Hibernate JPA annotations :

Above Class

@Entity : to make the class as a table and variables as columns;
@Table(name="mystudents") :- if the table name and the class name to be different.
@NamedQuery(name="account.greaterThanBalance",query="from Account where balance >:bal")
@NamedNativeQuery(name="allAccount",query= "select * from account",resultClass=Account.class)

Above Variable

@Id :- to make a field as the ID field (to map with PK of a table)
@GeneratedValue(strategy=GenerationType.AUTO) -> roll will be generated automatically for each row.
GenratioTypes
AUTO :- internally underlaying ORM s/w creates a secquence or a table called "hibernate_sequence" to maintain the Id value.
IDENTITY :- it is used for auto_increatement feature to auto generate the id value
SEQUENCE :- it is used for sequence feature to auto generate the id value

@Column(name="sname") :- if the column name of table and corresponding variable of the class is diff.

Has-A relationship
@Embeded :- will be placed placed over a variable which is connected using HAS-A relationship. **(or)**
@Embedable :- it needs to be placed over the class.

```
@AttributeOverrides ( {
        @AttributeOverride(name="pinCode",column=@Column(name="schlPin")),
        @AttributeOverride(name="street",column=@Column(name="schlStreet")),
  })
```

List of Has-A relationship
@Embedded
@ElementCollection -> this is used over List<userdefinedClass > with has-A relationship. (default lazy loading)
@ElementCollection(fetch=FetchType.EAGER) -> eager load the HAS-A class.
@JoinTable(name="MynewTable",joinColumns=@JoinColumn(name="MyPK")) -> to change joinTable name & joincolumn PK column name.

OneToMany

@OneToMany -> this will create a one two Many association
@OneToMany(cascade = CascadeType.All) -> this will automatically persist both class object, no need to explicitly persist it.
@JoinTable(name="dept_emp",joinColumns=@JoinColumn(name="did"),inverseJoinColumns=@JoinColumn(name="eid"))

ManyToOne

@ManyToOne
@ManyToOne(cascade = CascadeType.All)
@ManyToOne(mappedBy="class var which will have many instance", cascade = CascadeType.All)

ManyToMany

@ManyToMany
@ManyToMany(cascade = CascadeType.All)
@ManyToMany(mappedBy=”var name”, cascade = CascadeType.All)

OneToOne

@OneToOne
@JoinColumn(name = “”)
@OneToOne(mappedBy = "dept")

Inheritance (Single table) over class

@Inheritance(strategy=InheritanceType.SINGLE_TABLE) -> by default single
@DiscriminatorColumn(name="emptype",discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue(value="emp") -> to change class Name in the table

Inheritance (Multiple Table) over class (does not have repeating variable only FK)

@Inheritance(strategy=InheritanceType.JOINED) -> will create table for all classes extended

@PrimaryKeyJoinColumn(name="eid") -> to be places over the child class to change FK col name

Inheritance (Multiple Table) over class (has repeating variables in every table)

@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)

MappedSuperclass:-
in this strategy parent class will not be an Entity, it will be a normal java class.

@MappedSuperclass -> placed over parent class

Hibernate JPA:

<persistence-unit name="account-unit" > -> this is to be loaded by EntityFactoryManager(“account-unit”);
<class>com.ratan.model.Account</class> -> this is the entity Path
<property name="hibernate.show_sql" value="true"/>
<property name="hibernate.hbm2ddl.auto" value="update (or) create"/>

1. What is ORM in Hibernate?

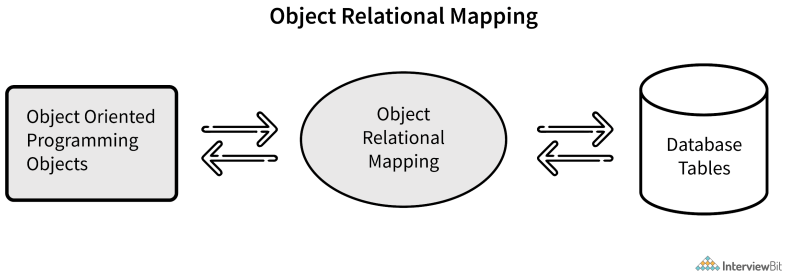
Hibernate ORM stands for **Object Relational Mapping**.

Object-Relational Mapping (ORM) is the process of converting Java objects to database tables

--the process of mapping java classes with the DB tables ,, java class member variables with the DB table columns and

making the object of java class reperesents the DB table records having synchronization btween them is called a ORM mapping.

the modification done in the obj will reflect the DB table row and vise-versa. This makes data retrieval, creation, and manipulation very simple as we dont need to atomic(simple data type) values.



--there are variaous ORM s/w are available in the market, these s/w will act as frame/w software to perform ORM based persistence logic.

ex:-
Hibernate
Toplink
Ibatis,

2. What is Hibernate? It is framework s/w to perform ORM logic

--it is a special type of s/w that provides abstraction layer on one or more existing core technology to simplyfy the process of application development.

--in java most of the f/w softwares comes in the form of jar files(one or more jar file)

--in order to use/work on these f/w softwares we need to add those jar files in our classpath.

--while working with the ORM based persistence logic we write all the logics in the form of objs without any sql quiery support. due to which our logic will become DB s/w independent logic.

--In ORM based logic, the ORM s/w takes objs as an input and gives objs as an output so no need to convert object data to the primitive values.

Object-Relational Mapping (ORM) is the process of converting Java objects to database tables

Java Persistence API (JPA) is a specification that defines how to persist data in Java applications.that it provides the flexibility to developer to change the implementation from one ORM to another (for example if application uses the JPA api and implementation is hibernate. In future it can switch to IBatis if required. But on the other if application directly lock the implementation with Hibernate without JPA platform, switching is going to be herculean task)

Hibernate: Its the implementation of **ORM concept** implementing **JPA specification (JPA is an API)**.

Java persistence:

--the process of saving/storing java objs state into the DB s/w is known as java persistence..

--for small application we can store business data (java object state) in the files using IO streams (serialization and deserailization approach).

--the logic that write to store java objs(which is holding business data) into the file using IO Streams is known as "IO stream based persistence logic".

--the logic that we write to store java objs data into the DB using JDBC is known as "jdbc based persistence logic"

the logic that we write to store java objs into the DB using ORM approach is called as ORM based persistence logic.

Miasmatches addressed in Hibernate (ORM)

1.inheritence mismatches / IS-A mismatch

2.Granularity mismatch / has-A mismatch

3. assotiation mismatch / table relation mismatch

limitation of JDBC based persistence logic:-

1.jdbc can't store the java **objs into the table directly,becoz sql queires does not allows the java objs as input**, here we need to convert obj data into the simple(atmoic) value to store them in a DB.

2.jdbc code is the **DB dependent** code becoz it uses DB s/w dependent queries.

3.jdbc code having **boiler plate code problem** (writing the same code except sql queries in multiple classes of our application again and again)..

4.jdbc code **throws lots of checked exceptions**, programmer need to handle them.

5.After the select operation, we get the **ResultSet object.this RS obj we can not transfer from one layer to another layer.**

6.there is **no any caching and transaction management** support is available in jdbc.

ORM s/w features:-

1.it can **persist/store java obj to the DB directly**.

2.it supports **POJO and POJl model**

3.it is a **lightweight s/w** becoz to excute the ORM based application we dont need install any kind of servers.

4.ORM persistance logic is **DB in-dependent**. it is portable accross multiple DB s/ws. (beocz here we deal with object, not with the sql queires)

5.**prevent the developers from boiler plate code** coding to perform CRUD operarions.

6.it generates fine tuned sql statements internally that improves the performance.

7.it **provides caching** mechanism (maintaining one local copy to inhance the performance)

8.it provides **implicit connection pooling**.

9.**exception handling is optional becoz it throws unchecked exceptions.**

10.it has a special Query language called **JPQL (JPA java persistence query language)** that totally depends upon the objects.

POJO class:- Plain old java object

--it is a normal java class not bounded with any technology or f/w s/ws.

i.e a java class that is not implementing or extending technology/framework api related classes or interfaces.

--a java class that can be compiled without adding any extra jar files in the classpath are known as a POJO class.

How to activate ORM engine

create a client application and activate ORM engine by using **JPA api** related following classes and interface and perform the DB operations.

