Module 1 : Overview of OOP Features & Exception Handling

Q1] Explain the concepts of inheritance and polymorphism with examples. How are they implemented in Java?

Inheritance

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that we can create new classes that are built upon existing classes. When we inherit methods from an existing class, we can reuse methods and fields of the parent class. However, we can add new methods and fields in your current class also.

Syntax

```
1. class Subclass-name extends Superclass-name
2. {
3.   //methods and fields
4. }
```

Types :

Single Inheritance Example

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

polymorphism :

polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

- **Compile-Time Polymorphism (Method Overloading):** Multiple methods share the same name but differ in parameters within the same class.

  Example

```
class Calculator {
   int add(int a, int b) {
      return a + b;
   }

   double add(double a, double b) {
      return a + b;
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("Sum of integers: " + calc.add(5, 10));
        System.out.println("Sum of doubles: " + calc.add(5.5, 10.5));
    }
}
```

- **Runtime Polymorphism (Method Overriding):** A subclass provides a specific implementation of a method already defined in its superclass.

```
class Animal {
    void sound() {
        System.out.println("This animal makes a sound.");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("The dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myAnimal = new Dog();
        myAnimal.sound(); // Calls the overridden method in Dog class
    }
}
```

2] Describe abstract classes and interfaces. What are the key differences between them? Provide examples

Abstract Classes:

- **Definition:** An abstract class cannot be instantiated on its own and may contain both complete (concrete) methods and incomplete (abstract) methods.
- The abstract class defined using Abstract Keyword..

- **Example:**

```
abstract class Animal {
   // Concrete method
   void eat() {
      System.out.println("This animal eats food.");
   }

   // Abstract method
   abstract void makeSound();
}

class Dog extends Animal {
   @Override
   void makeSound() {
      System.out.println("The dog barks.");
   }
}
```

In this example, Animal is an abstract class with a concrete method eat() and an abstract method makeSound(). The Dog class extends Animal and provides an implementation for makeSound().

- An interface in Java is a blueprint of a class. It has static constants and abstract methods.
- The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.
- In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.
- Java Interface also represents the IS-A relationship.
- It cannot be instantiated just like the abstract class.

- We can implements interface using implements keyword.
- **Example:**

```
interface Animal {
   void makeSound();
}

class Dog implements Animal {
   @Override
   public void makeSound() {
      System.out.println("The dog barks.");
   }
}
```

Here, Animal is an interface with the method makeSound(). The Dog class implements this interface and provides the specific behavior for makeSound().

Key Differences:

1. **Method Implementation:**
    - *Abstract Class:* Can have both abstract methods (without a body) and concrete methods (with a body).
    - *Interface:* interface have abstract methods. Interface have default and static methods with implementations.
2. **Multiple Inheritance:**
    - *Abstract Class:* A class can inherit from only one abstract class (single inheritance).
    - *Interface:* A class can implement multiple interfaces, allowing multiple inheritance of type.
3. **Constructors:**
    - *Abstract Class:* Can have constructors.
    - *Interface:* Cannot have constructors.
4. **Access Modifiers:**
    - *Abstract Class:* Can have members with any access modifier (private, protected, public).
    - *Interface:* All members are implicitly public; private methods are allowed .
5. **Fields:**
    - *Abstract Class:* Can have instance variables.
    - *Interface:* Can have only static final variables (constants).
6. **Inheritance:**
    - *Abstract Class:* Can extend another class and implement multiple interfaces.
    - *Interface:* Can extend multiple interfaces but cannot extend a class.

3] Define inner classes and anonymous classes. Explain their usage with appropriate examples.

An **inner class** is a class that is defined within another class. These classes can access the members (including private members) of the outer class. There are four types of inner classes in Java:

1. **Non-static Inner Class**: This class is associated with an instance of the outer class. It can access both static and non-static members of the outer class.

```
class OuterClass {
   private int outerVar = 10;

   class InnerClass {
      public void display() {
         System.out.println("Outer class variable: " + outerVar);
      }
   }
}

public class Main {
   public static void main(String[] args) {
            outer = new OuterClass();
      OuterClass.InnerClass inner = outer.new InnerClass();
      inner.display(); // Accesses outer class variable
   }
}
```

2. **Static Nested Class**: A static nested class is a static member of the outer class. It cannot access instance members (non-static) of the outer class, but it can access static members.

```
class OuterClass {
    static int staticVar = 20;

    static class StaticNestedClass {
        public void display() {
            System.out.println("Static variable: " + staticVar);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        OuterClass.StaticNestedClass nested = new OuterClass.StaticNestedClass();
        nested.display(); // Accesses static variable
    }
}
```

