Introduction to Hibernate framework:

What is ORM?

ORM==Object Relation Mapping

In java application , we will have objects which are representing the data that we atote

In the database.

Student s=new Student(1,”sarika”,0);

save (s);

persist(s);

Student s1=new Student(2,”jaya”,10);

Save(s1);

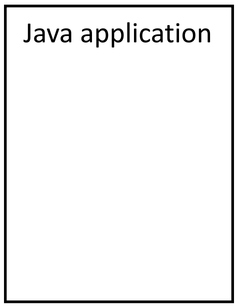
Table created in DB are called relational entities.

To store object we need to map class with tables so that each object od that class can be mapped to each row.

This kind of mapping between object and tables is called ORM.

Hibernate:

What is Hibernate??



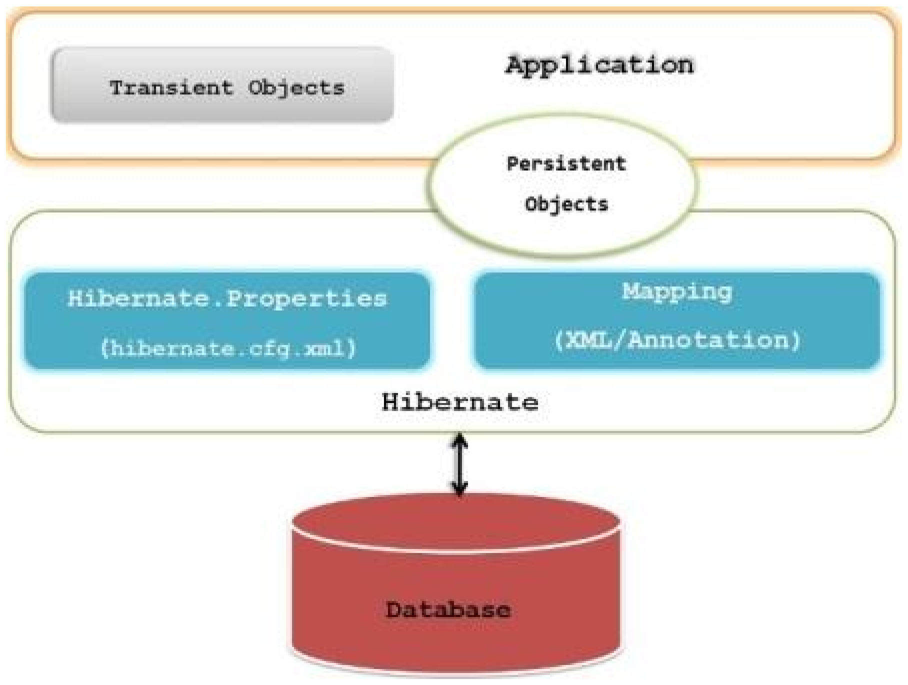
We will see Hibernate architecture in 2 views

1) High level view of Hibernate Architecture

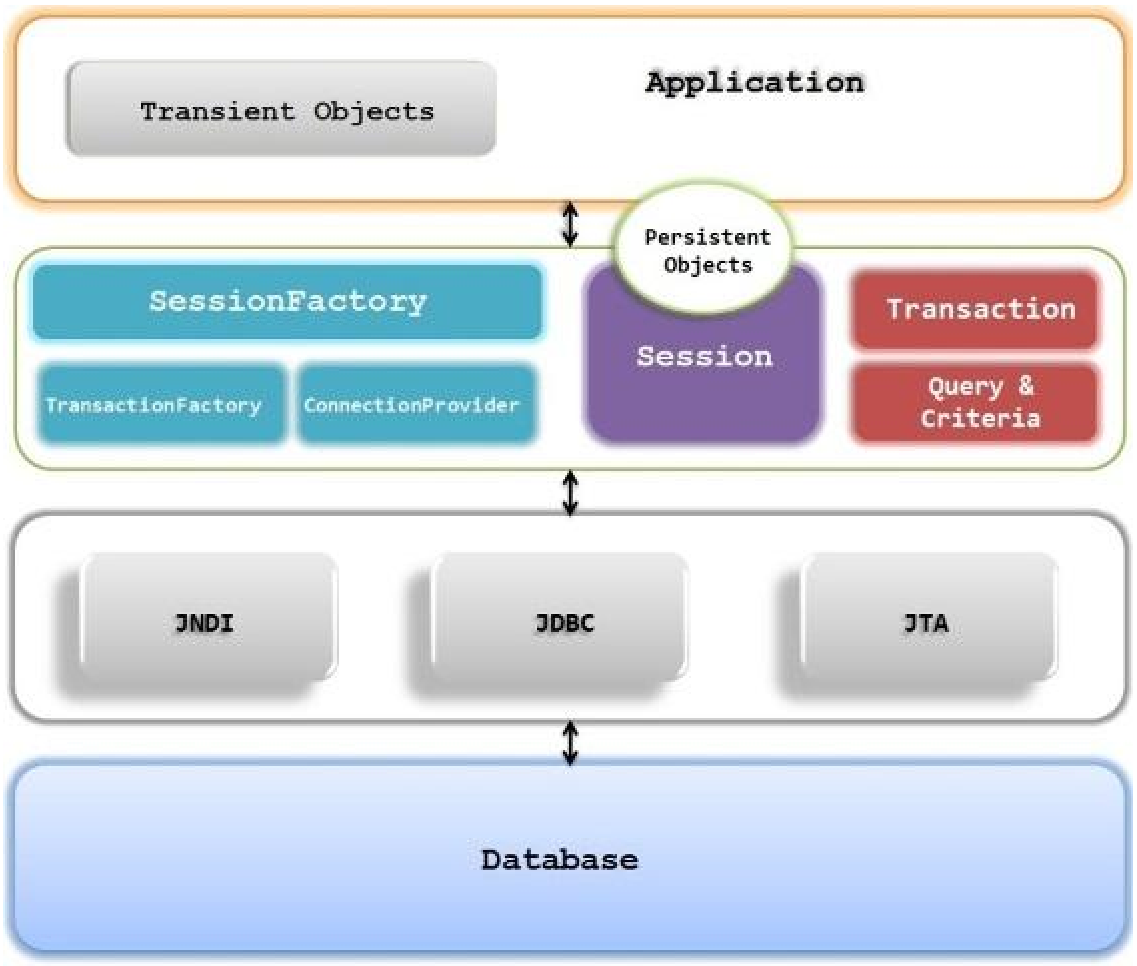
2) Detailed view of Hibernate Architecture

Hibernate architecture shows **configuration details** and various objects involved in each layer
Hibernate uses Java API like **JDBC API**,Java Transaction API(**JTA**) and Java Naming and directory Interface(**JNDI**)

The below figure illustrates High Level View of Hibernate Architecture



The below figure illustrates Detailed View of Hibernate Architecture



Let’s discuss various elements involved in Hibernate architecture

Configuration

It represents a configuration required to use the hibernate , which could be **properties file** or **XML file**.
The **Configuration object** is usually created once during **application initialization**.
The Configuration object uses the configuration details from the configuration file to get connected to a database.
A Configuration object is used to **create a SessionFactory**.