1.Persistence class

2.EntityManagerFactory

3.EntityManager

--if we use **Hibernate core api** then we need to use

Configuration class

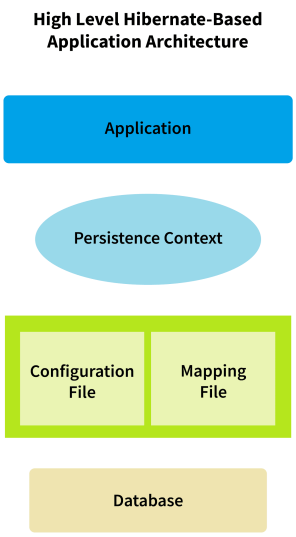
SessionFactory(I)

Session(I)

3. Explain Hibernate architecture

The Hibernate architecture consists of many objects such as a persistent object, session factory, session, query, transaction, etc. Applications developed using Hibernate is mainly categorized into 4 parts:

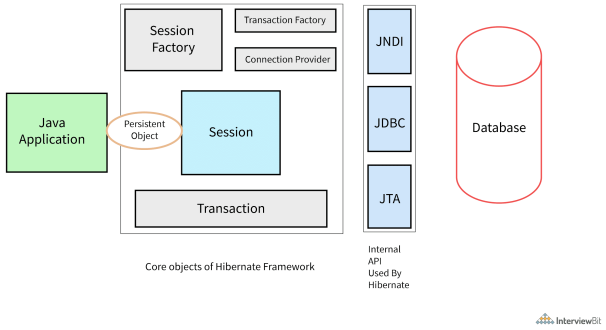
- Java Application
- Hibernate framework - Configuration and Mapping Files
- Internal API -
    - JDBC (Java Database Connectivity)
    - JTA (Java Transaction API)
    - JNDI (Java Naming Directory Interface).
- Database - MySQL, PostGreSQL, Oracle, etc



Hibernate Architecture

The main elements of Hibernate framework are:

- SessionFactory: This provides a factory method to get session objects and clients of ConnectionProvider. It holds a second-level cache (optional) of data.
- Session: This is a short-lived object that acts as an interface between the java application objects and database data.
    - The session can be used to generate transaction, query, and criteria objects.
    - It also has a mandatory first-level cache of data.
- Transaction: This object specifies the atomic unit of work and has methods useful for transaction management. This is optional.
- ConnectionProvider: This is a factory of JDBC connection objects and it provides an abstraction to the application from the DriverManager. This is optional.
- TransactionFactory: This is a factory of Transaction objects. It is optional.



4. What are the major advantages of Hibernate Framework?

- It is open-sourced and lightweight.
- Performance of Hibernate is very fast.
- Helps in generating database independant queries.
- Provides facilities to automatically create a table.
- It provides query statistics and database status.

5. What are the advantages of Hibernate over JDBC?

- The advantages of Hibernate over JDBC are listed below:

- **Clean Readable Code:** Using hibernate, helps in eliminating a lot of JDBC API-based boiler-plate codes, thereby making the code look cleaner and readable.
- **HQL (Hibernate Query Language):** Hibernate provides HQL which is closer to Java and is object-oriented in nature. This helps in reducing the burden on developers for writing database independent queries. In JDBC, this is not the case. A developer has to know the database-specific codes.
- **Transaction Management:** JDBC doesn't support implicit transaction management. It is upon the developer to write transaction management code using commit and rollback methods. Whereas, Hibernate implicity provides this feature.
- **Exception Handling:** Hibernate wraps the JDBC exceptions and throws unchecked exceptions like JDBCException or HibernateException. This along with the built-in transaction management system helps developers to avoid writing multiple try-catch blocks to handle exceptions. In the case of JDBC, it throws a checked exception called SQLException thereby mandating the developer to write try-catch blocks to handle this exception at compile time.
- **Special Features:** Hibernate supports OOPs features like inheritance, associations and also supports collections. These are not available in JDBC.

6. What are the different functionalities supported by Hibernate?

- Hibernate is an ORM tool.
- Hibernate uses Hibernate Query Language(HQL) which makes it database-independent.
- It supports auto DDL operations.
- This Java framework also has an Auto Primary Key Generation support.
- Supports cache memory.
- Exception handling is not mandatory in the case of Hibernate.

7. What are the technologies that are supported by Hibernate?

Hibernate supports a variety of technologies, like:

- XDoclet Spring
- Maven
- Eclipse Plug-ins
- J2EE

8. What are some of the important interfaces of Hibernate framework?

Hibernate core interfaces are:

- Configuration
- SessionFactory
- Session
- Criteria
- Query
- Transaction

9. What are the best practices to follow with Hibernate framework?

- Always check the primary key field access, if it's generated at the database layer then you should not have a setter for this.
- By default hibernate set the field values directly, without using setters. So if you want Hibernate to use setters, then make sure proper access is defined as @Access(value=AccessType.PROPERTY).
- If access type is property, make sure annotations are used with getter methods and not setter methods. Avoid mixing of using annotations on both filed and getter methods.
- Use native sql query only when it can't be done using HQL, such as using the database-specific feature.
- If you have to sort the collection, use ordered list rather than sorting it using Collection API.
- Use named queries wisely, keep it at a single place for easy debugging. Use them for commonly used queries only. For entity-specific query, you can keep them in the entity bean itself.
- For web applications, always try to use JNDI DataSource rather than configuring to create a connection in hibernate.
- Avoid Many-to-Many relationships, it can be easily implemented using bidirectional One-to-Many and Many-to-One relationships.
- For collections, try to use Lists, maps and sets. Avoid array because you don't get benefit of lazy loading.
- Do not treat exceptions as recoverable, roll back the Transaction and close the Session. If you do not do this, Hibernate cannot guarantee that the in-memory state accurately represents the persistent state.
- Prefer DAO pattern for exposing the different methods that can be used with entity bean
- Prefer lazy fetching for associations

10. How to achieve mapping in Hibernate?

Association mappings are one of the key features of Hibernate. It supports the same associations as the relational database model. They are:

- One-to-One associations
- Many-to-One associations

- Many-to-Many associations

You can map each of them as a uni- or bidirectional association.

11. What is HQL?

HQL is the acronym of Hibernate Query Language. It is an Object-Oriented Query Language and is independent of the database.

12. Name some of the important interfaces of Hibernate framework?

Hibernate interfaces are:

- **SessionFactory** (org.hibernate.SessionFactory)
- **Session** (org.hibernate.Session)
- **Transaction** (org.hibernate.Transaction)

13. What is a Session in Hibernate?

Hibernate Session is the interface between Java application layer and Hibernate. It is used to get a physical connection with the database. The Session object created is lightweight and designed to be instantiated each time an interaction is needed with the database. This Session provides methods to create, read, update and delete operations for a constant object. To get the Session, you can execute HQL queries, SQL native queries using the Session object.

14. What is a SessionFactory?

SessionFactory is the factory class that is used to get the Session objects. The SessionFactory is a heavyweight object so usually, it is created during application startup and kept for later use. This SessionFactory is a thread-safe object which is used by all the threads of an application. If you are using multiple databases then you would have to create multiple SessionFactory objects.

15. What is the difference between openSession and getCurrentSession?

getCurrentSession()	**openSession()**
This method returns the session bound to the context.	This method always opens a new session.

This session object scope belongs to the hibernate context and to make this work hibernate configuration file has to be modified by adding **<property name = "hibernate.current_session_context_class"> thread </property>**. If not added, then using the method would throw an HibernateException.	A new session object has to be created for each request in a multi-threaded environment. Hence, you need not configure any property to call this method.
This session object gets closed once the session factory is closed.	It's the developer's responsibility to close this object once all the database operations are done.
In a single-threaded environment, this method is faster than openSession().	In single threaded environment, it is slower than getCurrentSession()single-thread eda

16. What is One-to-One association in Hibernate?

In this type of mapping, you only need to model the system for the entity for which you want to navigate the relationship in your query or domain model. You need an entity attribute that represents the association, so annotate it with an @OneToOne annotation.

17. What is One-to-Many association in Hibernate?

In this type of association, one object can be associated with multiple/different objects. Talking about the mapping, the One-to-Many mapping is implemented using a Set Java collection that does not have any redundant element. This One-to-Many element of the set indicates the relation of one object to multiple objects.

