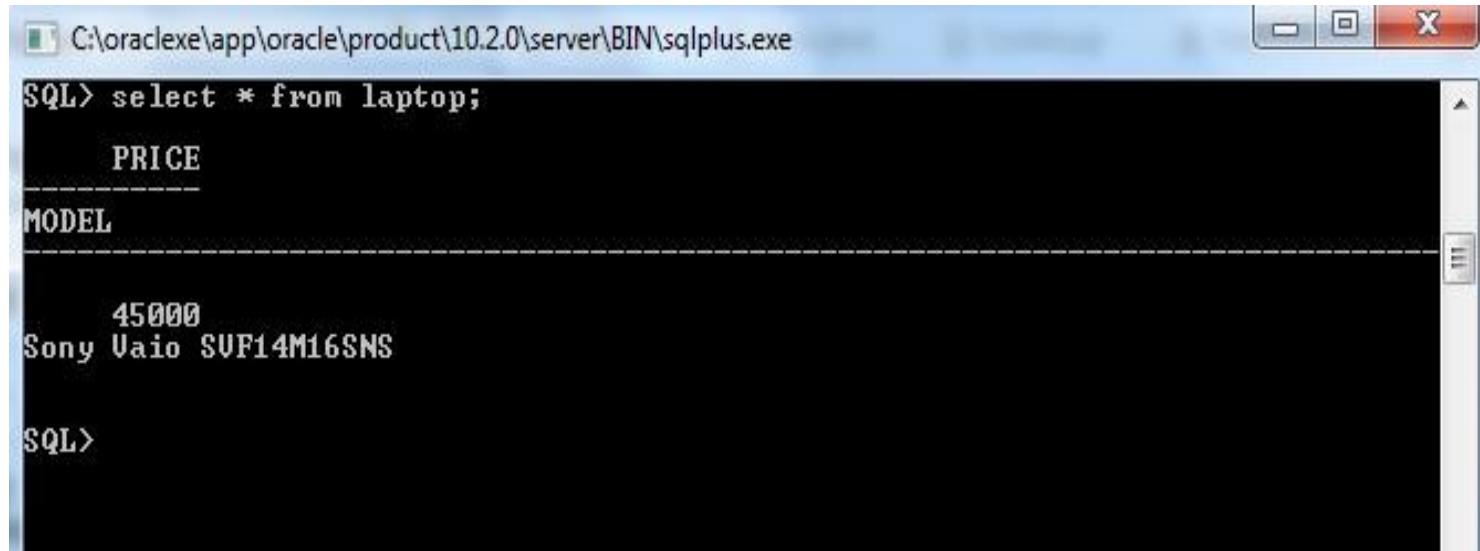
        SessionFactory sessionFactory = HibernateManager.getSessionFactory();
        Session session = sessionFactory.openSession();
        session.beginTransaction();

        session.save(lappy);

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

# Execution Output

When you execute the Test class Hibernate will create laptop table if it does not exists and add a row with the values of Laptop object. Note the name of table columns it's the same as members of your Java class. This is what Hibernate provides you don't have to write database specific SQL queries.



```
C:\oracle\app\oracle\product\10.2.0\server\BIN\sqlplus.exe
SQL> select * from laptop;

  PRICE
-----
MODEL
-----
    45000
Sony Vaio SVP14M16SNS

SQL>
```

# How it works ...

Lets go through the execution of Test class.

First we are creating a Laptop object with price 45000 and model Sony Vaio SVF14M16SNS.

Next we are retrieving the SessionFactory instance by calling getSessionFactory() method of `HibernateManager` class that we created.

Now we can retrieve Session object by calling `openSession()` method on `SessionFactory` instance. Once we have a Session object we can start a new transaction with database.

Now we are ready to interact with database. We are inserting Laptop object into database by calling save method.

You can perform any action like fetching data, deleting data or updating already present data. To commit the changes call `commit()` and end the transaction and make sure you close the session object once you are done.