Object Relational Mapping

The mapping between Java **POJO class** and the **database tables** is provided using either **XML** or **annotation**.

SessionFactory

It is the **factory** for the **session objects.**
It is **thread safe** and **immutable object** representing the mappings for a single database.
It can hold the second level cache in hibernate

Session

It’s a single-threaded, **short-lived** object which acts as a **bridge** between **Java application** and **Hibernate**.
It wraps a JDBC connection and is a factory for Transaction.
Session holds a mandatory **first-level cache** of persistent objects

Transient objects

Instances of persistent classes which are not currently associated with a Session

Persistent objects

They are associated with Session and Once the Session is closed, they will be detached and free to use in any application layer

Query

Query objects use **Native SQL** or Hibernate Query Language (**HQL**) to retrieve data from the database.
A Query instance is used to bind query parameters such as number of results returned by the query.
We can obtain the **query object** from the session using **session.createQuery()**

Criteria

The **org.hibernate.Criteria** is used to retrieve the entities based on some criteria like getting all employees by **specific ag**e or **specific date of joining** etc.
We can get the Criteria from the Session.

TransactionFactory

A factory for Transaction instances,It is not exposed to the application and it's optional.

Transaction

single-threaded, short-lived object used by the application to specify atomic units of work
We can obtain the **transaction** from the **Session object**.

ConnectionProvider

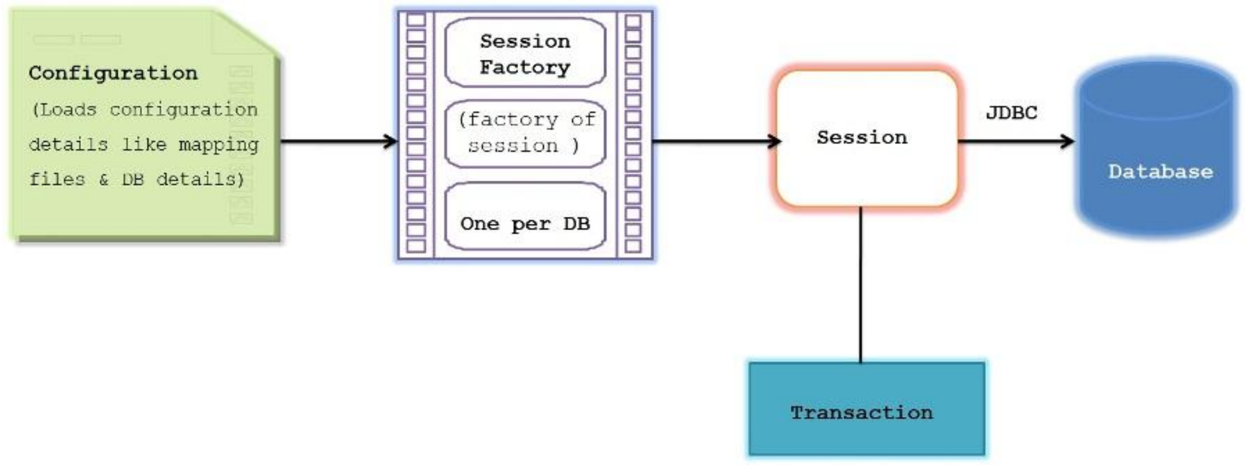
It's a pool of JDBC connections

We need to understand the below concepts in Hibernate before using them in coding

Configuration

SessionFactory

Session



Configuration

Hibernate should have the information to **map** the **java classes** with the **database tables**.
This information is ideally kept in the **mapping XML file**.

Hibernate also need to have the database related configuration like database **url, credentials , dialect** etc.

This information to find out the **mapping files** and the **database configuration** is provided in the **hibernate configuration file**.

The **default** names for this **configuration file** will be either **hibernate.properties** or **hibernate.cfg.xml**
We can use any other name as well for the configuration.

org.hibernate.cfg.Configuration is used to build an instance of immutable **org.hibernate.SessionFactory**
We will see about SessionFactory and its immutability soon.

Configuration will be created as below with **default configuration file**

```
Configuration cfg=new Configuration();
cfg=cfg.configure();
```

whenever we call **configure()** method It looks for **hibernate cfg.xml** and also looks for **hibernate mapping file**.

Configuration will be created as below with **custom configuration file**

```
Configuration cfg=new Configuration();
Configuration cfg1=cfg.configure("application.cfg.xml");
```

Here we are passing **custom file** name to **configure()** method and hence it looks for **application.cfg.xml** and also looks for **hibernate mapping file.**

Session Factory

SessionFactory is the **factory** for the **session objects**.

We use Configuration object to build the **SessionFactory**, It will be **only one per database** but there can be **multiple session objects**

We obtain **SessionFactory** using **Configuration object** as below

```
SessionFactory sessions = cfg.buildSessionFactory();
```

If we have **multiple databases** in our application, then we need to create **multiple SessionFactory objects**

Example:
If we are using **2** databases called **mysql** and **oracle** in our application then we need to build **2 SessionFactory objects** as below

```
Configuration cfg=new Configuration();
Configuration cfg1=cfg.configure("mysql.cfg.xml");
SessionFactory sf1=cfg1.buildSessionFactory();

Configuration cfg2=cfg.configure("oracle.cfg.xml");
SessionFactory sf2=cfg2.buildSessionFactory();
```

Now **sf1** is associated with **mysql** and **sf2** is associated with **Oracle** database.

Why SessionFactory is immutable and thread safe ?

In Real time, many threads can access the **SessionFactory** concurrently to request a session, If the object is **mutable** , then accessing such mutable objects by multiple threads concurrently can lead to **inconsistent state**.
Hence making such object as **immutable** will always be **thread safe**

Session

Session acts as a main **bridge** between **Java application** and **Hibernate**
We can establish a connection with **database** and **interact** with it using **Session**.

