**POJO vs Java Bean:**

**In Java, POJO (Plain Old Java Object) and Java Bean are both types of Java classes used to represent objects, but they have some key differences in terms of structure, usage, and restrictions.**

**1. What is a POJO?**

**A Plain Old Java Object (POJO) is a simple Java class that follows no special restrictions except for being a valid Java class. It does not require any specific inheritance, annotations, or methods.**

**Example of a POJO**

**public class Employee {**

**private int id;**

**private String name;**

**private double salary;**

**public Employee(int id, String name, double salary) {**

**this.id = id;**

**this.name = name;**

**this.salary = salary;**

**}**

**public int getId() {**

**return id;**

**}**

**public String getName() {**

**return name;**

**}**

**public double getSalary() {**

**return salary;**

**}**

**}**

**Characteristics of POJO**

**✅ No specific restrictions.  
✅ Can have public, private, or protected fields and methods.  
✅ Can use constructors with or without parameters.  
✅ No enforced getter/setter methods.  
✅ Can have business logic methods.**

**2. What is a Java Bean?**

**A Java Bean is a special type of POJO that follows specific conventions, making it more suitable for frameworks like Hibernate, Spring, and JSP.**

**Example of a Java Bean**

**import java.io.Serializable;**

**public class EmployeeBean implements Serializable {**

**private int id;**

**private String name;**

**private double salary;**

**// No-argument constructor**

**public EmployeeBean() {}**

**// Getter and Setter methods**

**public int getId() {**

**return id;**

**}**

**public void setId(int id) {**

**this.id = id;**

**}**

**public String getName() {**

**return name;**

**}**

**public void setName(String name) {**

**this.name = name;**

**}**

**public double getSalary() {**

**return salary;**

**}**

**public void setSalary(double salary) {**

**this.salary = salary;**

**}**

**}**

**Characteristics of Java Bean**

**✅ Must have a no-argument constructor.  
✅ Must have getter and setter methods for all fields.  
✅ Should implement Serializable (optional, but recommended).  
✅ Fields should be private (encapsulation).  
✅ Should not have business logic methods inside the Bean class.**

**3. Key Differences Between POJO and Java Bean**

| **Feature** | **POJO** | **Java Bean** |
| --- | --- | --- |
| **Definition** | **A simple Java class with no strict rules.** | **A Java class that follows specific conventions.** |
| **Constructors** | **Can have parameterized constructors.** | **Must have a no-argument constructor.** |
| **Encapsulation** | **Not required.** | **Mandatory (fields should be private with getters and setters).** |
| **Getters/Setters** | **Not mandatory.** | **Mandatory.** |
| **Serialization** | **Not required.** | **Recommended (should implement Serializable).** |
| **Business Logic** | **Allowed.** | **Should not contain business logic.** |
| **Framework Usage** | **General use cases.** | **Used in frameworks like Hibernate, JSP, and Spring.** |

**4. When to Use POJO vs Java Bean?**

* **Use POJO when you just need a simple Java object without constraints.**
* **Use Java Bean when working with frameworks that require strict conventions, such as Hibernate, JSP, and Spring.**

**Summary**

**A Java Bean is a specialized POJO that follows stricter conventions, making it useful for frameworks that rely on standard naming conventions and serialization.**

**What is Hibernate?**

Hibernate is an **open-source Object-Relational Mapping (ORM) framework** for Java. It simplifies database interaction by mapping Java objects to database tables and handling SQL operations automatically.

**Key Features:**

* Eliminates boilerplate JDBC code
* Supports **HQL (Hibernate Query Language)**
* Provides **automatic table creation** from entity classes
* Manages **caching** for performance optimization
* Supports various relational databases (MySQL, PostgreSQL, Oracle, etc.)

**Why Use Hibernate?**

1. **Simplifies Database Interactions**
   * Converts Java objects to database records and vice versa automatically.
2. **Eliminates JDBC Complexity**
   * No need to manually write SQL queries for CRUD operations.
3. **Portable Across Databases**
   * Can switch databases (e.g., from MySQL to PostgreSQL) with minimal changes.
4. **Supports Annotations & XML Configuration**
   * Offers flexibility in defining mappings.
5. **Better Performance with Caching**
   * Supports first-level and second-level caching to reduce database load.

**Hibernate Usage in Java**

**1. Basic Steps to Use Hibernate**

1. **Configure Hibernate** (via hibernate.cfg.xml or properties file).
2. **Create Entity Classes** (POJOs with annotations).
3. **Define Mappings** (using annotations or XML).
4. **Create a SessionFactory** (to manage sessions).
5. **Perform CRUD Operations** (using Hibernate Session).