18. What is Many-to-Many association in Hibernate?

Many-to-Many mapping requires an entity attribute and a @ManyToMany annotation. It can either be unidirectional and bidirectional. In **Unidirectional**, the attributes model the association and you can use it to navigate it in your domain model or JPQL queries. The annotation tells Hibernate to map a Many-to-Many association. The **bidirectional** relationship, mapping allows you to navigate the association in both directions.

19. What can you tell about Hibernate Configuration File?

Hibernate Configuration File or **hibernate.cfg.xml** is one of the most required configuration files in Hibernate. By default, this file is placed under the src/main/resource folder.

The file contains database related configurations and session-related configurations.

Hibernate facilitates providing the configuration either in an XML file (like hibernate.cfg.xml) or a properties file (like hibernate.properties).

This file is used to define the below information:

- Database connection details: Driver class, URL, username, and password.
- There must be one configuration file for each database used in the application, suppose if we want to connect with 2 databases, then we must create 2 configuration files with different names.
- Hibernate properties: Dialect, show_sql, second_level_cache, and mapping file names.

20. What do you mean by Hibernate configuration file

--it is an xml file its name is "persistence.xml".

-this file must be created under src/META-INF folder in normal java application, where as in maven or gradle based application this file should be inside the src/main/resources/META-INF folder

-this file content will be used by ORM s/w (ORM engine) to locate the destination DB s/w.

--in this file generally 3 types of details we specify:-

1.DB connection details

2.ORM specific details (some instruction to the ORM s/w like dialect info,show_sql, second_level_cache, and mapping file names) Dialect **allows Hibernate to generate SQL optimized for a particular relational database**

3. annotation based entity/persistence class name.

4. There must be one configuration file for each database used in the application, suppose if we want to connect with 2 databases, then we must create 2 configuration files with different names.

check

<persistence> with some xml-namespace

--the child tag of <persistence> tag is <persistence-unit>

--this <persistence-unit> has 2 child tags:-

1. **<class> tag**:-using which we specify the Entity class name(fully qualified name) that used

annotations to map a table

2.**<properties> tag** :- using this tag,we specify some configuration details to the ORM s/w

Persistence-unit:- it is a collection of Entity/Persistence class instance reffered by a unique user-defiend name

ORM engine :

--it is a specialized s/w written in java that performs translation of JPA calls into the sql call by using mapping annotation and configuration file details and send the mapped sql to the DB s/w using JDBC.

--ORM engine is provided by any **ORM s/w.**

JPA application ----------->EntityManager ---------->ORM engine ------>JDBC---------->DB s/w

What is Persistence-Unit in persistence.xml?

A persistence unit defines a set of all entity classes that are managed by EntityManager instances in an application.

How to load persistence-unit in ORM engine?

Step - 1 : EntityManagerFactory emf=Persistence.createEntityManagerFactory("studentUnit");

--this method loads the "persistence.xml" file into the memory

--EntityManagerFactory obj should be only one per application per DB.

–This contains connection pool & some META information.

Connection pooling is **a technique of creating and managing a pool of connections that are ready for use by any thread that needs them**. Connection pooling can greatly increase the performance of your Java application, while reducing overall resource usage as we dont need to open and close connections manually.

Step : 2 EntityManager em= emf.createEntityManager();

Note:- inside every DAO method we need to get the EntityManager obj

Steps to persist :-

Step : 3 In order to perform any DML (insert update delete) the method calls should be in a transactional area.

- **em.getTransaction().begin();** (begins the transaction area)
- **em.persist(st); - >** (to persist or store in database)
- **em.getTransaction().commit();** (does flush the session, but it also ends the unit of work.)

this EntityTransaction obj is a singleton object, i.e per EntityManager obj, only one Transaction object is created.

Step : 3

em.find(Student.class, 60) -> (to find obj)

em.remove(student); -> (to remove)

Life-cycle of persistence/entity object:-

an entity has the 3 life-cycle state:-

1.new state/transient state

2.persistence state/managed state

3.detached state

1.new state/transient state:-

--if we create a object of persistence class and this class is not attached with the EM obj, then

this stage is known as new state/transient state

Student s=new Student(10,"ram",780);

2.persistence state:-

--if a persistence class obj is associated with EM obj, then this obj will be in persistence state.

ex:-

when we call a persist(-) method by supplying Student entity obj then at time student obj will be in persistence state

or

when we call find() method and this method returns the Student obj, then that obj will also be in persistence state.

Note:- when an entity class obj is in persisitence state ,it is in-sync with the DB table ,i.e

any change made on that obj inside the tx area will reflect to the table automatically.

ex:-

```
Student s=new Student(150,"manoj",850); // here student obj is in transient state ;

em.getTransaction().begin();

em.persist(s); // here it is in the persistence state

s.setMarks(900);

em.getTransaction().commit();
```

detached state:-

--when we call close() method or call **clear()** method on the EM obj, then all the associated entity obj will be in detached state.

--in this stage the entity objs will not be in-sync with the table.

Note:- we have a **merge()** method in EM obj, when we call this method by supplying any detached object then that detaced object will bring back in the persistence state.

limitation of EM methods in performing CRUD operations:-

1.Retrieving Entity obj based on only ID field(PK field) @Id

2.multiple Entity obj retrieval is not possible (multiple record)

3.bulk update and bulk delete is also not possible

4.to access Entity obj we can not specify some extra condition.

--to overcome the above limitation JPA has provided JPQL (java persistence query language).

diff bt JPQL and sql:-

- sql queries are expressed in the term of table and columns, where as jpql query is expressed in the term of Entity class names and its variables.
- the name of the class and its variables are case sensitive.
- sql is not portable across multiple dbms, where jpql is portable.

Step : 4

To get list of data

```
//String jpql= "select a from Account a";
Query q= em.createQuery(jpql);
List<Account> list= q.getResultList();
```

To get object by non-PK

```
//String jpql= "select a from Account a where a.name='Ram' ";
Query q= em.createQuery(jpql);
List<Account> list= q.getResultList();
```

if we confirm that only one row will come then :-

```
//String jpql = ” select a from Account a where a.name='Ram' ";
Query q= em.createQuery(jpql);
Object ob= (Object)q.getSingleResult();
```

- if the above query will return more that one result then it will throw a runtime exception
- in order to avoid the downcasting problem we should use TypedQuery instead of Query obj.
- If we want to update delete we can use Query (or) if returns anything use TypesQeruy

```
TypedQuery<Account> q= em.createQuery(jpql,Account.class);
Account acc = q.getSingleResult();
```

bulk update:-

```
String jpql= "update Account set balance=balance+500";
Query q= em.createQuery(jpql);

em.getTransaction().begin();
int x= q.executeUpdate();   //will provide number of rows affected//
em.getTransaction().commit();
System.out.println(x+" row updated...");
```

using positional parameter:-

```
String jpql= "update Account set balance=balance+?5 where name=?6";
Query q= em.createQuery(jpql);
q.setParameter(5, 1000);
q.setParameter(6, "Amit");
```

using named parameter

```
String jpql= "update Account set balance=balance+:bal where name=:nm";
Query q= em.createQuery(jpql);
q.setParameter(“bal”, 1000);
q.setParameter(“nm”, "Amit");
```

for 1 row and 1 column:-

- EntityManager em= EMUtil.provideEntityManager();
- String jpql= "select name from Account where accno=:ano";
- TypedQuery<String> q=em.createQuery(jpql,String.class);
- q.setParameter("ano", 4);
- String n= q.getSingleResult();
- System.out.println(n);

ex: multiple row and 1 column:-

- EntityManager em= EMUtil.provideEntityManager();
- String jpql= "select balance from Account";
- TypedQuery<Integer> q=em.createQuery(jpql,Integer.class);
- List<Integer> list= q.getResultList();
- System.out.println(list);

few column and all rows:-

EntityManager em= EMUtil.provideEntityManager();

- String jpql= "select name,balance from Account";
- TypedQuery<Object[]> q=em.createQuery(jpql,Object[].class);
- List<Object[]> list= q.getResultList();
- for(Object[] or:list){
  - String name=(String)or[0];
  - System.out.println("Name is "+name.toUpperCase());
  - System.out.println("Balance is "+or[1]);
  - System.out.println("------------------------"); }