# hibernate.cfg.xml

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

<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
    <property name="connection.url">jdbc:oracle:thin:@localhost:1521:XE</property>
    <property name="connection.username">system</property>
    <property name="connection.password">aaa</property>

    <!-- JDBC connection pool (use the built-in) -->
    <property name="connection.pool_size">1</property>
    <!-- SQL dialect -->
    <property name="dialect"> org.hibernate.dialect.OracleDialect</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 the existing tables and create new one -->
    <property name="hbm2ddl.auto">create</property>

    <!-- Mention here all the model classes along with their package name -->
    <mapping class="co.edureka.Laptop"/>
  </session-factory>
</hibernate-configuration>
```

# For add/list/update/delete

Following session methods are used to perform add/update/delete/list records of the database.

Save (object) → Saves the object in the database.

`createQuery("FROM Employee").list()` --> gets the result of `Select * from Employee` and returns it in a collection `List`.

`update()` --> is used to update a record in the database.

`delete()` --> is used to delete a record from the table.

We can also write separate Java methods in which we can use `update()`, `delete()` and any other hibernate provided method.

# Updating record

```
public void UpdateRecord(Integer EmpId, int salary ) throws HibernateException
{
    Session session = factory.openSession();
    session.beginTransaction();

    Employee employee =
        (Employee)session.get(Employee.class, EmpId);
    employee.setSalary( salary );
    session.update(employee);
    session.getTransaction().commit();
    session.close();
}
```

# Deleting record

```
public void DeleteRecord(Integer EmployeeID) throws HibernateException
{
    Session session = factory.openSession();
    Transaction tx = session.beginTransaction();
    Employee employee =
        (Employee)session.get(Employee.class, EmployeeID);
    session.delete(employee);
    tx.commit();
    session.close();
}
```

# Group By and Order By with Hibernate

In HQL itself, order by and group by clauses can be added for sorting and grouping columns. Examples are given below:

For Order By:

```
String hql = "FROM Employee WHERE salary > 5000 order by name";  
Query query = session.createQuery(hql);  
List results = query.list();
```

For Group By:

```
String hql = "SELECT SUM(salary), firstName FROM Employee GROUP BY firstName";  
Query query = session.createQuery(hql);  
List results = query.list();
```

# Batch Processing In Hibernate

Batch Processing is used to send multiple s statements at once rather than one by one.

To use the batch processing feature, first set hibernate.jdbc.batch\_size as batch size to a number in hibernate.cfg.xml file. If 100 is set for this batch then 100 sql statements will be executed in a batch

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://127.0.0.1:3306/test</property>
    <property name="hibernate.connection.username">root</property>
    <property name="connection.password">charan</property>
    <property name="hibernate.jdbc.batch_size">100</property>
    <property name="connection.pool_size">10</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="show_sql">>false</property>
    <property name="hibernate.hbm2ddl.auto">create</property>
    <mapping resource="employee.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

# Batch Processing In Hibernate

```
public class Employee {  
    private int id;  
    private String firstName;  
    private String lastName;  
    private int salary;  
  
    //with getter and setter methods.  
}
```

```
// Client file which inserts the data into the db:  
for ( int i=0; i<100000; i++ ) {  
    String fname = "First Name " + i;  
    String lname = "Last Name " + i;  
    Integer salary = i;  
    Employee emp=new Employee();  
  
    session.save(emp);  
    if( i % 100 == 0 ) {  
        session.flush();  
        session.clear();  
    }  
  
    if (i % 1000 == 0)  
        System.out.println("Record : " + i + " is inserting into db...Please wait...");  
}
```

This code will insert 100000 records into the employee table with the batch size of 100. i.e., one transaction for every 100 records.

# Criteria and Restrictions

Criteria is an interface to query on a particular persistent class.

Session has `CreateCriteria()` method which returns a Criteria object of a particular class to fire the query on.

Where clause of SQL can be added by using conditional methods of restrictions class. This can be added by `add()` method of criteria class. For example:

```
Criteria c = new Criteria (Employee.class);  
c.add (Restrictions.gt ("salary", 5000));  
List lst = cr.list();
```

The above 3 lines create a criteria object c. With `add()` method of Criteria object a condition is added as salary is > 5000. `list()` method of Criteria object returns the result set in collection list.





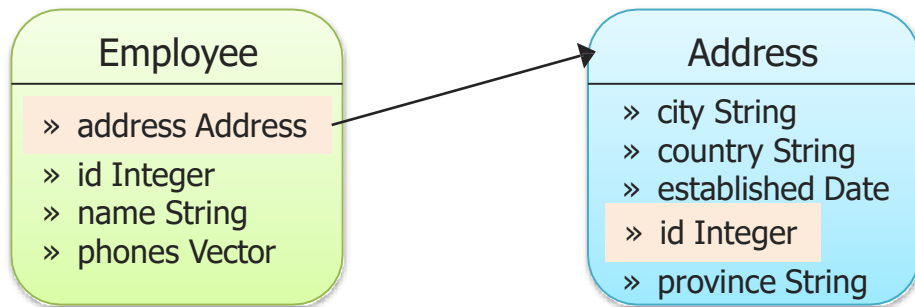
# LAB

# Mapping in Hibernate

There are 3 types of relationship mappings in database/hibernate. They are:

- (a) **One to One** → Example: One student has one teacher.
- (b) **One to Many** → Example: One organization has many employees.
- (c) **Many to Many** → Example: Many employees has many certificates.

# One to one Mapping



Java Class (one to one relationship)

EMPLOYEE table

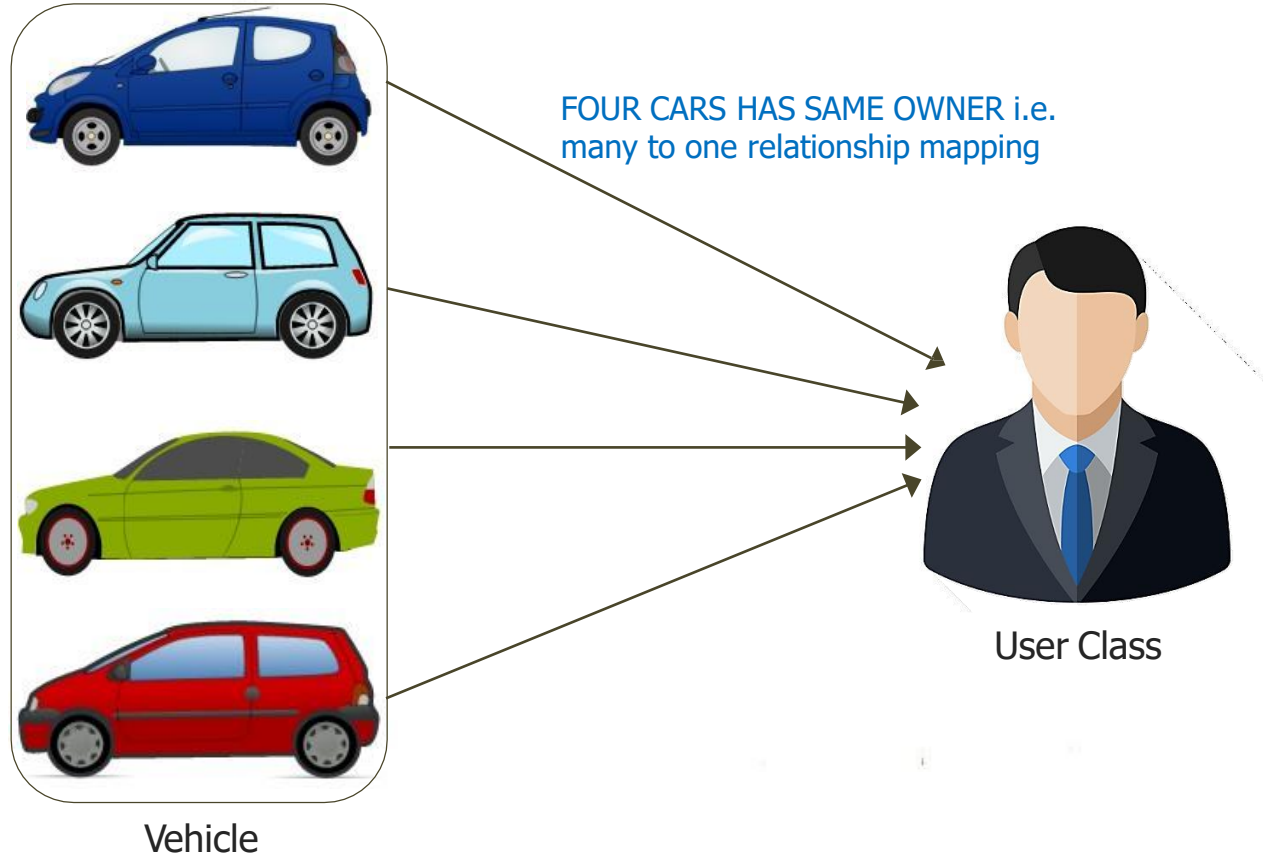
EMP_ID	NAME	ADDR_ID
203	John	312
204	Jack	226
205	Tom	230

ADDRESS table

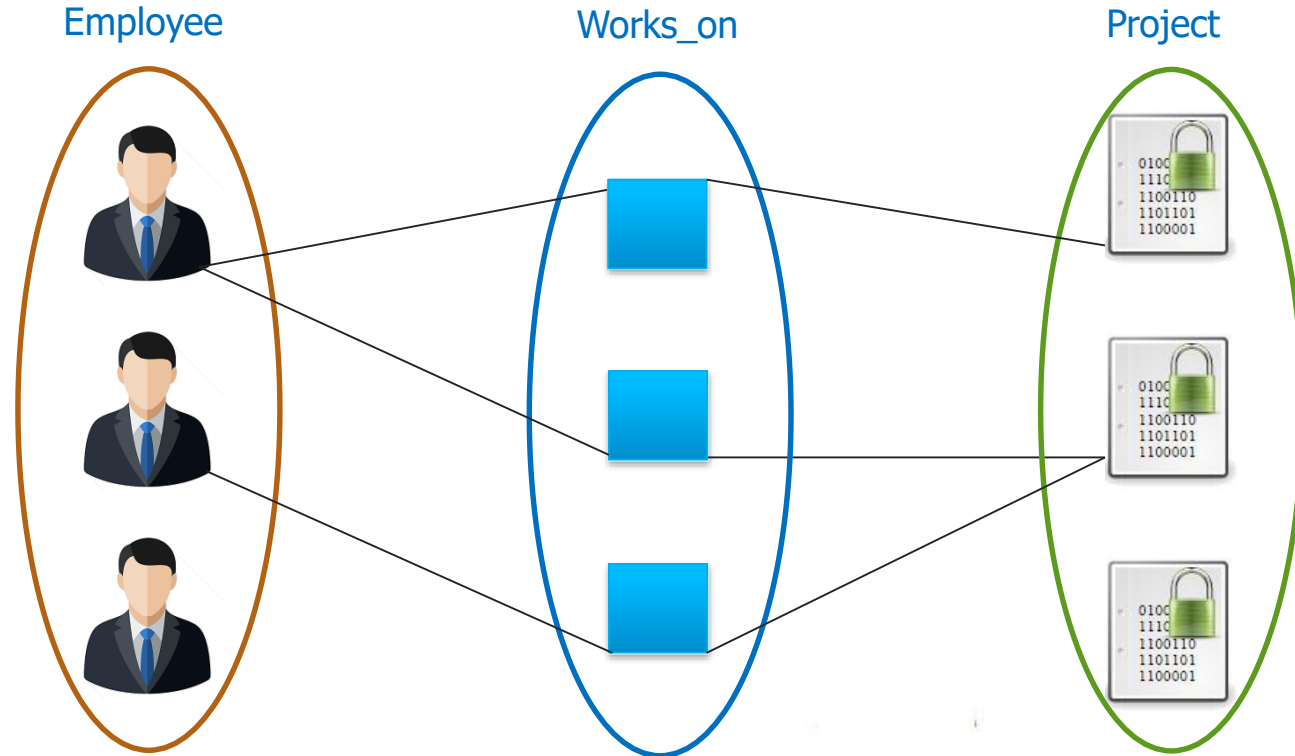
ADD_ID	CITY	COUNTRY	EST_DATE	PROVINCE
205	Washington	USA	05/03/2013	Washington
226	Texas	USA	08/02/2012	Texas
419	New York	USA	07/06/2013	New York

Relational Database (one to one relationship)

# Many to one Mapping



# Many to Many Mapping



# One to One relationship mapping

Here two persistent classes are required to connect.

**For example:** An employee having an address. Here Employee is a persistent class and Address is another.

There is one to one mapping between Employee and Address class/tables.

In the hbm file both the classes and their corresponding tables has to be mentioned. Attributes of class and fields of the table has to be mapped accordingly.

One extra paramter <one-to-one> or <Many-to-one with unique="true"> can be used to have one to one relationship.



# LAB

# Many to One and Many to Many

<Many-to-One> attribute has to be used in hbm file for many to one relationship mapping.

<Many-to-Many> attribute has to be used in hbm file for many to many relationship mapping.





# LAB

# Inheritance in Hibernate

In hibernate, inheritance classes can be mapped with database tables. One of the type of inheritance in Hibernate is [Table per hierarchy](#).

Table per hierarchy – Only one table is required for the entire hierarchy.

For implementing inheritance, define the base class and derived classes with its attributes.

In the hbm file, use the base class as the class parameter and derived classes as subclass under base class.

# Inheritance in Hibernate