**Example: Using Hibernate to Manage Employees**

**Step 1: Add Hibernate Dependency**

<!-- Hibernate Core -->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>5.6.5.Final</version>

</dependency>

<!-- MySQL Driver -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.33</version>

</dependency>

**Step 2: Configure Hibernate (hibernate.cfg.xml)**

<hibernate-configuration>

<session-factory>

<property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property>

<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/mydb</property>

<property name="hibernate.connection.username">root</property>

<property name="hibernate.connection.password">password</property>

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

<property name="hibernate.hbm2ddl.auto">update</property>

</session-factory>

</hibernate-configuration>

**Step 3: Create an Entity Class (Employee.java)**

import javax.persistence.\*;

@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;

// Constructors, Getters & Setters

}

**Step 4: Hibernate SessionFactory Setup**

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

public class HibernateUtil {

private static final SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();

public static Session getSession() {

return sessionFactory.openSession();

}

}

**Step 5: CRUD Operations (Example - Saving an Employee)**

import org.hibernate.Session;

import org.hibernate.Transaction;

public class EmployeeDAO {

public static void main(String[] args) {

Session session = HibernateUtil.getSession();

Transaction tx = session.beginTransaction();

Employee emp = new Employee();

emp.setName("John Doe");

emp.setSalary(50000);

session.save(emp);

tx.commit();

session.close();

System.out.println("Employee Saved Successfully!");

}

}

**Conclusion**

Hibernate is a powerful ORM framework that: ✔ **Simplifies database interactions**  
✔ **Eliminates JDBC complexities**  
✔ **Provides portability & caching features**  
✔ **Supports automatic schema management**

**Hibernate CRUD Operations (Create, Read, Update, Delete)**

Hibernate provides multiple methods for performing CRUD operations on a database. Below are the different ways to achieve each operation.

**1. Create (Insert) Operations**

Hibernate provides methods to insert records into the database.

**Methods to Insert Data**

1. save(Object entity) – Saves an entity and returns the generated identifier.
2. persist(Object entity) – Similar to save(), but works only within a transaction.
3. saveOrUpdate(Object entity) – Inserts if the record doesn’t exist, updates otherwise.

**Example: Saving an Employee**

Session session = HibernateUtil.getSession();

Transaction tx = session.beginTransaction();

Employee emp = new Employee();

emp.setName("John Doe");

emp.setSalary(50000);

// Method 1: save()

int id = (int) session.save(emp); // Returns generated ID

System.out.println("Employee saved with ID: " + id);

// Method 2: persist()

session.persist(emp);

// Method 3: saveOrUpdate()

session.saveOrUpdate(emp);

tx.commit();

session.close();

**2. Read (Retrieve) Operations**

Hibernate allows fetching records using different methods.

**Methods to Retrieve Data**

1. get(Class<T>, Serializable id) – Returns an entity or null if not found.
2. load(Class<T>, Serializable id) – Returns a proxy; throws exception if not found.
3. createQuery(HQL Query) – Uses Hibernate Query Language (HQL) for custom queries.
4. createSQLQuery(SQL Query) – Uses native SQL queries.
5. Criteria API – Uses an object-oriented approach for fetching records.

**Example: Fetching an Employee**

Session session = HibernateUtil.getSession();

// Method 1: get()

Employee emp1 = session.get(Employee.class, 1);

System.out.println("Employee Name: " + emp1.getName());

// Method 2: load()

Employee emp2 = session.load(Employee.class, 2);

System.out.println("Employee Name: " + emp2.getName());

// Method 3: HQL Query

Query query = session.createQuery("FROM Employee WHERE name = :name");

query.setParameter("name", "John Doe");

List<Employee> employees = query.list();

// Method 4: Native SQL Query

Query sqlQuery = session.createSQLQuery("SELECT \* FROM employee").addEntity(Employee.class);

List<Employee> employeeList = sqlQuery.list();

session.close();

**3. Update Operations**

Hibernate provides different ways to update records in the database.

**Methods to Update Data**

1. update(Object entity) – Updates an existing record.
2. merge(Object entity) – Merges detached objects into a persistent state.
3. saveOrUpdate(Object entity) – Either saves a new record or updates an existing one.
4. createQuery("UPDATE ...") – Uses HQL for batch updates.

**Example: Updating an Employee**

Session session = HibernateUtil.getSession();

Transaction tx = session.beginTransaction();

// Fetch employee

Employee emp = session.get(Employee.class, 1);

emp.setSalary(60000);

// Method 1: update()

session.update(emp);

// Method 2: merge()

session.merge(emp);

// Method 3: saveOrUpdate()

session.saveOrUpdate(emp);

// Method 4: HQL Query

Query query = session.createQuery("UPDATE Employee SET salary = :salary WHERE id = :id");

query.setParameter("salary", 70000);

query.setParameter("id", 1);

query.executeUpdate();

tx.commit();

session.close();

**4. Delete Operations**

Hibernate allows deleting records in multiple ways.

**Methods to Delete Data**

1. delete(Object entity) – Deletes a specific entity.
2. createQuery("DELETE FROM ...") – Uses HQL for bulk deletion.
3. createSQLQuery("DELETE FROM ...") – Uses native SQL.

**Example: Deleting an Employee**

Session session = HibernateUtil.getSession();

Transaction tx = session.beginTransaction();

// Method 1: delete()

Employee emp = session.get(Employee.class, 1);

session.delete(emp);

// Method 2: HQL Query

Query query = session.createQuery("DELETE FROM Employee WHERE id = :id");

query.setParameter("id", 2);

query.executeUpdate();

// Method 3: Native SQL Query

Query sqlQuery = session.createSQLQuery("DELETE FROM employee WHERE id = :id");

sqlQuery.setParameter("id", 3);

sqlQuery.executeUpdate();

tx.commit();

session.close();

**Conclusion**

Hibernate provides various methods for **CRUD (Create, Read, Update, Delete)** operations. The best method depends on your use case:

* save(), persist(), saveOrUpdate() → Insert records
* get(), load(), createQuery() → Read records
* update(), merge(), saveOrUpdate() → Update records
* delete(), createQuery(DELETE) → Delete records