Aggregrate function:-

--any aggregrate function will return :-

- min,max: Integer
- avg : Double
- sum : Long

Ex:-

- EntityManager em= EMUtil.provideEntityManager();
- String jpql= "select sum(balance) from Account";
- TypedQuery<Long> q=em.createQuery(jpql,Long.class);
- long result= q.getSingleResult();
- System.out.println(result);

Named Query:

@NamedQuery(name="account.greaterBalance",query="from Account where balance >:bal") -> **placed above class**

- TypedQuery<Account>tq=em.createNamedQuery("account.greaterBalance",Account.class);
- tq.setParameter("bal", 5000);
- List<Account> list= tq.getResultList();
- list.forEach(a -> System.out.println(a));

NativeQuery (normal SQL query in terms of tables and column)

- String nq="select * from account"; //here account is the table name
- Query q= em.createNativeQuery(nq, Account.class);
- List<Account> list= q.getResultList();
- list.forEach(a -> System.out.println(a));

NamedNativeQuery:-

@NamedNativeQuery(name="allAccount",query= "select * from account",resultClass=Account.class)

- Query q= em.createNamedQuery("allAccount");
- List<Account> list= q.getResultList();

Mismatched bt Object Oriented Representation and relational representaion of data:

- 1.granularity mismatch :- HAS-A relationship problem
- 2.Association Mismatch :- table relationship problem
- 3.inheritence mismatch :- IS-A relationship problem

HAS-A relationship problem (or granularity)

Approach-1 :

Condition 1-> with one Has-A variable

@Embeddable at the top of Address class or

@Embedded at the top of Address addr variable inside the Employee Entity.

Note : This will add all the coarse class tables(address class) inside Entity class

Condition 2 -> with two Has-A variable in same type

If we have two Has-A with same class(ex HomeAdress, officeAddress for address class)

@AttributeOverrides ({

@AttributeOverride(name="state",column=@Column(name="HOME_STATE")),

@AttributeOverride(name="city",column=@Column(name="HOME_CITY")),

})

Condition 3 -> with List of Has-A variable

- If any class has more than one HAS-A relationship of same type then it will violate the rule of normalization.
- To solve this problem we have to use **@ElementCollection** annotation and add to collection classes
- in this case ORM s/w will generate a seperate table to maintain the addresses details with a Foreign key that reffers the PK of Employee table.

eager and lazy loading:-

--by default ORM s/w perform lazy loading while fetching the objs, when we fetch the parent obj(first level obj),then only the first level obj related data will be loaded into the memory,but the 2nd level obj related data will be loaded at time of calling the 2nd level object related methods.

Association Mismatch:- table relationship problem:-

--at the table level different types of tables will participate in different kind of relationships. To establish these relationship entity classes also must be in a relationship.

- 1.one to one (person ----- Driving licence) :- PK and FK(unique)
- 2.one to many (Dept ----Emp) :- PK and FK (i.e PK of Dept will be inside the Emp as FK)
- 3.many to many (student --- course) :- we need to take the help of 3rd table(linking table)

JPA supports the relationship between the Entity classes not only with the cardinality but also with the direction.

Uni-directional :

- we can define parent class inside child class (or) child class inside parent class. Both are not possible at same time.
- We can access child class from parent class (or) pracent class from child class but not both.

Bi-directional :

- we define child Entity obj inside the parent Entity and parent Entity obj inside the child Entity,(navigation is possible from the either one of the any obj)

JPA supports:

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

One-many

@OneToMany -> this will create one to many relationship in table.

Note: don't forget we need to persist both the entity objects. This will create two object tables with one linking table.

If we want to persist when one class with Many objects?

@OneToMany(cascade= CascadeType.ALL)

Many-One (unidirectional)

Here we only annotate Employee class dept var with ManyToOne

@ManyToOne(cascade=CascadeType.All) -> this will create many to one relationship

Note : this will not have a third table, instead the Many class ex.Employye will have FK which will refer to the Department primary key. (Ex. Many Employees will have one Department.)

We can find only with Employee object

Many-One (bi-directional)

Here we need to combine above both approach, inside dept class we need to take List<Emp> and inside dept class variable with annotations

Dept class	Emp class
List<Emp (@oneToMany)	Dept (@ManyToOne)

Here 3rd table will be created

We can find with both the classes.

If we dont want to use third table? Because we already will have relationship Fk in employee table. So use

- **@OneToMany(mappedBy="dept" ,cascade=CascadeType.ALL)**
- **private List<Employee> emps=new ArrayList<Employee>();**

ManyToMany

We need to apply @ManyToMany annotation over the list. In this both the class will contain List of objects.

When we use @ManyToMany we will get 4-linking tables. To avoid this we need to use Mapped by in any of the annotation

@ManyToMany(cascade = CascadeType.ALL,mappedBy = "deptList") -> deptList is the var name.

This is Bi-direstional we can get objects of both with find method.

OneToOne

Assume one department has only one employee and one employee belongs from only one dept -> we can use @OneToOne

here 2 table will be created

1. employee (empid,name,salary)
2. department(did,dname,dlocation, emp_empid) (this emp_empid will be the FK)

--if we want to change this auto generated FK column name then we need to apply

- @OneToOne
- @JoinColumn(name="eid")
- private Employee emp

Note:- in the above application, we can create another dept and add the same emp again which is working in another dept, but it seems like a one to many relation,but it will be on the table level,at object level it is not an OTM, becoz we don't have List,and more ever we can not navigate from employee table to Department table.

onetoone bidirectional :-

here on both side define opposit class variables:-

ex:-

Department:-

- @OnetoOne
- private Employee emp

Employee:-

- @OneToOne
- private Department dept

--in this case 2 table will be created both will contains the id of each other as FK as an extra column.

--if we want that only one table should maintain the FK col then we use mappedBy on any side.

Department:-

- @OneToOne(mappedBy = "dept")
- private Employee emp;

Inheritence Mapping

one table for entire hirarchy/Single table.

@Inheritence(strategy=InheritenceType.SINGLE_TABLE)

This will create everything in a single table.

@DiscriminatorColumn(name="emptype",discriminatorType=DiscriminatorType.STRING)

This wii change the DTYPE column name to given name.

@DiscriminatorValue(value="emp")

This will change the name of the class in table.

Table per sub-classes strategy/Joined Table:-

Adv of table per subclasses strategy:-

1. DB tables can be designed by satisfying normalization forms/rules.
2. no need to take any discriminator value.
3. not null constraint can be applied.

@Inheritance(strategy=InheritanceType.JOINED) -> above parent class.

21. What are the key components of a Hibernate configuration object?

The configuration provides 2 key components, namely:

- Database Connection: This is handled by one or more configuration files.
- Class Mapping setup: It helps in creating the connection between Java classes and database tables.

23. How to integrate Hibernate and Spring?

Spring is also one of the most commonly used Java frameworks in the market today. Spring is a JavaEE Framework and Hibernate is the most popular ORM framework. This is why Spring Hibernate combination is used in a lot of enterprise applications.

1. Add Hibernate-entity manager, Hibernate-core and Spring-ORM dependencies.
2. Create Model classes and corresponding DAO implementations for database operations. The DAO classes will use SessionFactory that will be injected by the Spring Bean configuration.
3. Note that you don't need to use Hibernate Transaction Management, as you can leave it to the Spring declarative transaction management using @Transactional annotation.

24. What do you think about the statement - "session being a thread-safe object"?

No, Session is not a thread-safe object which means that any number of threads can access data from it simultaneously.

25. Mention some important annotations used for Hibernate mapping?

Hibernate supports JPA annotations. Some of the major annotations are:

1. **javax.persistence.Entity:** This is used with model classes to specify they are entity beans.
2. **javax.persistence.Table:** It is used with entity beans to define the corresponding table name in the database.
3. **javax.persistence.Access:** Used to define the access type, field or property. The default value is field and if you want Hibernate to use the getter/setter methods then you need to set it to a property.
4. **javax.persistence.Id:** Defines the primary key in the entity bean.
5. **javax.persistence.EmbeddedId:** It defines a composite primary key in the entity bean.
6. **javax.persistence.Column:** Helps in defining the column name in the database table.
7. **javax.persistence.GeneratedValue:** It defines the strategy to be used for the generation of the primary key. It is also used in conjunction with javax.persistence.GenerationType enum.
8. javax.persistence.OneToOne: "@OneToOne" is used for defining the one-to-one mapping between two bean entities. Similarly, hibernate provides OneToMany, ManyToOne and ManyToMany annotations for defining different mapping types.
9. org.hibernate.annotations.Cascade: "@Cascade" annotation is used for defining the cascading action between two bean entities. It is used with org.hibernate.annotations.CascadeType enum to define the type of cascading.