Session is a **lightweight object** and hence can be created whenever the **interaction with Database is necessary.**
Unlike SessionFactory, **Session** is **not thread safe** and hence it should **not** be kept **open** for a **long time**.
So Create and close the session on a need basis.
Each Session is bounded by the **beginning** and **end** of a **logical transaction**.

We can use the session as below

```
Session session = sessionFactory.openSession();
 Transaction tx;
 try {
     tx = session.beginTransaction();
     //do DB related work
     ...
     tx.commit();
 }
 catch (Exception e) {
     if (tx!=null) tx.rollback();
     throw e;
 }
 finally {
     session.close();
 }
```

sessionFactory.openSession() is used to obtain the session from SessionFactory

session.beginTransaction() is used to begin the transaction

tx.commit() is used to commit the transaction

tx.rollback() is used to roll back the transaction if any exception is thrown

session.close() is used to close the session.

It is good practice to **Rollback** the **entire transaction** whenever any exception is thrown within the session as shown in the Catch block.

The **internal state** of the Session **might not be consistent** with the database after the exception is thrown and hence we should **discard** the session hence closing it in the **finally** block.

Object Lifecycle and states in Hibernate

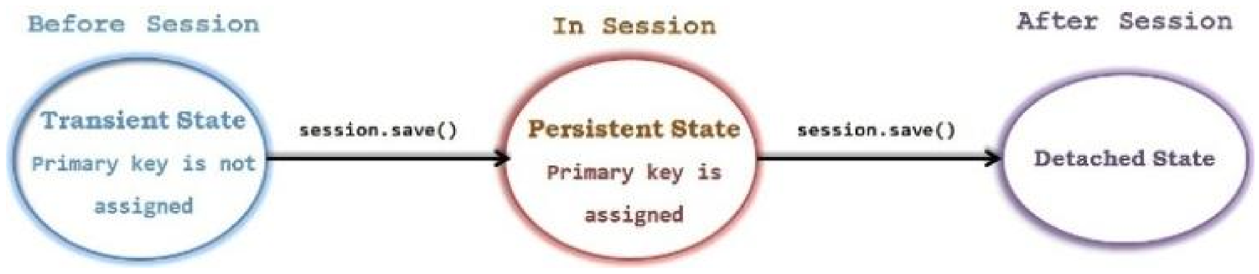
Looking at the state of an object in Hibernate, we will get to know whether object is **saved** into **Database or not**.

The 3 states of an object in Hibernate

Transient

Persistent

Detached



Let’s see these states in detail with example

Transient state

A transient object is one that Hibernate has no idea on it.

It’s the **first state of any object** in the Hibernate Life cycle.

Any object which is **not associated** with hibernate **session** and does not represent a row in the database is considered as transient.

It will be **garbage collected** by JVM,If **no** other **object** is **referencing** it.

An object that is created using the **new()** operator is in transient state.

When the object is in transient state then it will **not contain** any identifier (**primary key value**).

We have to use session methods like **save, persist** or **saveOrUpdate** methods to persist the transient object.

Example:

```
User user = new User();
user.setName("kb");
```

Now **user** object is not associated with any session and hence it can be called as **Transient** object.

Persistent state

An object that is **associated** with the hibernate **session** is called as Persistent object.

When an object is in a persistent state, **Hibernate** is totally **aware of it**.

It also represents **one row of the database** and consists of an identifier value.

We can make a **transient instance persistent** by associating it with a **Session** like below

```
Long id = (Long)session.save(user);
//This object is in persistent state
```

Object can get the Persistent state in the below 2 scenarios

1) Load the object from the Database using Hibernate API
In this case, we load the **existing object** from the database and it will be **automatically associated with the session.**

2) Save the object into the database using Hibernate API
In this case new transient object created using “**new**” operator will be **attached** to the **session** and it becomes Persistent object.

Note: *Object in Persistent state will be saved only when the Transaction is committed.*

Detached state

The detached state is given to an object that was **previously “attached**” to the **Session** (persistent state) but has **now left** the association with the Session.

Example:

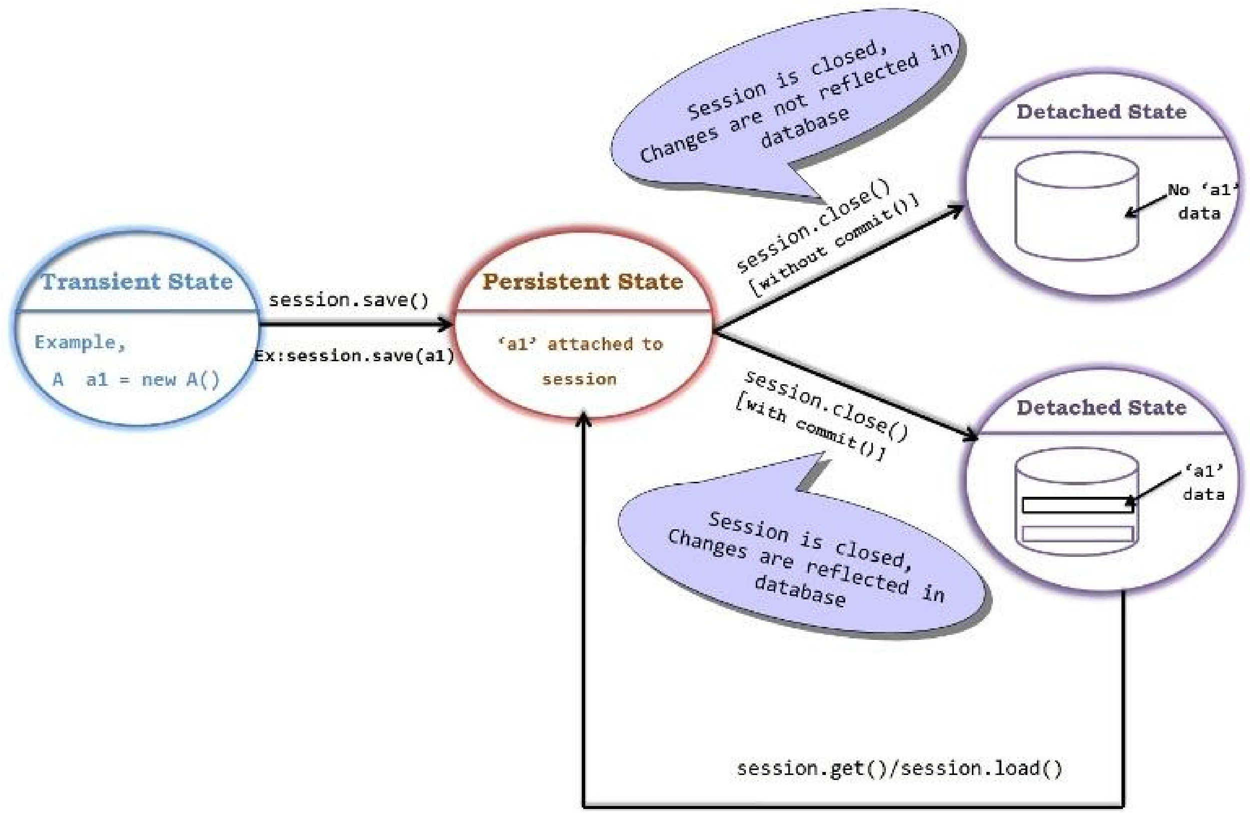
```
Session.close();
//All the objects will be in detached state after this line execution
```