```
<hibernate-mapping>

  <class name="Student" table="stu121" discriminator-value="stu">
    <id name="id">
      <generator class="increment"></generator>
    </id>
    <discriminator column="type" type="string"></discriminator>
    <property name="name"></property>

    <subclass name="Regular_Student" discriminator-value="reg_stu">
      <property name="college_address"></property>
    </subclass>

    <subclass name="Correspondance_student" discriminator-value="corr_stu">
      <property name="house_address"></property>
    </subclass>
  </class>

</hibernate-mapping>
```

# Caching in Hibernate

When a query is repeatedly fired at the database to fetch the data, performance of the system goes down. To resolve this issue, caching can be provided so that results are cached in the program to improve the performance.

There are 3 levels of caching in hibernate.

**First level cache** – Session object provides the first level caching.

Developer need not do anything for this. Hibernate provides this caching by default. Once the session is closed, the cached data will be lost.

# Annie's Question

What is Caching and why do we need it?  
What is the advantage of caching?

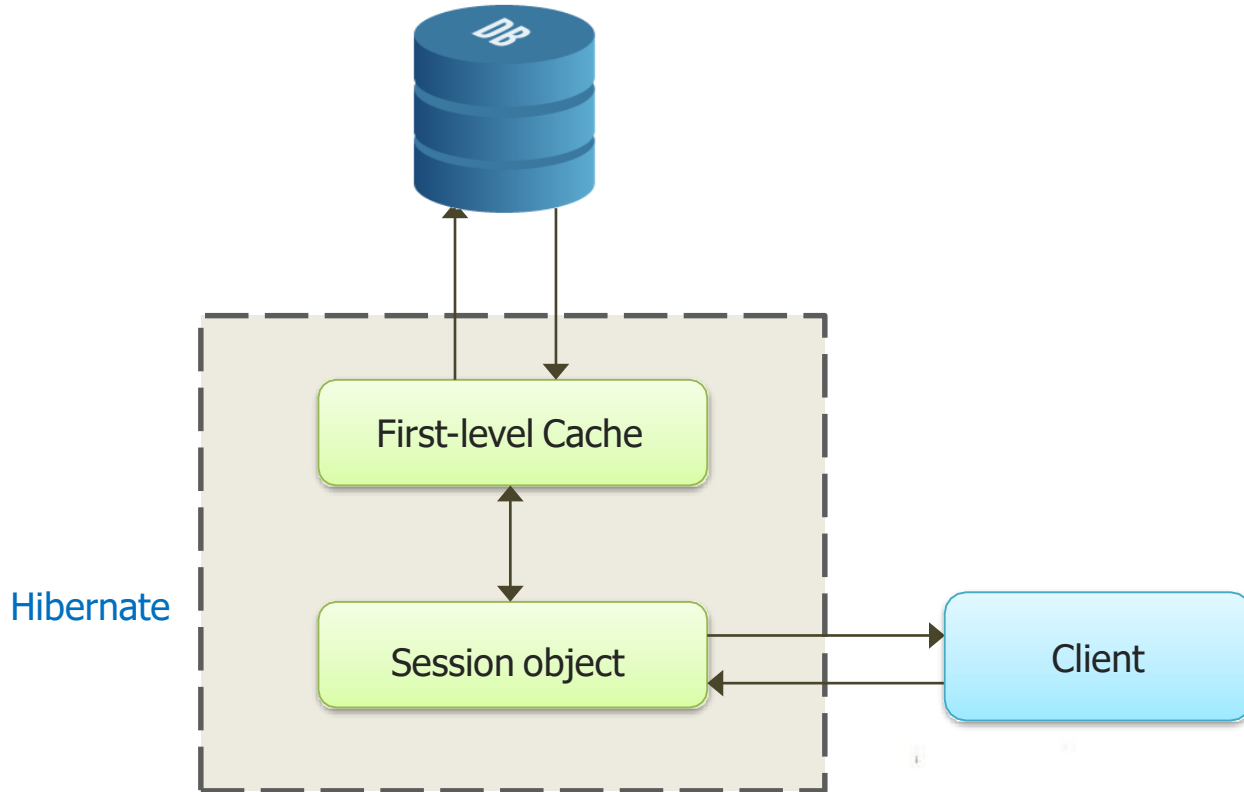


# Annie's Answer

Storing the data in memory instead of querying again and again with the database will improve the performance of the system.



# First level cache



# Second level cache

- Second level cache is build using `SessionFactory`.
- Cache parameter has to be used in hbm file for caching the class.
- Include `<cache usage="read-write"/>` in the hbm file to cache the specified class.



# Query Cache

- Query results can be cached using Query Cache in hibernate.
- To use the query cache, parameter `hibernate.cache.use_query_cache` should be set to true in configuration file.