26. What is the difference between first level cache and second level cache?

Hibernate has 2 cache types. First level and second level cache for which the difference is given below:

First Level Cache	Second Level Cache

This is local to the Session object and cannot be shared between multiple sessions.	This cache is maintained at the SessionFactory level and shared among all sessions in Hibernate.
This cache is enabled by default and there is no way to disable it.	This is disabled by default, but we can enable it through configuration.
The first level cache is available only until the session is open, once the session is closed, the first level cache is destroyed.	The second-level cache is available through the application's life cycle, it is only destroyed and recreated when an application is restarted.

If an entity or object is loaded by calling the get() method then Hibernate first checked the first level cache, if it doesn't find the object then it goes to the second level cache if configured. If the object is not found then it finally goes to the database and returns the object, if there is no corresponding row in the table then it returns null.

27. Discuss the Collections in Hibernate

Hibernate provides the facility to persist the Collections. A Collection basically can be a List, Set, Map, Collection, Sorted Set, Sorted Map. java.util.List, java.util.Set, java.util.Collection, etc, are some of the real interface types to declared the persistent collection-value fields. Hibernate injects persistent Collections based on the type of interface. The collection instances generally behave like the types of value behavior.

28. What are the collection types in Hibernate?

There are five collection types in hibernate used for one-to-many relationship mappings.

- Bag
- Set
- List
- Array
- Map

29. How do you create an immutable class in hibernate?

Immutable class in hibernate creation could be in the following way. If we are using the XML form of configuration, then a class can be made immutable by markingmutable=false. The default value is true there which indicating that the class was not created by default.

In the case of using annotations, immutable classes in hibernate can also be created by using @Immutable annotation.

30. Can you explain the concept behind Hibernate Inheritance Mapping?

Java is an Object-Oriented Programming Language and Inheritance is one of the most important pillars of object-oriented principles. To represent any models in Java, inheritance is most commonly used to simplify and simplify the relationship. But, there is a catch. Relational databases do not support inheritance. They have a flat structure.

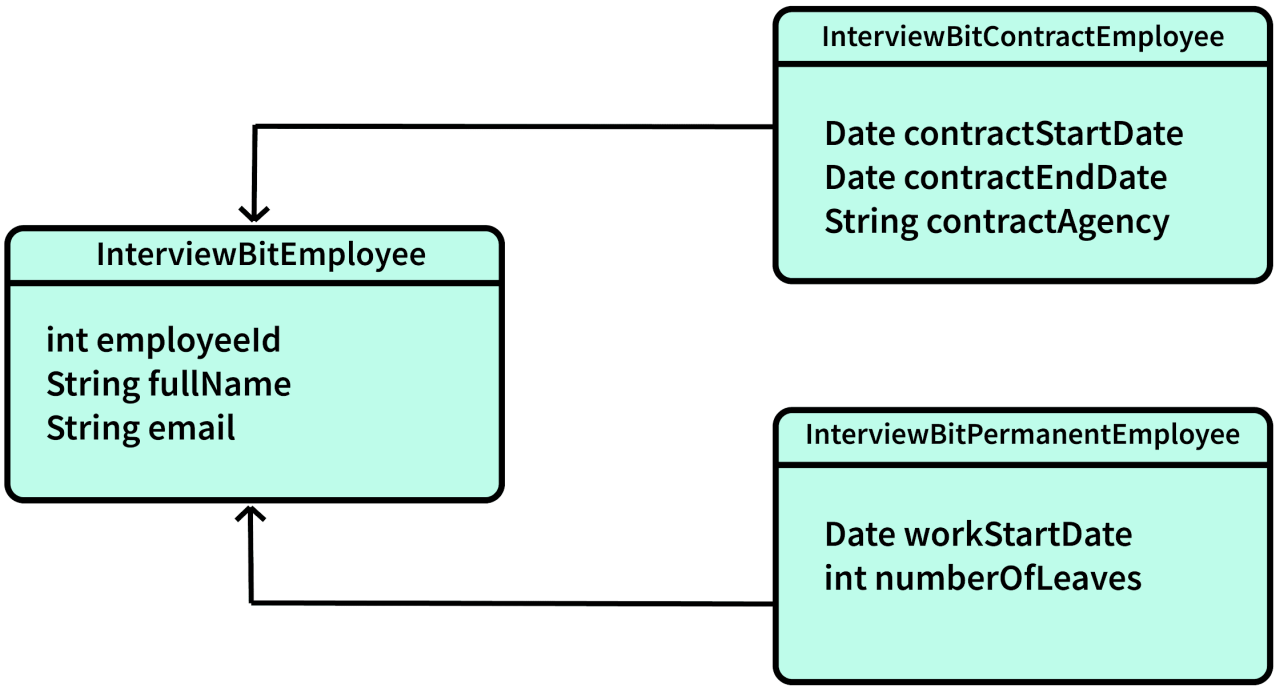
Hibernate's Inheritance Mapping strategies deal with solving how to hibernate being an ORM tries to map this problem between the inheritance of Java and flat structure of Databases.

Consider the example where we have to divide InterviewBitEmployee into Contract and Permanent Employees represented by IBContractEmployee and IBPermanentEmployee classes respectively. Now the task of hibernate is to represent these 2 employee types by considering the below restrictions:

The general employee details are defined in the parent InterviewBitEmployee class.

Contract and Permanent employee-specific details are stored in IBContractEmployee and IBPermanentEmployee classes respectively

The class diagram of this system is as shown below:



InterviewBit

Hibernate's Inheritance Mapping

There are different inheritance mapping strategies available:

- Single Table Strategy
- Table Per Class Strategy
- Mapped Super Class Strategy
- Joined Table Strategy

31. Is hibernate prone to SQL injection attack?

SQL injection attack is a serious vulnerability in terms of web security wherein an attacker can interfere with the queries made by an application/website to its database thereby allowing the attacker to view sensitive data which are generally irretrievable. It can also give the attacker to modify/ remove the data resulting in damages to the application behavior.

Hibernate does not provide immunity to SQL Injection. However, following good practices avoids SQL injection attacks. It is always advisable to follow any of the below options:

- Incorporate Prepared Statements that use Parameterized Queries.
- Use Stored Procedures.
- Ensure data sanity by doing input validation.

32. What is a Hibernate Template class?

When you integrate Spring and Hibernate, Spring ORM provides two helper classes – HibernateDaoSupport and HibernateTemplate. The main reason to use them was to get two things, the Session from Hibernate and Spring Transaction Management. However, from Hibernate 3.0.1, you can use the SessionFactory getCurrentSession() method to get the current session. The major advantage of using this Template class is the **exception translation** but that can be achieved easily by using @Repository annotation with service classes.

33. What are the benefits of using Hibernate template?

The following are the benefits of using this Hibernate template class:

- Automated Session closing ability.
- The interaction with the Hibernate Session is simplified.
- Exception handling is automated.

34. Define Hibernate Validator Framework

Data validation is considered as an integral part of any application. Also, data validation is used in the presentation layer with the use of Javascript and the server-side code before processing. It occurs before persisting it in order to make sure it follows the correct format. Validation is a cross-cutting task, so we should try to keep it apart from the business logic. This Hibernate Validator provides the reference implementation of bean validation specs.

35. What is Dirty Checking in Hibernate?

Hibernate incorporates Dirty Checking feature that permits developers and users to avoid time-consuming write actions. This Dirty Checking feature changes or updates fields that need to be changed or updated, while keeping the remaining fields untouched and unchanged.

36. How can you share your views on mapping description files?

- Mapping description files are used by the Hibernate to configure functions.
- These files have the ***.hbm** extension, which facilitates the mapping between database tables and Java class.
- Whether to use mapping description files or not this entirely depends on business entities.

