Hibernate:
==========

1)What is ORM
Object-Relational Mapping
ORM is a technique that maps Java objects to database tables, allowing developers to interact with databases using objects instead of SQL.

```
Employee emp = new Employee(1, "Anjali", 30000);
session.save(emp);
```

ORM automatically:

- Creates SQL
- Talks to DB
- Stores data

2)JDBC vs Hibernate . Explain advantages and disadvantages

1️⃣ **What is JDBC?**

JDBC (Java Database Connectivity) is a Java API used to **connect Java applications directly to databases** and execute SQL queries.
You write **SQL manually** and handle everything yourself.

2️⃣ **What is Hibernate?**

Hibernate is an **ORM (Object-Relational Mapping) framework** that lets you **work with Java objects instead of SQL.**

✅ **Advantages of JDBC**

✓ Simple and lightweight
✓ Full control over SQL
✓ Better performance for complex queries
✓ No extra frameworks required
✓ Good for **small applications**

❌ **Disadvantages of JDBC**

✗ Too much boilerplate code
✗ Manual object mapping
✗ Hard to maintain large projects
✗ Database dependent
✗ Error-prone

✅ **Advantages of Hibernate**

✓ No SQL needed (HQL / Criteria / JPQL)
✓ Automatic object-table mapping
✓ Database independent
✓ Built-in caching
✓ Transaction management
✓ Faster development
✓ Easy CRUD operations

❌ **Disadvantages of Hibernate**

✗ Learning curve is higher
✗ Slightly slower than pure JDBC
✗ Complex configuration for beginners
✗ Not ideal for bulk operations
✗ Debugging SQL is harder

3)What is Configuration class in hibernate
The **Configuration class** is a **bootstrap class in Hibernate** used to **configure Hibernate and load database & mapping details.**
It **reads configuration files, creates SessionFactory,** and **initializes Hibernate.**
What does Configuration class do?
✓ Reads **hibernate.cfg.xml**
✓ Reads **mapping files / annotated classes**
✓ Loads DB details (URL, username, password, dialect)
✓ Builds **SessionFactory**
✓ Initializes Hibernate framework

4)What is session in hibernate

Session is a lightweight, non-thread-safe object that represents a single unit of work and provides methods to perform database operations in Hibernate.
What does a Session do?
✓ Opens a connection to the database
✓ Saves Java objects into DB
✓ Retrieves data from DB
✓ Updates DB records
✓ Deletes DB records
✓ Manages persistence context (1st level cache)
✓ Works with transactions

5)What is sessionfactory

SessionFactory is a **heavyweight, thread-safe object** used to **create Session objects.**
It is created **once per application** and shared across the entire app.
What does SessionFactory do?

✓ Reads Hibernate configuration (via Configuration)
✓ Holds DB connection settings
✓ Manages Hibernate mappings
✓ Creates Session objects
✓ Manages **Second-Level Cache**
✓ Thread-safe

6)What is first level cache in hibernate
First Level Cache is a **default cache associated with a Hibernate Session.** It stores objects **within the same session,** so repeated DB calls are avoided.
Each Session has its own first-level cache.

First level cache is enabled by default and cannot be disabled.

7)What is second level cache in hibernate
Second Level Cache is an **optional, SessionFactory-level cache** that stores objects **across multiple sessions.**
👉 Unlike first-level cache, it is **shared by all sessions.**

- ❌ **Not enabled by default**
- ✅ Needs **explicit configuration**
- 📦 Works at **SessionFactory level**
- 🔄 Reduces repeated DB calls across sessions

8)get vs load methods in hibernate
get() fetches data immediately and returns null if not found, while load() returns a proxy object and throws an exception if the record does not exist.
get() Hits DB immediately
load() Returns proxy object (lazy loading)

```
Employee emp = session.get(Employee.class, 1);
System.out.println(emp.getName());
```

✓ DB hit happens immediately
✓ If ID not found → null

9)explain hibernate pojo life cycle

Hibernate POJO life cycle describes the **different states of an object** as it moves between **Java application, Hibernate session, and database.**

🔄 **States of POJO in Hibernate**

Hibernate entity has **4 states**:

1️⃣ **Transient**

2️⃣ **Persistent**

3️⃣ **Detached**

4️⃣ **Removed**

1️⃣ **Transient State**

📌 Object is **created using new keyword**

📌 Not associated with Session

📌 Not stored in DB

```
Employee emp = new Employee(); // Transient
emp.setId(1);
```

✓ No session

✓ No DB record

2️⃣ **Persistent State**

📌 Object is associated with a **Session**

📌 Changes are **automatically synchronized** with DB

```
Session session = factory.openSession();
Transaction tx = session.beginTransaction();

session.save(emp); // Persistent
tx.commit();
```

✓ Managed by Hibernate

✓ Stored in DB

✓ 1st level cache applies

3️⃣ **Detached State**

📌 Session is **closed,** but object still exists

📌 Object is **not managed** by Hibernate

```
session.close(); // Detached
emp.setSalary(50000); // No DB update
```

✓ Exists in memory

✓ No automatic DB sync

To reattach:

```
session.update(emp);
```

4️⃣ **Removed State**

📌 Object is **scheduled for deletion**

📌 Record removed from DB

```
session.delete(emp); // Removed
```

✓ DB row deleted

✓ Object no longer persistent

🔄 **Life Cycle Diagram (Easy to remember)**

```
new → Transient
save() → Persistent
close() → Detached
delete() → Removed
```

10)Explain Hibernate mappings and how it works(One to One , One to Many , Many to one and Many to Many

Hibernate mappings define relationships between entities such as One-to-One, One-to-Many, Many-to-One, and Many-to-Many, enabling Hibernate to automatically manage table relationships using foreign keys or join tables.
It connects:

- **Java objects ↔ Database tables**
- **Object relationships ↔ Table relationships (FK / Join tables)**

1️⃣ **One-to-One Mapping**

📌 **Meaning**

One object is associated with **exactly one** other object.

🧠 **Example**

- Person ↔ Passport
- User ↔ Profile

🧩 **Java Example**

```
@Entity
class User {
   @Id
   private int id;

   @OneToOne
   @JoinColumn(name="profile_id")
   private Profile profile;
}
@Entity
class Profile {
   @Id
   private int id;
}
```

🗄️ **Database**

USER ── profile_id ──▶ PROFILE

2️⃣ **One-to-Many Mapping**

📌 **Meaning**

One object is associated with **multiple objects.**

🧠 **Example**

- Department → Employees
- Customer → Orders

🧩 **Java Example**

```
@Entity
class Department {
   @Id
   private int id;

   @OneToMany(mappedBy="department")
```

```
   private List<Employee> employees;
}
@Entity
class Employee {
   @ManyToOne
   @JoinColumn(name="dept_id")
   private Department department;
}
```

🗄️ **Database**

```
DEPARTMENT
EMPLOYEE (dept_id)
```

3️⃣ **Many-to-One Mapping**

📌 **Meaning**

Many objects are associated with **one common object.**

🧠 **Example**

- Many Employees → One Department
- Many Orders → One Customer

🧩 **Java Example**

```
@Entity
class Employee {
   @ManyToOne
   @JoinColumn(name="dept_id")
   private Department department;
}
```

✓ This is the **most commonly used mapping**

✓ Foreign key exists on **many side**

4️⃣ **Many-to-Many Mapping**

📌 **Meaning**

Many objects are associated with **many objects.**

🧠 **Example**

- Students ↔ Courses
- Users ↔ Roles

🧩 **Java Example**

```
@Entity
class Student {
   @ManyToMany
   @JoinTable(
      name="student_course",
      joinColumns=@JoinColumn(name="student_id"),
      inverseJoinColumns=@JoinColumn(name="course_id")
   )
   private List<Course> courses;
}
@Entity
class Course {
   @ManyToMany(mappedBy="courses")
   private List<Student> students;
}
```

🗄️ **Database**

```
STUDENT
COURSE
```

STUDENT_COURSE (student_id, course_id)

🔄 **How Hibernate Mapping Works Internally**

1️⃣ Read annotations / XML
2️⃣ Create table relationships
3️⃣ Generate SQL automatically
4️⃣ Manage foreign keys
5️⃣ Sync objects ↔ DB rows

11)explain the inheritance in hibernate

Hibernate inheritance allows you to **map Java inheritance (IS-A relationship)** to **database tables.**

👉 One **parent class** and multiple **child classes** share common properties.