All the **persistent objects** within the Session will get the **detached state** when we close the session.

If we would have done **transaction commit before closing the Session** then all **Persistent objects** will be **saved** into Database otherwise those persistent objects will be **lost**.

So object in Detached State **may present** in the Database but **not guaranteed**.

So most important thing is to commit the transaction before closing the Session.


Transient State
Example,
A a1 = new A()
session.save()
Ex:session.save(a1)
Persistent State
'a1' attached to session
Session is closed, Changes are not reflected in database
session.close() [without commit()]
Detached State
No 'a1' data
session.close() [with commit()]
Session is closed, Changes are reflected in database
Detached State
'a1' data
session.get()/session.load()

```
public class HibernateObjectStates {
public static void main(String[] args)
 {
  User user= new User();
                  user.setName("John");
                  user.setCity("Newyork");
  //'user'  is in  TRANSIENT state

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

                  session.save(user);
  //Here 'user' is in PERSISTENT state
```

```
                session.getTransaction().commit();
  //'user' will be saved to Database

                session.close();
  //'user'  is in  DETACHED object
 }
}
```

Transient VS Detached state

Few people get confused with **Transient and Detached State** as both of them are **not associated** with the **Session**.

Transient objects do not have any association with either database or session objects.
They are just created using **"new"**.

Once the **reference is lost**, object is collected by garbage collector.

The **commits** and **rollbacks** operation will have **no effects** on these objects.

They can become persistent objects through the **save()** method of Session object.

Hibernate does not even know that there is an object if it's in Transient state

Detached objects will have corresponding **entries in the database** but they are not associated with the Session.

Detached objects were **associated with Session** and known to hibernate in the past but not currently.

The **detached object** can be **reattached** to session to make it persistent again.

Note:
*If object is in transient state, then it is **guaranteed** that it has **no existence** in the database.*
*If object is in detached state, even though it is **not guaranteed** (depends on transaction is committed or not) but there is a **possibility** that object may have **existence** in the database already.*

Inheritance in Hibernate overview

Let us understand about Inheritance in Hibernate

Inheritance is an object oriented feature which provides the power of **re-usability**.

The same concept of **re-usability** is provided in **Hibernate** as well.

It is one of the advantages of Hibernate when compared with JDBC.

Assume we have **Parent** and **Child** classes, we know that Child class object will have an access to Parent class attributes as well.

In Hibernate when we save **Child** class object , **Parent** class **attributes** will also be **saved**.

Now you may get multiple questions in your mind

1. Whether **parent** and **child** class attributes are saved in the **same** table

2. Whether **parent** and **child** class attributes are saved in **separate** table

3. If they are stored in **separate** table, how will they be linked

Yes if you have above questions in your mind, Don't worry you will be getting answers to these questions shortly.