37. What is meant by Light Object Mapping?

The means that the syntax is hidden from the business logic using specific design patterns. This is one of the valuable levels of ORM quality and this Light Object Mapping approach can be successful in case of applications where there are very fewer entities, or for applications having data models that are metadata-driven.

38. What is meant by Hibernate tuning?

Optimizing the performance of Hibernate applications is known as Hibernate tuning.

The performance tuning strategies for Hibernate are:

1. SQL Optimization
2. Session Management
3. Data Caching

39. What is Transaction Management in Hibernate? How does it work?

Transaction Management is a property which is present in the Spring framework. Now, what role does it play in Hibernate?

Transaction Management is a process of managing a set of commands or statements. In hibernate, Transaction Management is done by transaction interface. It maintains abstraction from the transaction implementation (JTA, JDBC). A transaction is associated with Session and is instantiated by calling session.beginTransaction().

40. What are the different states of a persistent entity?

It may exist in one of the following 3 states:

- Transient: This is not associated with the Session and has no representation in the database.

- Persistent: You can make a transient instance persistent by associating it with a Session.

- Detached: If you close the Hibernate Session, the persistent instance will become a detached instance.

41. Which are the design patterns that are used in Hibernate framework?

There are a few design patterns used in Hibernate Framework, namely:

- Domain Model Pattern: An object model of the domain that incorporates both behavior as well as data.
- Data Mapper: A layer of the map that moves data between objects and a database while keeping it independent of each other and the map itself.
- Proxy Pattern: It is used for lazy loading.
- Factory Pattern: Used in SessionFactory.

42. Explain about Hibernate Proxy and how it helps in Lazy loading?

- Hibernate uses a proxy object in order to support Lazy loading.
- When you try loading data from tables, Hibernate doesn't load all the mapped objects.
- After you reference a child object through getter methods, if the linked entity is not present in the session cache, then the proxy code will be entered to the database and load the linked object.
- It uses Java assist to effectively and dynamically generate sub-classed implementations of your entity objects.

43. How can we see Hibernate generated SQL on console?

In order to view the SQL on a console, you need to add following in Hibernate configuration file to enable viewing SQL on the console for debugging purposes:

```
1
```
<property name="show_sql">true</property>

44. What is Query Cache in Hibernate?

Hibernate implements a separate cache region for queries resultset that integrates with the Hibernate second-level cache. This is also an optional feature and requires a few more steps in code.

45. What is the benefit of Native SQL query support in Hibernate?

Hibernate provides an option to execute Native SQL queries through the use of the SQLQuery object. For normal scenarios, it is however not the recommended approach because you might lose other benefits like Association and Hibernate first-level caching.

Native SQL Query comes handy when you want to execute database-specific queries that are not supported by Hibernate API such query hints or the Connect keyword in Oracle Database.

50. What is Named SQL Query?

Hibernate provides another important feature called Named Query using which you can define at a central location and use them anywhere in the code.

You can create named queries for both HQL as well as for Native SQL. These Named Queries can be defined in Hibernate mapping files with the help of JPA annotations @NamedQuery and @NamedNativeQuery.

51. When do you use merge() and update() in Hibernate?

This is one of the tricky Hibernate Interview Questions asked.

update(): If you are sure that the Hibernate Session does not contain an already persistent instance with the same id .

merge(): Helps in merging your modifications at any time without considering the state of the Session.

52. Difference between the transient, persistent and detached state in Hibernate?

Transient state: New objects are created in the Java program but are not associated with any Hibernate Session.

Persistent state: An object which is associated with a Hibernate session is called Persistent object. While an object which was earlier associated with Hibernate session but currently it's not associate is known as a detached object. You can call save() or persist() method to store those object into the database and bring them into the Persistent state.

Detached state: You can re-attach a detached object to Hibernate sessions by calling either update() or saveOrUpdate() method.

53. Difference between managed associations and Hibernate associations?

Managed associations: Relate to container management persistence and are bi-directional.

Hibernate Associations: These associations are unidirectional.

54. Differentiate between save() and saveOrUpdate() methods in hibernate session.

Both the methods save records to the table in the database in case there are no records with the primary key in the table. However, the main differences between these two are listed below:

save()	saveOrUpdate()
save() generates a new identifier and INSERT record into a database	Session.saveOrUpdate() can either INSERT or UPDATE based upon existence of a record.

The insertion fails if the primary key already exists in the table.	In case the primary key already exists, then the record is updated.
The return type is Serializable which is the newly generated identifier id value as a Serializable object.	The return type of the saveOrUpdate() method is void.
This method is used to bring only a transient object to a persistent state.	This method can bring both transient (new) and detached (existing) objects into a persistent state. It is often used to re-attach a detached object into a Session

Clearly, saveOrUpdate() is more flexible in terms of use but it involves extra processing to find out whether a record already exists in the table or not.

55. Differentiate between get() and load() in Hibernate session

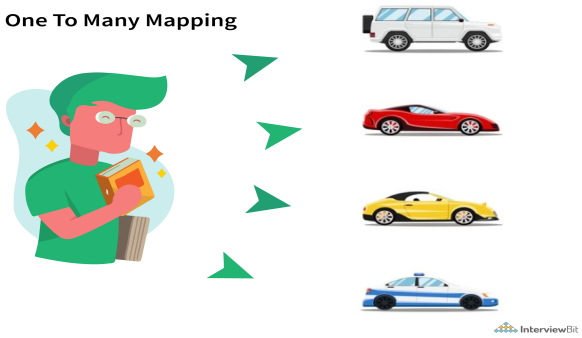
These are the methods to get data from the database. The primary differences between get and load in Hibernate are given below:

get()	load()
This method gets the data from the database as soon as it is called.	This method returns a proxy object and loads the data only when it is required.
The database is hit every time the method is called.	The database is hit only when it is really needed and this is called Lazy Loading which makes the method better.
The method returns null if the object is not found.	The method throws ObjectNotFoundException if the object is not found.
This method should be used if we are unsure about the existence of data in the database.	This method is to be used when we know for sure that the data is present in the database.

56. Can you tell something about one to many associations and how can we use them in Hibernate?

The **one-to-many association** is the most commonly used which indicates that one object is linked/associated with multiple objects.

For example, one person can own multiple cars.



In Hibernate, we can achieve this by using @OnetoMany of JPA annotations in the model classes. Consider the above example of a person having multiple cars as shown below:

```
@Entity

@Table(name="Person")

public class Person {


  //...


  @OneToMany(mappedBy="owner")

  private Set<Car> cars;


  // getters and setters

}
```

In the Person class, we have defined the car's property to have @OneToMany association. The Car class would have owned property that is used by the mappedBy variable in the Person class. The Car class is as shown below:

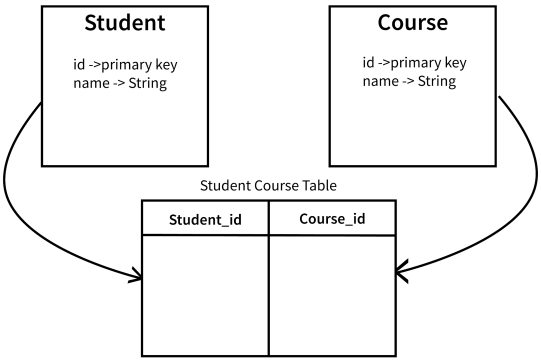
```
@Entity

@Table(name="Car")

public class Car {


  // Other Properties


  @ManyToOne

  @JoinColumn(name="person_id", nullable=false)

  private Person owner;


  public Car() {}


  // getters and setters

}
```

@ManyToOne annotation indicates that many instances of an entity are mapped to one instance of another entity – many cars of one person.

57. What are Many to Many associations?

Many-to-many association indicates that there are multiple relations between the instances of two entities. We could take the example of multiple students taking part in multiple courses and vice versa.

Since both the student and course entities refer to each other by means of foreign keys, we represent this relationship technically by creating a separate table to hold these foreign keys.



InterviewBit Many To Many Associations

Here, Student-Course Table is called the Join Table where the student_id and course_id would form the composite primary key.

58 What does session.lock() method in hibernate do?

session.lock() method is used to reattach a detached object to the session. **session.lock()** method does not check for any data synchronization between the database and the object in the persistence context and hence this reattachment might lead to loss of data synchronization.