✅ **Advantages**

✓ Fast performance
✓ No joins
✓ Simple queries

❌ **Disadvantages**

✕ Many NULL columns
✕ Table grows wide

12)what is dialect in hibernate ? what is benefit of dialect

A **Hibernate Dialect** tells Hibernate **which database you are using** and **how to generate database-specific SQL.**

👉 Different databases have different SQL syntax.

Dialect acts as a **translator between Hibernate and the database.**

In hibernate.cfg.xml

```
<property name="hibernate.dialect">
  org.hibernate.dialect.MySQLDialect
</property>
```

✅ **Benefits of Hibernate Dialect**

✓ Database independence
✓ Automatic SQL generation
✓ Handles DB-specific features
✓ Simplifies migration between DBs
✓ Supports pagination & sequences

❌ **What happens if Dialect is wrong?**

❌ SQL syntax errors
❌ Table creation fails
❌ Pagination not working
❌ Wrong data types

13)Explain the Hibernate Criteria.
Hibernate Criteria is an **object-oriented, programmatic way to fetch data from the database** without writing SQL or HQL.
👉 Queries are built **dynamically at runtime** using Java objects.

```
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Employee> cq = cb.createQuery(Employee.class);
Root<Employee> root = cq.from(Employee.class);

cq.select(root).where(cb.equal(root.get("department"), "IT"));

List<Employee> result = session.createQuery(cq).getResultList();
```

14) difference b/w Save And Persist Methods
save() is a Hibernate-specific method that immediately inserts and returns the generated ID, while persist() is a JPA method that inserts the entity at transaction commit and returns no value.

15) difference b/w Merge And Update Methods In Hibernate
Both merge() and update() are used to **synchronize a detached object with the database**, but they work **very differently**
update() reattaches the same object to the session, while merge() copies the state of the object into a managed instance and returns it.

update() attaches the same detached object to the session and may cause conflicts, whereas merge() safely copies the object state and returns a managed instance.

16) What is projections in hibernate
Projections are used to **retrieve partial data instead of full entity objects** from the database.

👉 Instead of fetching the entire object, you fetch **only required columns or aggregated values.**

```
Criteria cr = session.createCriteria(Employee.class);
cr.setProjection(Projections.property("name"));

List<String> names = cr.list();
```

17) Explain Mappings and configuration files in hibernate

Hibernate needs **two things** to work properly:

1. **Configuration files** – tell Hibernate how to connect to the database and behave.
2. **Mapping files** – tell Hibernate how Java classes relate to database tables.

1️⃣ **Hibernate Configuration Files**

a) hibernate.cfg.xml (Most common)

This is an **XML file** that contains:

- Database connection details (URL, username, password)
- JDBC driver class
- Hibernate properties (dialect, show_sql, cache settings)
- Mapping files or annotated classes

```
<!DOCTYPE hibernate-configuration PUBLIC
 "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
 "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property
name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</proper
ty>
    <property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/testdb</prope
rty>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">root</property>
    <property
name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.show_sql">true</property>

    <!-- Mapping file -->
    <mapping resource="Employee.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

ibernate.properties (Optional)

```
hibernate.connection.driver_class=com.mysql.cj.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost:3306/testdb
hibernate.connection.username=root
hibernate.connection.password=root
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.show_sql=true
```

You can use either cfg.xml or properties file, but cfg.xml is preferred.

2️⃣ Hibernate Mapping Files

Mapping files tell Hibernate **how to map Java classes to database tables.**

a) XML-based Mapping (.hbm.xml)

Example: Employee.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="com.example.Employee" table="EMPLOYEE">
        <id name="id" column="ID">
            <generator class="increment"/>
        </id>
        <property name="name" column="NAME"/>
        <property name="salary" column="SALARY"/>
    </class>
</hibernate-mapping>
```

---

b) Annotation-based Mapping (Modern approach)

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

    @Column(name="NAME")
    private String name;

    @Column(name="SALARY")
    private double salary;
}
```

Modern Hibernate uses **annotations** more than XML.

How it Works Together

1. Hibernate reads hibernate.cfg.xml
2. Config loads DB connection and properties
3. Mapping files / annotations tell Hibernate how classes map to tables
4. Hibernate builds **SessionFactory → Session → DB operations**