- Use `setCacheable(true)` in the code to cache the query results.

# QUESTIONS



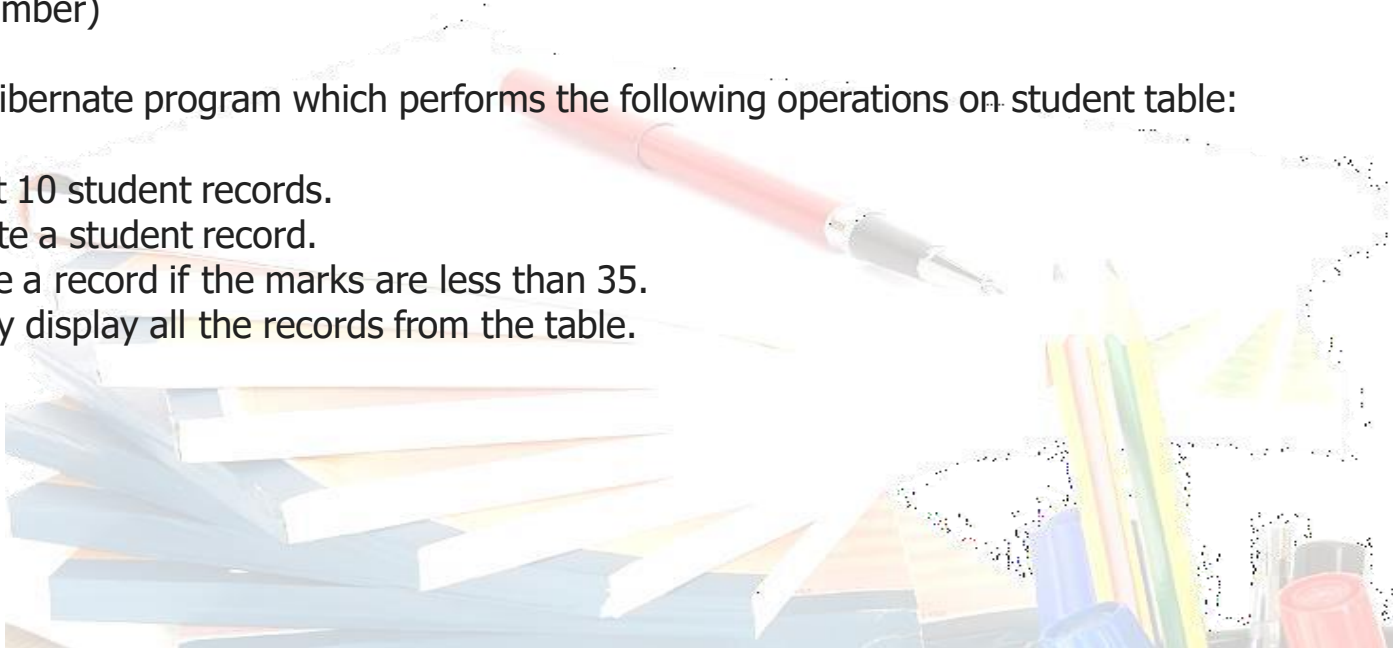
# Assignment

Student table has the following structure:

(id number,  
name varchar (20),  
marks number)

Write a hibernate program which performs the following operations on student table:

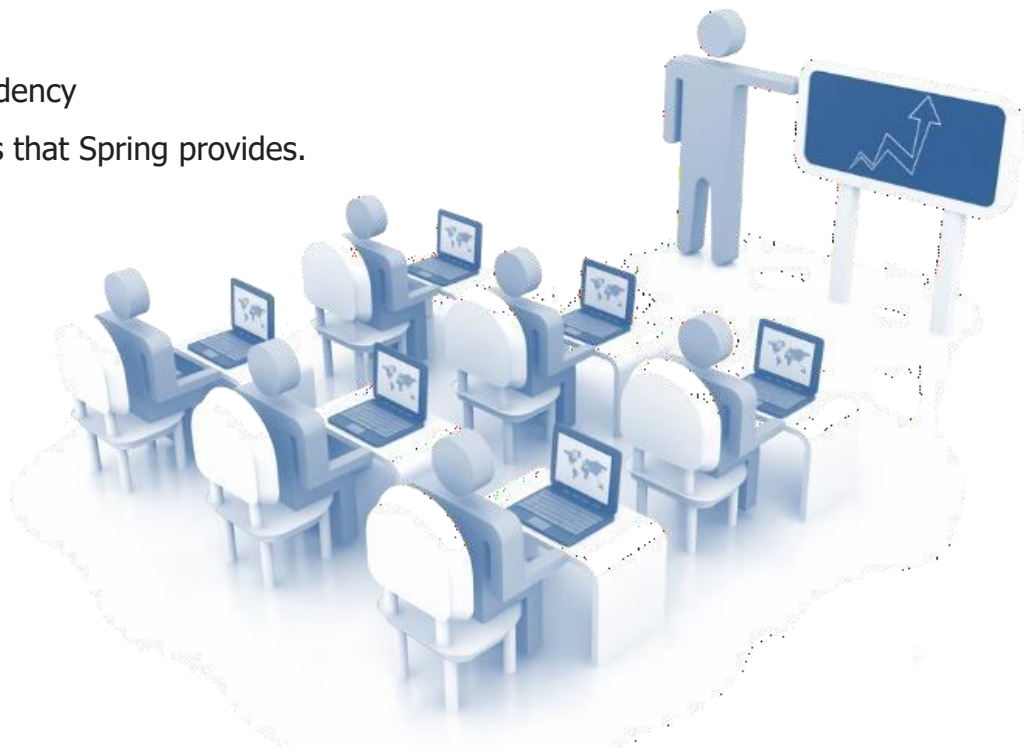
- (a) Insert 10 student records.
- (b) Update a student record.
- (c) Delete a record if the marks are less than 35.
- (d) Finally display all the records from the table.



# Agenda of the Next Module

In the next module, you will be able to

- Understand Spring
- Understand Spring Architecture and Dependency
- Learn about Injection and all other features that Spring provides.



# Pre-work

Preparation for the next module: Go through the XML module we have covered and Object Oriented concepts



Thank you!