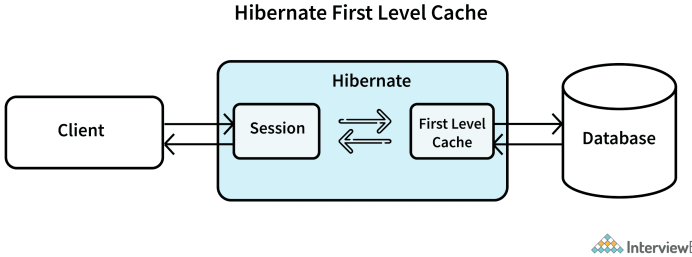
59. What is hibernate caching?

Hibernate caching is the strategy for improving the application performance by pooling objects in the cache so that the queries are executed faster. Hibernate caching is particularly useful when fetching the same data that is executed multiple times. Rather than hitting the database, we can just access the data from the cache. This results in reduced throughput time of the application.

Types of Hibernate Caching

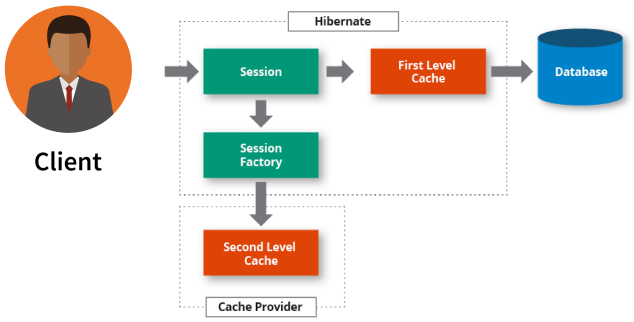
First Level Cache:

- This level is enabled by default.
- The first level cache resides in the hibernate session object.
- Since it belongs to the session object, the scope of the data stored here will not be available to the entire application as an application can make use of multiple session objects.



Second Level Cache:

- Second level cache resides in the SessionFactory object and due to this, the data is accessible by the entire application.
- This is not available by default. It has to be enabled explicitly.
- EH (Easy Hibernate) Cache, Swarm Cache, OS Cache, JBoss Cache are some example cache providers.



InterviewBit

60. When is merge() method of the hibernate session useful?

Merge() method can be used for updating existing values. The specialty of this method is, once the existing values are updated, the method creates a copy from the entity object and returns it. This result object goes into the persistent context and is then tracked for any changes. The object that was initially used is not tracked.

61. Collection mapping can be done using One-to-One and Many-to-One Associations. What do you think?

False, collection mapping is possible only with **One-to-Many** and **Many-to-Many associations.**

62. Can you tell the difference between setMaxResults() and setFetchSize() of Query?

setMaxResults() the function works similar to LIMIT in SQL. Here, we set the maximum number of rows that we want to be returned. This method is implemented by all database drivers.

setFetchSize() works for optimizing how Hibernate sends the result to the caller for example: are the results buffered, are they sent in different size chunks, etc. This method is not implemented by all the database drivers.

63. Does Hibernate support Native SQL Queries?

Yes, it does. Hibernate provides the createSQLQuery() method to let a developer call the native SQL statement directly and returns a Query object.

Consider the example where you want to get employee data with the full name “Hibernate”. We don’t want to use HQL-based features, instead, we want to write our own SQL queries. In this case, the code would be:

```
Query query = session.createSQLQuery( "select * from interviewbit_employee ibe where ibe.fullName = :fullName")

        .addEntity(InterviewBitEmployee.class)

        .setParameter("fullName", "Hibernate"); //named parameters

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

Alternatively, native queries can also be supported when using NamedQueries.

64. What happens when the no-args constructor is absent in the Entity bean?

Hibernate framework internally uses Reflection API for creating entity bean instances when get() or load() methods are called. The method Class.newInstance() is used which requires a no-args constructor to be present. When we don't have this constructor in the entity beans, then hibernate fails to instantiate the bean and hence it throws HibernateException.

65. Can we declare the Entity class final?

No, we should not define the entity class final because hibernate uses proxy classes and objects for lazy loading of data and hits the database only when it is absolutely needed. This is achieved by extending the entity bean. If the entity class (or bean) is made final, then it cant be extended and hence lazy loading can not be supported.

66. What are the states of a persistent entity?

A persistent entity can exist in any of the following states:

Transient:

- This state is the initial state of any entity object.
- Once the instance of the entity class is created, then the object is said to have entered a transient state. These objects exist in heap memory.
- In this state, the object is not linked to any session. Hence, it is not related to any database due to which any changes in the data object don't affect the data in the database.

```
InterviewBitEmployee employee=new InterviewBitEmployee(); //The object is in the transient state.

  employee.setId(101);
```

```
employee.setFullName("Hibernate");

employee.setEmail("hibernate@interviewbit.com");
```

Persistent:

- This state is entered whenever the object is linked or associated with the session.
- An object is said to be in a persistence state whenever we save or persist an object in the database. Each object corresponds to the row in the database table. Any modifications to the data in this state cause changes in the record in the database.

Following methods can be used upon the persistence object:

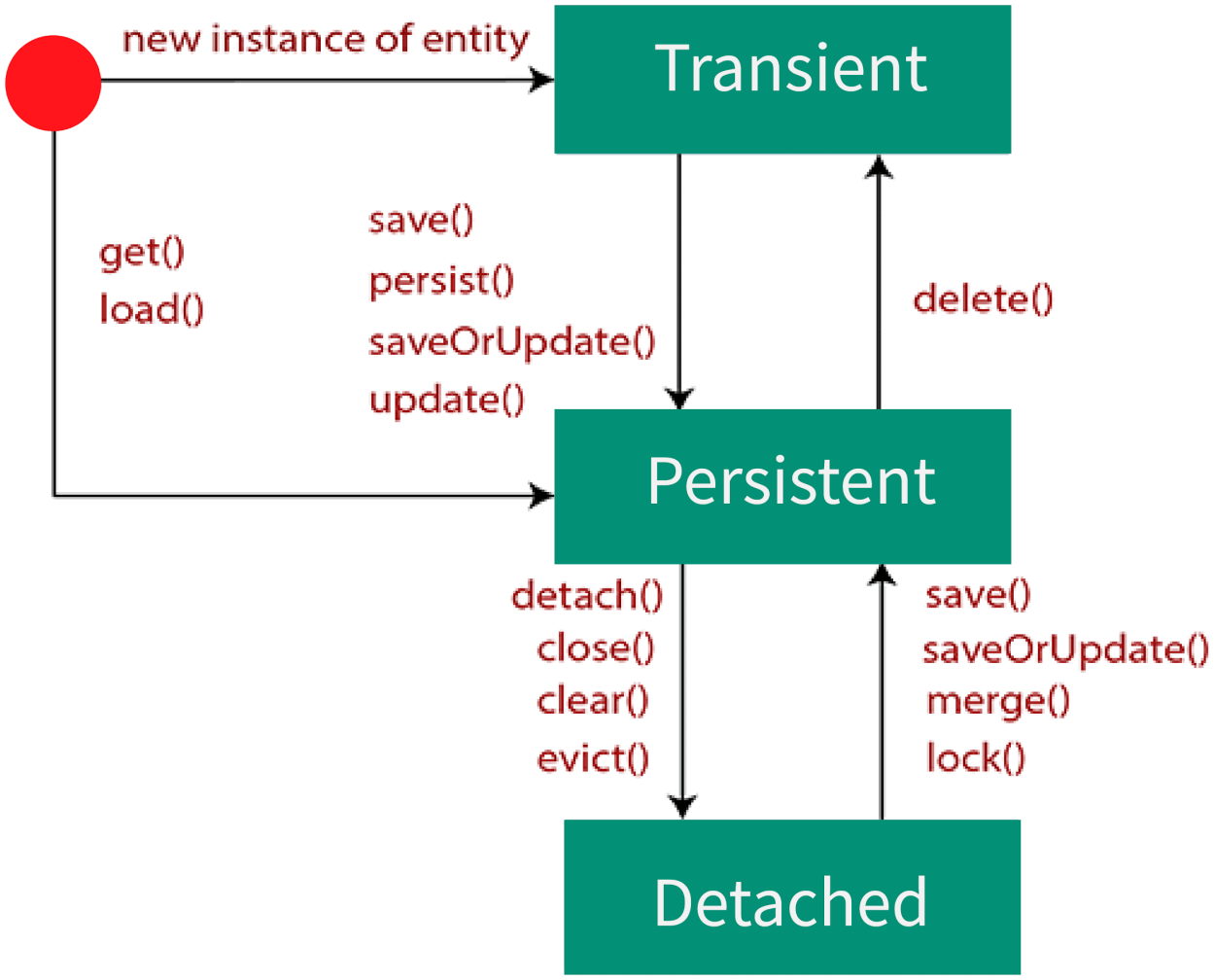
```
session.save(record);