Hibernate provides 3 different ways to represent the inheritance

1.Table per Hierarchy

2.Table per Concrete class

3.Table per Subclass

Let us consider the below example to discuss all the inheritance types

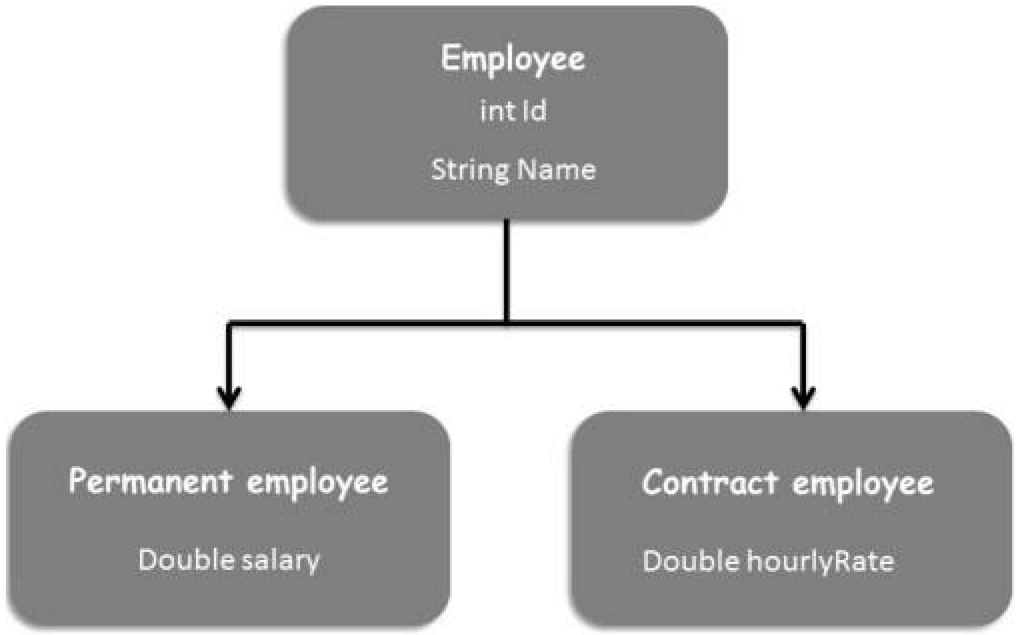


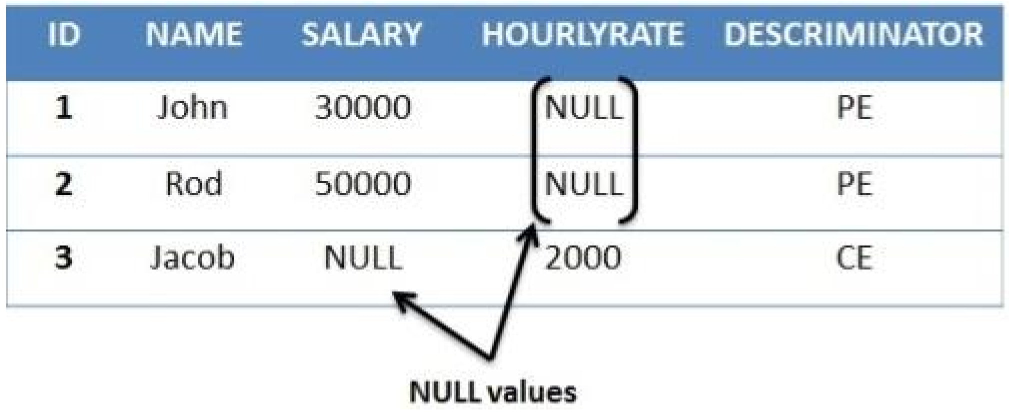
Table per hierarchy

In this approach, as the **name** suggests the entire hierarchy is mapped to a **single table**. I.e. **All attributes** of all the classes are stored in a **single table**.

A **discriminator column** is used to distinguish different classes.

Null values will be stored in the table for which there is **no column applicable**.

Example:
We have stored **2** permanent and **1** contract employee details in the below table



ID	NAME	SALARY	HOURLYRATE	DESCRIMINATOR
1	John	30000	NULL	PE
2	Rod	50000	NULL	PE
3	Jacob	NULL	2000	CE

NULL values

We can see **NULL** values are stored in **Salary** column for **Contract employee** and NULL value is stored in **HourlyRate** column for **Permanent employees**.

Note:
Table per hierarchy is not an optimized way of storing relational data as many columns will get NULL values.

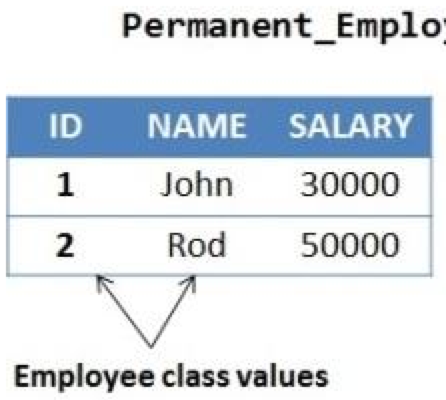
In this approach Number of tables equals to **one**, no matter how many classes are involved.

Table per Concrete class

In this case, **one table for each concrete class** will be created.

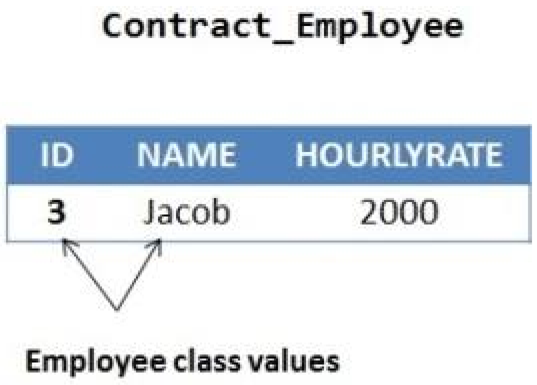
All the attributes of parent class will be stored in separate table created for each concrete class.

Example:
Assuming Employee class as Abstract,we will get only these 2 tables created for concrete classes.


Permanent_Employee

ID	NAME	SALARY
1	John	30000
2	Rod	50000

Employee class values


Contract_Employee

ID	NAME	HOURLYRATE
3	Jacob	2000

Employee class values

The main **disadvantage** of this approach is that parent class attrbiutes are **duplicated** in **each subclass table**.

In this approach, Number of tables equals to number of Concrete class.

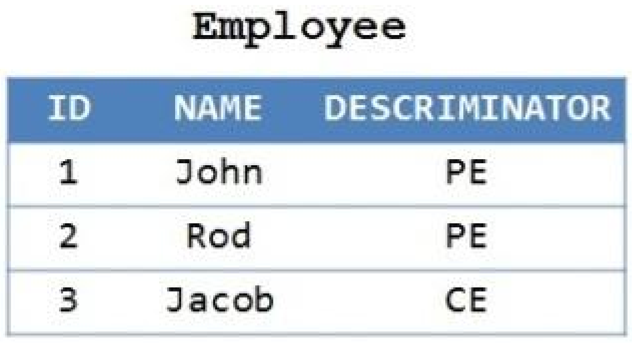
Table per Subclass

In this approach, **separate table** is required for **each class**.

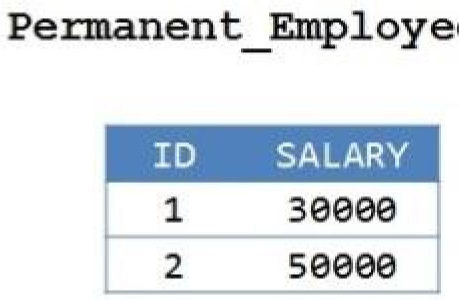
Each Subclass table will have only subclass specific columns by having one extra column to have a relationship with the Parent table.

Since there will be a **foreign key relationship** between the **tables**,there will **not** be any **duplicate columns**.

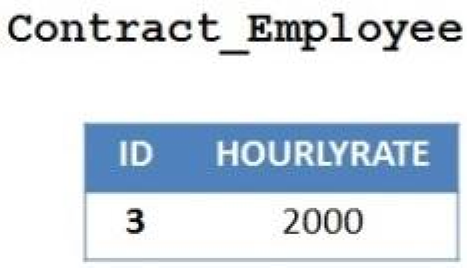
Example:


Employee

ID	NAME	DESCRIMINATOR
1	John	PE
2	Rod	PE
3	Jacob	CE


Permanent_Employee

ID	SALARY
1	30000
2	50000


Contract_Employee

ID	HOURLYRATE
3	2000

As shown above, **columns are not duplicated** as they are **referenced** using **foreign key** in each subclass table.

In this approach, Number of tables equals to number of classes.

Let us understand Hibernate Criteria Query Language (HCQL)

What is HCQL

It's a Criteria based query language mainly used to **fetch the records** based on specific search **criteria**.

It supports complete object oriented approach for querying and retrieving the result from database based on search criteria.

HCQL **can not** be used to perform **DML operations** like Insert,Update and Delete.

It can be used **only for retrieving the records** based on search conditions.