3. **Method Local Inner Class**: These classes are defined inside a method and can only be used within that method.

```
class OuterClass {
    public void outerMethod() {
        class MethodLocalInnerClass {
            public void display() {
                System.out.println("Inside method local inner class");
            }
        }
        MethodLocalInnerClass inner = new MethodLocalInnerClass();
        inner.display();
    }
}

public class Main {
    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        outer.outerMethod(); // Calls method that contains inner class
    }
}
```

4. **Anonymous Inner Class**: An anonymous class is a class without a name and is used for instantiating a class at the time of its creation. It is commonly used to implement interfaces or extend classes in a concise manner.

```
interface Greet {
    void sayHello();
}

public class Main {
    public static void main(String[] args) {
        Greet greet = new Greet() {
```

```
            public void sayHello() {
                System.out.println("Hello, Anonymous Class!");
            }
        };
        greet.sayHello();
    }
}
```

Anonymous Classes

An **anonymous class** is a type of inner class that does not have a name. These are useful when you need to override methods of an interface or a class without the need for creating a separate named class.

Implementing Interfaces:

```
interface Display {
  void show();
}

public class Main {
  public static void main(String[] args) {
    Display display = new Display() {
      public void show() {
        System.out.println("This is an anonymous class implementation of an interface.");
      }
    };
    display.show();
  }
}
```

Extending Classes:

```
class Animal {
  void sound() {
    System.out.println("Animal makes a sound");
  }
}

public class Main {
  public static void main(String[] args) {
    Animal animal = new Animal() {
      void sound() {
        System.out.println("Anonymous Animal makes a sound");
      }
    };
    animal.sound();
  }
}
```

Q5] What are exceptions and errors in Java? How do they differ? Explain with examples.

An **exception** is an event that disrupts the normal flow of the program's execution. It typically occurs when the program encounters a condition it cannot handle (e.g., trying to divide by zero, accessing an invalid index in an array). Java provides a robust exception-handling mechanism using try, catch, throw, throws, and finally

```
public class Example {
   public static void main(String[] args) {
      try {
         int[] arr = new int[5];
         arr[10] = 100;  // This will throw ArrayIndexOutOfBoundsException
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Exception caught: " + e);
      }
   }
}
```

Errors in Java:

An **error** is a serious problem that usually indicates a fault in the Java runtime environment. Errors typically occur due to hardware failure, system crashes, or lack of system resources. Errors are not meant to be caught or handled by the program.

```
public class Example {
   public static void main(String[] args) {
      try {
         // Simulating a StackOverflowError
         main(null);  // This will cause a StackOverflowError
      } catch (StackOverflowError e) {
         System.out.println("Error caught: " + e);
      }
   }
}
```

Q4] Discuss the types of exceptions in Java. Differentiate between checked and unchecked exceptions.

In Java, **exceptions** are events that disrupt the normal flow of a program's execution. They occur when the program encounters a condition it cannot handle, such as trying to divide by zero or accessing an invalid index in an array.

Types of Exceptions in Java:

1. **Checked Exceptions:**
    - **Definition:** Exceptions that are checked at compile-time. The programmer is required to handle them either by using a try-catch block or by declaring them in the method signature using throws.
    - **Examples:** IOException, SQLException.
2. **Unchecked Exceptions:**
    - **Definition:** Exceptions that are not checked at compile-time, but rather at runtime. They typically result from programming errors and do not need to be explicitly handled.

- **Examples:** NullPointerException, ArrayIndexOutOfBoundsException.

Differences Between Checked and Unchecked Exceptions:

- **Checked Exceptions:**
  - Are checked at compile-time.
  - Must be handled using try-catch or declared with throws.
  - Typically occur due to external factors (e.g., I/O operations, database access).
  - Example: IOException, SQLException.
- **Unchecked Exceptions:**
  - Are not checked at compile-time.
  - Can be ignored (not mandatory to handle).
  - Typically occur due to programming errors (e.g., null reference, invalid array index).
  - Example: NullPointerException, ArithmeticException.