session.persist(record);

session.update(record);

session.saveOrUpdate(record);

session.lock(record);

session.merge(record);
```

Detached:

- The object enters this state whenever the session is closed or the cache is cleared.
- Due to the object being no longer part of the session, any changes in the object will not reflect in the corresponding row of the database. However, it would still have its representation in the database.
- In case the developer wants to persist changes of this object, it has to be reattached to the hibernate session.
- In order to achieve the reattachment, we can use the methods load(), merge(), refresh(), update(), or save() methods on a new session by using the reference of the detached object.

The object enters this state whenever any of the following methods are called:

```
session.close();

session.clear();

session.detach(record);

session.evict(record);
```



InterviewBit

Persistent Entity

67. Explain Query Cache

Hibernate framework provides an optional feature called cache region for the queries' resultset. Additional configurations have to be done in code in order to enable this. The query cache is useful for those queries which are most frequently called with the same parameters. This increases the speed of the data retrieval and greatly improves performance for commonly repetitive queries.

This does not cache the state of actual entities in the result set but it only stores the identifier values and results of the value type. Hence, query cache should be always used in association with second-level cache.

Configuration:

In the hibernate configuration XML file, set the use_query_cache property to true as shown below:

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

In the code, we need to do the below changes for the query object:

```
Query query = session.createQuery("from InterviewBitEmployee");

query.setCacheable(true);

query.setCacheRegion("IB_EMP");
```

70. What are the concurrency strategies available in hibernate?

Concurrency strategies are the mediators responsible for storing and retrieving items from the cache. While enabling second-level cache, it is the responsibility of the developer to provide what strategy is to be implemented to decide for each persistent class and collection.

Following are the concurrency strategies that are used:

- **Transactional:** This is used in cases of updating data that most likely causes stale data and this prevention is most critical to the application.
- **Read-Only:** This is used when we don't want the data to be modified and can be used for reference data only.
- **Read-Write:** Here, data is mostly read and is used when the prevention of stale data is of critical importance.
- **Non-strict-Read-Write:** Using this strategy will ensure that there wouldn't be any consistency between the database and cache. This strategy can be used when the data can be modified and stale data is not of critical concern.

71. What is Single Table Strategy?

Single Table Strategy is a hibernate's strategy for performing inheritance mapping. This strategy is considered to be the best among all the other existing ones. Here, the inheritance data hierarchy is stored in the single table by making use of a discriminator column which determines to what class the record belongs.

For the example defined in the Hibernate Inheritance Mapping question above, if we follow this single table strategy, then all the permanent and contract employees' details are stored in only one table called InterviewBitEmployee in the database and the employees would be differentiated by making use of discriminator column named employee_type.

Hibernate provides @Inheritance annotation which takes strategy as the parameter. This is used for defining what strategy we would be using. By giving them value, InheritanceType.SINGLE_TABLE signifies that we are using a single table strategy for mapping.

- @DiscriminatorColumn is used for specifying what is the discriminator column of the table in the database corresponding to the entity.
- @DiscriminatorValue is used for specifying what value differentiates the records of two types.

The code snippet would be like this:

InterviewBitEmployee class:

```
@Entity

@Table(name = "InterviewBitEmployee")

@Inheritance(strategy = InheritanceType.SINGLE_TABLE)

@DiscriminatorColumn(name = "employee_type")

@NoArgsConstructor

@AllArgsConstructor

public class InterviewBitEmployee {

  @Id

  @Column(name = "employee_id")

  private String employeeId;

  private String fullName;

  private String email;

}
```

InterviewBitContractEmployee class:

```
@Entity

@DiscriminatorValue("contract")

@NoArgsConstructor
```

```
@AllArgsConstructor

public class InterviewBitContractEmployee extends InterviewBitEmployee {

   private LocalDate contractStartDate;

   private LocalDate contractEndDate;

   private String agencyName;

}
```

InterviewBitPermanentEmployee class:

```
@Entity

@DiscriminatorValue("permanent")

@NoArgsConstructor

@AllArgsConstructor

public class InterviewBitPermanentEmployee extends InterviewBitEmployee {

   private LocalDate workStartDate;

   private int numberOfLeaves;

}
```

72. Can you tell something about Table Per Class Strategy.

Table Per Class Strategy is another type of inheritance mapping strategy where each class in the hierarchy has a corresponding mapping database table. For example, the InterviewBitContractEmployee class details are stored in the interviewbit_contract_employee table and InterviewBitPermanentEmployee class details are stored in interviewbit_permanent_employee tables respectively. As the data is stored in different tables, there will be no need for a discriminator column as done in a single table strategy.

Hibernate provides @Inheritance annotation which takes strategy as the parameter. This is used for defining what strategy we would be using. By giving them value, InheritanceType.TABLE_PER_CLASS, it signifies that we are using a table per class strategy for mapping.

The code snippet will be as shown below:

InterviewBitEmployee class:

```
@Entity(name = "interviewbit_employee")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
@NoArgsConstructor
@AllArgsConstructor
public class InterviewBitEmployee {
  @Id
  @Column(name = "employee_id")
  private String employeeId;
  private String fullName;
  private String email;
}
```

InterviewBitContractEmployee class:

```
@Entity(name = "interviewbit_contract_employee")
@Table(name = "interviewbit_contract_employee")
@NoArgsConstructor
@AllArgsConstructor
public class InterviewBitContractEmployee extends InterviewBitEmployee {
  private LocalDate contractStartDate;
  private LocalDate contractEndDate;
  private String agencyName;
}
```

InterviewBitPermanentEmployee class:

```
@Entity(name = "interviewbit_permanent_employee")
@Table(name = "interviewbit_permanent_employee")
```

```
@NoArgsConstructor

@AllArgsConstructor

public class InterviewBitPermanentEmployee extends InterviewBitEmployee {

  private LocalDate workStartDate;

  private int numberOfLeaves;

}
```

Disadvantages:

- This type of strategy offers less performance due to the need for additional joins to get the data.
- This strategy is not supported by all JPA providers.
- Ordering is tricky in some cases since it is done based on a class and later by the ordering criteria.

73. Can you tell something about Named SQL Query

A named SQL query is an expression represented in the form of a table. Here, SQL expressions to select/retrieve rows and columns from one or more tables in one or more databases can be specified. This is like using aliases to the queries.

In hibernate, we can make use of @NameQueries and @NameQuery annotations.

- @NameQueries annotation is used for defining multiple named queries.
- @NameQuery annotation is used for defining a single named query.

Code Snippet: We can define Named Query as shown below

```
@NamedQueries(

  {

    @NamedQuery(

    name = "findIBEmployeeByFullName",

    query = "from InterviewBitEmployee e where e.fullName = :fullName"

    )

  }
```

)

:fullName refers to the parameter that is programmer defined and can be set using the query.setParameter method while using the named query.

Usage:

```
TypedQuery query = session.getNamedQuery("findIBEmployeeByFullName");

query.setParameter("fullName","Hibernate");

List<InterviewBitEmployee> ibEmployees = query.getResultList();
```

The getNamedQuery method takes the name of the named query and returns the query instance.

74. What are the benefits of NamedQuery?

In order to understand the benefits of NamedQuery, let's first understand the disadvantage of HQL and SQL. The main disadvantage of having HQL and SQL scattered across data access objects is that it makes the code unreadable. Hence, as good practice, it is recommended to group all HQL and SQL codes in one place and use only their reference in the actual data access code. In order to achieve this, Hibernate gives us named queries.

A named query is a statically defined query with a predefined unchangeable query string. They are validated when the session factory is created, thus making the application fail fast in case of an error.