Criteria queries should be **preferred** when we have **many optional** search condtions.

org.hibernate.Criteria interface has provided several methods to add search conditions.

Most commonly used methods in Criteria are

public Criteria add(Criterion c):
This method is used to add restrictions on the search results.

public Criteria addOrder(Order o)
This method is used to define the ordering of result like ascending or descending.

public Criteria setFirstResult(int firstResult)

public Criteria setMaxResult(int totalResult)

These 2 methods are used to achieve pagination by specifying first and maximum records to be retrieved.

Example: If there are 50 records in database and if we are defining the pagination with 25 records per page
FirstResult – 1 and MaxResult – 25 and then FirstResult-26 and MaxResult -25

public List list()
This method returns the list of object on which we are searching.

public Criteria setProjection(Projection projection)
This method is used to set the projection to retrieve only specific columns in the result.

Possible restriction used in HCQL are

lt(*less than*)
le(*less than or equal*)
gt(*greater than*)
ge(*greater than or equal*)
eq(*equal*)
ne(*not equal*)
between
like

Criteria with Projection

Projections will become **handy** when we want to load the **partial object**.

Partial object means **only few attributes** will be loaded rather than **all the attributes**.

In some cases, it is unnecessary to load all the attributes of an object.

Main points to remember about Projections

Projection is an Interface defined in "org.hibernate.criterion" package

Projections is a class and it is a factory for producing the projection objects.

Projection is mainly used to **retrieve partial object**.

To add a Projection object to Criteria , we need to call a **setProjection()** method on Criteria.

We can **add as many projection objects** as we want but **only latest object** will be **considered**, so no use in adding multiple projection objects to criteria.

We need to create one projection object for specifying one property.

If we want **more than one property** to be included in the result, then we need to **create multiple Projection objects** one for each property and add all of them to **ProjectionList** and then we need to set **ProjectionList object** to Criteria.

Example:
Below example illustrates retrieving only one column of a table using Projection.

```
Criteria criteria = session.createCriteria(Employee.class);
Projection projection = Projections.property("firstName");
criteria.setProjection(projection);
List list = criteria.list();
```

In the above example, we applied Projection on only one property "**FirstName**".

The equivalent SQL query is "**select firstName from Employee**".

If we want to apply projections to **retrieve multiple properties,** we can use **ProjectionList** as shown below

```
Criteria criteria = session.createCriteria(Employee.class);
Projection projection1 = Projections.property("firstName");
Projection projection2 = Projections.property("salary");
Projection projection3 = Projections.property("age");

ProjectionList projectionList = Projections.projectionList();
projectionList.add(projection1);
projectionList.add(projection2);
projectionList.add(projection3);
criteria.setProjection(projectionList);
List list = criteria.list();
```

Named Queries in Hibernate

If we want to use **same queries in multiple places** of an application, then instead of writing same query in multiple places,

we can **define the query in one place** with the name assigned to it and use that name in all the places wherever required.

The concept of defining the query with name in one place and accessing that query by using its query name wherever required is called Named Query.

Named query will be defined in the **hibernate mapping file**.

Named query can be used for both **HQL queries** and **Native SQL queries**.

Hibernate Named query example with HQL

```
< query name="hql_select_employee" >from Employee  e  where e.id=:empId< /query >
```

For HQL named query, **< query >** tag should be used.

we have given the **name** to it, so that we can access the query with that **name** from required class.

We can also pass the **dynamic parameters** as well to named query as shown above

Accessing the HQL named query in our class

```
Query query = session.getNamedQuery("hql_select_employee");

query.setParameter("empId",new Integer(1));
```

Hibernate Named query example with SQL

```
< sql-query name="sql_select_employee" >select * from Employee e where e.id=:empId</sql-query>
```

For SQL named query, **< sql-query >** tag should be used.

we have given the **name** to it, so that we can access the query with that **name** from required class.

We can also pass the **dynamic parameters** as well to named query as shown above.

Accessing the SQL named query in our class

```
Query query = session.getNamedQuery("sql_select_employee");

query.setParameter("empId",new Integer(1));
```

We can define Named query using either XML mapping or Annotation mapping.

//==

overview of Hibernate Caching

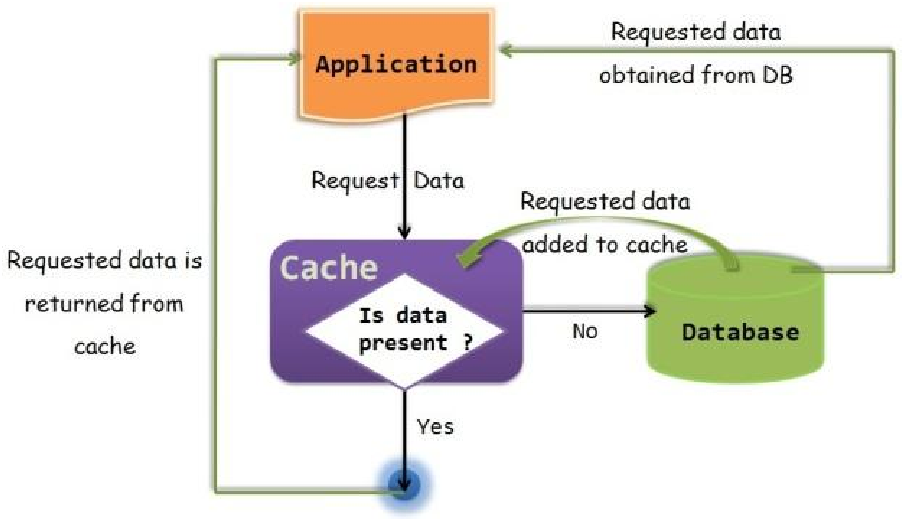
Caching is a mechanism to store the frequently retrieving data from DB into Cache Memory.

The main advantage of using Cache is, it reduces the number of database calls and increases the performance of the application.

Cache sits between application and database.

How Application gets data from Cache ?

Application first tries to find the data in cache and if requested data is not available in cache then only retrieve it from the DB and also put the same data in cache for later use.


Application
Requested data obtained from DB
Request Data
Requested data added to cache
Requested data is returned from cache
Cache
Is data present ?
No
Database
Yes

Cache in Hibernate

One of the most powerful features of hibernate is Caching.