Example of a Checked Exception:

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class CheckedExceptionExample {
   public static void main(String[] args) {
      try {
         File file = new File("example.txt");
         FileReader fr = new FileReader(file);
         // Perform file operations
         fr.close();
      } catch (IOException e) {
         System.out.println("An IOException occurred: " + e.getMessage());
      }
   }
}
```

Example of an Unchecked Exception:

```
 public class UncheckedExceptionExample {
   public static void main(String[] args) {
      int[] numbers = new int[5];
      try {
         numbers[10] = 100; // This will throw ArrayIndexOutOfBoundsException
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("An ArrayIndexOutOfBoundsException occurred: " + e.getMessage());
      }
   }
}
```

In summary, **checked exceptions** are conditions that a reasonable application might want to catch and handle, while **unchecked exceptions** are typically programming errors that are not expected to be caught

6] Explain the use of try, catch, finally, throw, and throws in exception handling with examples.

In Java, exception handling is a mechanism used to handle runtime errors, ensuring the normal flow of the program is maintained. The main keywords used for exception handling are: try, catch, finally, throw, and throws.

1. try Block

- **Definition:** The try block contains code that may throw an exception. If an exception occurs, the remaining code in the try block is skipped, and control is transferred to the catch block.

Example

```
try {
        int result = 10 / 0; // This will throw an ArithmeticException
}
```

catch Block

- **Definition:** The catch block is used to handle exceptions thrown by the try block. It catches and processes exceptions, preventing the program from crashing.

```
catch (ArithmeticException e) {
   System.out.println("An error occurred: " + e.getMessage());
}
```

finally Block

- **Definition:** The finally block is optional and is always executed, regardless of whether an exception was thrown or not. It is typically used for cleanup activities, like closing files or database connections.

```
finally {
   System.out.println("This block executes regardless of an exception.");
}
```

throw Statement

- **Definition:** The throw statement is used to explicitly throw an exception. It can be used to create custom exceptions or rethrow an existing one

```
throw new ArithmeticException("Custom exception thrown.");
```

throws Keyword

- **Definition:** The throws keyword is used in a method signature to declare exceptions that a method might throw. The caller of the method must handle or declare these exceptions.

```
public static void riskyMethod() throws ArithmeticException {
        int result = 10 / 0;                    }
```

Example

```
public class ExceptionHandlingExample {

  // Method that may throw an exception
  public static void riskyMethod() throws ArithmeticException {
    int result = 10 / 0;  // This will throw ArithmeticException
  }

  public static void main(String[] args) {
    try {
      riskyMethod();  // Calling the method that may throw an exception
    } catch (ArithmeticException e) {
```

```
            System.out.println("An error occurred: " + e.getMessage());
        } finally {
            System.out.println("This block executes regardless of an exception.");
        }

        try {
            throw new CustomException("This is a custom exception.");
        } catch (CustomException e) {
            System.out.println(e.getMessage());
        }
    }
}

// Custom exception class
class CustomException extends Exception {
    public CustomException(String message) {
        super(message);
    }
}
```

Explanation:

- **try block:** The method riskyMethod() is called, which may throw an ArithmeticException.
- **catch block:** Catches the ArithmeticException and prints an error message.
- **finally block:** Ensures that the message "This block executes regardless of an exception" is printed, regardless of whether an exception occurred.
- **throw statement:** Explicitly throws a CustomException.
- **throws keyword:** Used in the method riskyMethod() to indicate that it may throw an exception (in this case, ArithmeticException).

7] How can you create user-defined exceptions in Java? Illustrate with a program

---

Steps to Create a User-Defined Exception

1. Create a class that extends Exception (for checked exceptions) or RuntimeException (for unchecked exceptions).
2. Provide constructors to initialize the exception with a custom message or cause.
3. Use the throw keyword to throw the custom exception where required.

Example

```
// Custom exception class
class InvalidBalanceException extends Exception {
    public InvalidBalanceException(String message) {
        super(message); // Call the constructor of the parent Exception class
    }
```

```
}
class BankAccount {
    private double balance;

    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }

    // Method to withdraw money
    public void withdraw(double amount) throws InvalidBalanceException {
        if (amount > balance) {
            // Throwing the custom exception
            throw new InvalidBalanceException("Insufficient balance! Your current balance is " + balance);
        } else {
            balance -= amount;
            System.out.println("Withdrawal successful! Remaining balance: " + balance);
        }
    }
}