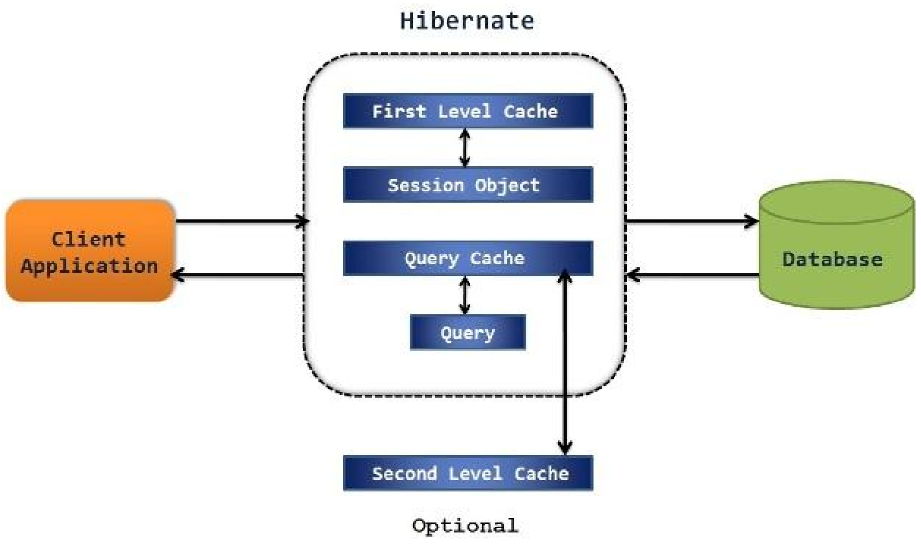
Hibernate keeps the objects in the Cache memory to reduce the DB calls thereby increases the performance.

Hibernate supports cache at different levels as explained below

1)Primary Cache or First level Cache

2)Secondary Cache or Second level Cache

3)Query Cache



Primary Cache

First level cache also called **Primary cache or Session level cache**

Primary cache is associated with the Session object, so all the **objects** within the hibernate session are **kept in Primary cache** to avoid multiple DB calls but once the **session is closed** all the **cached objects** will be **lost**.

How to enable/disable Primary cache ?

Primary cache is enabled by default and we don't have any control to disable it.

However we can **delete the objects from the primary cache or clear the primary cache completely** using Hibernate provided methods.

Secondary Cache

Second level cache also called **Secondary cache or SessionFactory level cache**

Secondary cache is associated with the **SessionFactory** and hence its available to the entire application.

So objects kept in the secondary cache is **available across multiple sessions**.

How to enable/disable Secondary cache ?

Secondary cache is disabled by default and we can enable it anytime using Configuration.

There are various **third party implementation providers** for secondary cache and some of them are

EH Cache
JBoss Cache
OS Cache
Swarm Cache

Query Cache

Query cache will cache the **results of the query against the object.**

If we have **queries which runs multiple times with the same parameters** then Query caching is best to use to **avoid multiple DB calls**.

How to enable/disable Query cache ?

Query cache is disabled by default and we can enable it using configuration.

We just need to set the ***hibernate.cache.use_query_cache*** property to true to **enable Query cache.**

Since most queries do not benefit from caching of their results, **we need to enable caching for individual queries**, even after enabling query caching overall.

To enable results caching for a particular query, we need to call ***org.hibernate.Query.setCacheable(true)***.

This call allows the query to look for **existing cache results** or add its results to the cache when it is executed.

Let us understand Primary cache in Hibernate

Primary cache is **associated with Session** and its enabled by **default**.

So we **don't need to configure anything** to enable the Primary cache in our application.

Since it is associated with Session, **primary cache** will be **available** as long as **hibernate session is alive.**

Once the session is closed, all the objects in the cache will be lost.

Important points about Primary cache

Primary cache is always associated with "Hibernate Session"

Primary cache is enabled by default and we don't need to do any additional settings to enable it.

Primary cache can not be disabled but can be cleared.

When we query for an entity , first time it loads from database and it will keep it in the First level cache associated with the Session.

If we request the same entity using same session again , then it will be loaded from cache and no database call will be made.

If we want to remove a specific entity from primary cache , we can use evict() method.

If we want to remove all the objects stored in the primary cache, then we can use clear() method.

Let us understand Secondary cache in Hibernate

Secondary cache is associated with the SessionFactory and hence its available to the **entire application.**

So objects kept in the **secondary cache** are available across **multiple sessions**.

*Once the session factory is closed, secondary cache is cleared***.**

How secondary cache works ?

Whenever we try to load an entity , **Hibernate first looks at a primary cache** associated with a particular session.

If **cached entity is found** in the primary cache itself then it will be **returned**.

If requested entity is **not found in primary cache**,then hibernate looks at the **second level cache**.

If **requested entity is found in second level cache**,then it will be **returned**.

If requested entity is **not found in secondary cache** then database call is made to **get the entity** and it will be **kept** in both **primary and secondary cache** and then it will be **returned**.

How to enable secondary cache ?

We just need to follow 3 simple steps to enable secondary cache.

1) Add below configuration setting in hibernate.cfg.xml file

< property name=”cache.provider_class” >org.hibernate.cache.EhCacheProvider< /property >

< property name=”hibernate.cache.use_second_level_cache” >true< /property >

2) Add cache usage setting in hbm file or annotated class **as below**

XML file – *< cache usage="read-only" / >*

Annotated class **–** *@Cache(usage=CacheConcurrencyStrategy.READ_ONLY, region=”employeeCache”)*

3) Create ehcache.xml file to configure the cache region

Let us see complete project on secondary cache using “Ehcache”

There are many other Cache providers but **EhCache is one of the most widely used cache provider with Hibernate**.

By default, Ehcache stores the cached files in temp folder. Which can be configured as below

< diskStore path=”java.io.tmpdir” / > but in the above file, we have stored in specific folder “cache” in C drive, for this we have used < diskStore > tag.

We have defined our cache name as “employeeCache”

defaultCache
It is a mandatory configuration, which is used when an Object needs to be cached and there are no caching regions defined.

cache name=”employeeCache” : This is used to define the cache region and its configuration.

eternal=true/false

If we specify eternal=”true”, hibernate will internally define the idle time and time span of cache region.

Hibernate Fetch types

Let us understand the different modes of Fetch and When to use them

In any **ORM** framework, it's very important to understand *how it loads the entity* especially when entity has **relations** and **collections** in it.

In **Hibernate** we call it as **Fetch mode** or **fetching strategy**.

Yes Hibernate decides how to load the entity based on the **fetch mode**.

Example:

If **User** entity has a Collection of **Address** entity in it.
When we load User, Should we load Addresses as well ? Or We should load only User ?