// Main class
public class CustomExceptionExample {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(500.0); // Initial balance is 500

        try {
            account.withdraw(600.0); // Attempting to withdraw more than the balance
        } catch (InvalidBalanceException e) {
            System.out.println("Exception: " + e.getMessage());
        }

        try {
            account.withdraw(100.0); // Valid withdrawal
        } catch (InvalidBalanceException e) {
            System.out.println("Exception: " + e.getMessage());
        }
    }
}
```

Explanation

1. **Custom Exception Class (InvalidBalanceException)**
    - Extends Exception to create a checked exception.
    - Includes a constructor to accept and pass a custom error message.
2. **Throwing the Custom Exception**
    - The withdraw method checks if the withdrawal amount exceeds the current balance.
    - If the condition is true, it throws the custom exception using the throw keyword.
3. **Handling the Custom Exception**
    - The main method wraps the calls to withdraw in a try-catch block to handle the exception.

8] What is the Java Collection Framework? Discuss its components and usage in detail.

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Key Components of the Collection Framework

1. **Interfaces**

Interfaces define the contract that concrete classes implement.

- **Collection<E>:** Root interface of the collection framework.
- **List<E>:** Ordered collection that allows duplicate elements. Examples: ArrayList, LinkedList,vector
- **Set<E>:** Collection that does not allow duplicate elements. Examples: HashSet, TreeSet.
- **Map<K, V>:** Represents key-value pairs. Examples: HashMap, TreeMap.

2. **Classes**

ArrayList: Resizable array implementation of the List interface.
LinkedList: Doubly linked list implementation of both List and Deque.
HashSet: Implements the Set interface using a hash table.
TreeSet: Implements the Set interface using a red-black tree.
HashMap: Implements the Map interface using a hash table.
TreeMap: Implements the Map interface using a red-black tree.

Example ..

Module 2 : Multithreading and Concurrency

1] Explain inter-thread communication in Java. How is it achieved using wait(), notify(), and notifyAll()?

What is Inter-Thread Communication in Java?

Inter-thread communication in Java allows threads to communicate with each other in a synchronized manner. It is essential for coordination between threads, especially in scenarios where one thread must wait for another thread to complete its task before proceeding.

Java provides three methods for inter-thread communication:

1. **wait()**: Causes the current thread to wait until it is notified.
2. **notify()**: Wakes up a single thread that is waiting on the object's monitor.
3. **notifyAll()**: Wakes up all threads that are waiting on the object's monitor.

```
4. class shareResource{
5.     public int value=0;
6.     public synchronized void consumer()throws Exception{
7.         while(value!=0){
8.             wait();
9.         }
10.        value=1;
11.        System.out.println("Consumer "+value);
12.        notify();
13.    }
14.    public synchronized void produce() throws Exception{
15.        while(value==0){
16.            wait();
17.        }
18.        value=0;
19.        System.out.println("product : "+value);
20.        notify();
21.    }
22.}
23.class demo{
24.    public static void main(String[] args){
25.        shareResource obj=new shareResource();
26.        Thread producer=new Thread(()-> {
27.            try{
28.                obj.produce();
29.            }catch(Exception e){
30.                System.out.println(e);
31.            }
32.        });
33.        Thread consumer=new Thread(()->{
```

```
            try{
                obj.consumer();
            }catch(Exception ex){
                System.out.println(ex);
            }
        });
        producer.start();
        consumer.start();
    }
}
```

2] What is a deadlock? How does it occur in a multithreaded environment? Illustrate with an example.

A **deadlock** in a multithreaded environment refers to a situation where two or more threads are unable to proceed because each is waiting for the other to release a resource. As a result, the threads become stuck indefinitely, and the program fails to continue its execution.

Conditions for Deadlock:

Deadlock occurs when all of the following four conditions are met:

1. **Mutual Exclusion**: Resources are non-shareable, meaning only one thread can hold and use a resource at a time.
2. **Hold and Wait**: A thread holds at least one resource and is waiting to acquire additional resources held by other threads.
3. **No Preemption**: Resources cannot be forcibly taken from threads. A thread must release the resources voluntarily.
4. **Circular Wait**: A set of threads exists where each thread is waiting for a resource held by the next thread in the set, creating a cycle.

Example of Deadlock:

Imagine two threads, **Thread 1** and **Thread 2**, and two resources, **Resource A** and **Resource B**.

- **Thread 1** holds **Resource A** and requests **Resource B**.
- **Thread 2** holds **Resource B** and requests **Resource A**.

At this point:

- **Thread 1** is waiting for **Resource B**, which is held by **Thread 2**.
- **Thread 2** is waiting for **Resource A**, which is held by **Thread 1**.

Neither thread can proceed because each is waiting for the other to release a resource, causing a **circular wait** and leading to a **deadlock**. Neither thread can continue, and the program is stuck indefinitely.

3] Compare different approaches to implementing threads in Java: extending Thread and implementing Runnable

In Java, there are two primary ways to implement threads: **extending the Thread class** and **implementing the Runnable interface**.

1. Extending the Thread class

When you extend the Thread class, you create a new class that inherits the properties and methods of the Thread class. You override the run() method to define the code that should execute in the new thread.

Advantages:

- **Simple to use**: You only need to extend the Thread class and override its run() method.
- **Direct access to Thread methods**: You get direct access to Thread methods like start(), sleep(), join(), etc.

Example