Answer to this question in **Hibernate is based on Fetch mode** we specify.

When we load **User**,at the same time if we load **Address also**, then its called as **EAGER loading**.

If we **delay** the loading of **Addresses** while loading **User** until we **require** Addresses then it's called **Lazy loading**.

Fetch mode helps us in customizing the number of queries generated and amount of data retrieved.

Different Fetch modes supported by Hibernate

1) **FetchMode JOIN**

Eager loading which loads all the collections and relations at the same time.

2) **FetchMode SELECT(default)**

Lazy loading which loads the collections and relations only when required.

3) **FetchMode SELECT with Batch Size**

Fetch upto "**N**"collections or entities("Not number of records")

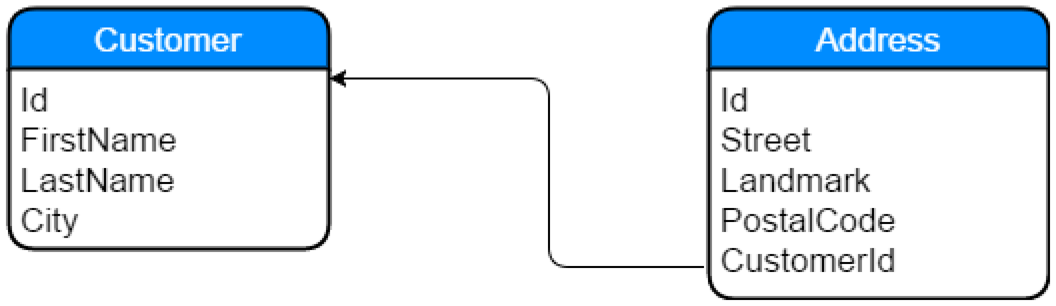
4) **FetchMode SUBSELECT**

Group the collection of an entity into a Sub-Select query.

Now let us understand these fetch modes in detail with below example

Customer entity has a Collection of **Address** entity

It is a **one to many** relation (one Customer will have many addresses)



Code to define the fetching strategy

We can define Fetch mode either through XML mapping or through annotation mapping

1) Let's see XML mapping for the same

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.kb.model">
    <class name="Customer" table="customers">
        <id name="id" type="int" column="Id">
            <generator class="increment" />
        </id>
        <property name="firstName">
            <column name="FirstName" />
        </property>
        <property name="lastName">
            <column name="LastName" />
```

```
        </property>
        <property name="city">
            <column name="City" />
        </property>

        <list name="addresses"  cascade="all" inverse="true"
            table="address" fetch="select" batch-size="10">
            <key>
                <column name="CUSTOMER_ID" not-null="true" />
            </key>
            <list-index column="idx" />
               <one-to-many class="Address" />
        </list>
    </class>
</hibernate-mapping>
```

We can see that while defining list of addresses using **< list >** tag, we have specified **fetch** attribute.

We have also specified **batch-size** as **10** which is required in Batch fetch mode.

2) Now Let's see how can we achieve the same through Annotations

```
@Entity
@Table(name = "customers", catalog = "javainsimpleway")
public class Customer implements Serializable{
...
          @OneToMany(fetch = FetchType.LAZY, mappedBy = "customers")
          @Cascade(CascadeType.ALL)
          @Fetch(FetchMode.SELECT)
        @BatchSize(size = 10)
          public List<Address> getAddresses() {
                    return this.addresses;
          }
...
}
```

We can see that while defining One to Many relation using **@OneToMany** annotation, we have specified **fetch** attribute.

We have also specified **batch-size** as 10 using **@BatchSize** annotation which is required in Batch fetch mode.

We will use these 2 mapping files(XML and Annotation) to address each Fetch Mode as below

FetchMode JOIN

Modify above xml with fetch=”join”

(or)

Modify above annotated class with @Fetch(FetchMode.JOIN)

This is also called **Eager loading** which means **load** all the **collections** and **relations** no matter we use it or not.

This fetching strategy loads **User** and List of **Address** in a **single query** when we request only User.

It will load **all** the **addresses** when we load the **User** no matter whether we use the Address or notCopy this code

```
Customer  customer = session.get(Customer.class, customerId);
List<Address> addresses = customer.getAddresses();
```

Hibernate generated only one select statement, it retrieve all its related collections when the Customer is loaded using session.get(Customer.class, 1)

Select statement to retrieve the Customer records

Outer join to retrieve its related collections.

FetchMode SELECT(default)

Modify above xml with fetch=”select”

(or)

Modify above annotated class with @Fetch(FetchMode.SELECT)

This is also called Lazy loading which means loads the collections and relations only when required

It loads only Customer when we request the Customer, it will not load the addresses until we request for it.

It will load all the addresses only when we explicitly request the addresses to use in our application.

```
Customer  customer = session.get(Customer.class, customerId); //Select
from Customer table only

List<Address> addresses = customer.getAddresses();

//Select from Address table will start here
          for (Address address : addresses) {
                              System.out.println(address);
                    }
```

Hibernate generated 2 select statements

First Select statement to retrieve the Customer records – session.get(Customer.class, 1)

Second Select statement to retrieve its related collections – when we iterate addresses using enhanced for loop

FetchMode SELECT with Batch Size

No change in above xml as we have specified the batch size as 10 already

(or)

No change in above annotated class as we have specified the batch size as 10 already

Fetch upto “N” collections or entities(“Not number of records”)

The biggest misunderstanding of this fetch mode is, Batch size corresponds to number of records fetched per collection.
This is Complete misunderstanding.

`Batch size` decides the number of collections to be loaded when we load one entity.

Example:

We have given batch size as “**10**”

Now when we load **one Customer**, Hibernate loads the **address collection** for **additional 10 Customers** which are currently in the session.

It means **10 address collections** are loaded one for each Customer.

Suppose, We have **20** Customers in the session and **batch size** is set as **10**

In this case, when we load **one** Customer, **3** queries will be executed

1. One query to load **all** the **20** Customers.
2. One query to load the **Address collections** for **10** Customers.
3. Another query to load the **Address collections** for other **10** Customers

If we have only one User then queries generated with batch size is same as without batch size as below

Queries generated by Hibernate

```
List<Customer> customers = session.createQuery("from
Customer").getResultList();
        for (Customer customer : customers) {
                        List<Address> addresses =
customer.getAddresses();
                                for (Address address : addresses) {
                                        System.out.println(address);
                                }


                        }
Query query = session.createQuery("from Applicant ");

List list = query.list();
```