```
class MyThread extends Thread {
   public void run() {
      System.out.println("Thread is running.");
   }
}
public class Main {
   public static void main(String[] args) {
      MyThread t = new MyThread();
      t.start();  // Starts the thread and calls run()
   }
}
```

2. Implementing the Runnable interface

When you implement the Runnable interface, you define a run() method in a separate class that implements Runnable. You then pass an instance of this class to a Thread object, which starts the thread.

```
class MyRunnable implements Runnable {
   @Override
   public void run() {
      System.out.println("Runnable is running.");
   }
}

public class Main {
   public static void main(String[] args) {
      MyRunnable task = new MyRunnable();
      Thread t = new Thread(task);
      t.start();  // Starts the thread and calls run()    } }
```

Comparison Table

Feature	Extending `Thread`	Implementing `Runnable`
Inheritance	Can only extend `Thread`, limiting other inheritance	Can implement other interfaces or extend other classes
Flexibility	Less flexible, tied to `Thread` class	More flexible, separate task and thread management
Task Reusability	Can only run one task per `Thread` object	Can reuse the same `Runnable` in multiple `Thread` objects
Code Separation	Thread and task are in the same class	Task and thread management are separated
Thread Management	Direct access to `Thread` methods	Must create a `Thread` object to start the thread
Recommended Usage	Simple tasks when no inheritance is required	Complex tasks, better for modular and scalable designs

4] Describe the lifecycle of a thread in Java. Provide examples to explain each state.

The lifecycle of a thread in Java represents the different stages a thread goes through from when it is created until it finishes its execution. Let's break down these stages in simple terms with examples

1. New (Born) State

A thread is in the **New** state when it is created but hasn't started yet. At this point, the thread is not running.

Example

```
Thread t = new Thread(); // The thread is in the "New" state
```

2. Runnable (Ready to Run) State

A thread moves to the **Runnable** state when you call the start() method. This means the thread is ready to run, but it has to wait for the CPU to assign it some time to actually start executing.

Example :

```
Thread t = new Thread(() -> {
  System.out.println("Thread is running");
});
t.start(); // The thread is now in the "Runnable" state
```

3. Blocked (Waiting for Resources) State

A thread enters the **Blocked** state when it is waiting to acquire a lock or resource (such as a file or shared memory) that is currently held by another thread.

Example :

```
synchronized (lock) {
  // Thread will be blocked here if another thread is holding the lock
}
```

4. Waiting (Indefinite Waiting) State

A thread moves to the **Waiting** state when it is waiting for another thread to notify it or when it needs to wait for some condition to be met (like waiting for a specific signal). This state is used when the thread is waiting indefinitely

```
synchronized (lock) {
   try {
      lock.wait(); // Thread enters the "Waiting" state
   } catch (InterruptedException e) {
      e.printStackTrace();
   }
}
```

5. Timed Waiting State

A thread enters the **Timed Waiting** state when it is waiting for a specific period. After this time, the thread will automatically return to the **Runnable** state.

Example:

```
Thread.sleep(1000);  // The thread will sleep for 1 second (1000 milliseconds)
```

6. Terminated (Dead) State

A thread enters the **Terminated** state when its task (i.e., the run() method) has finished executing. Once a thread is terminated, it cannot be started again.

Example

```
Thread t = new Thread(() -> {
   System.out.println("Thread has finished executing");
});
t.start(); // Thread starts
// Once the thread finishes its task, it enters the "Terminated" state.
```

Summary of Thread States:

1. **New**: The thread is created but hasn't started yet.
2. **Runnable**: The thread is ready to run, waiting for CPU time.
3. **Blocked**: The thread is waiting for a resource or lock.
4. **Waiting**: The thread is waiting indefinitely for a signal or condition.
5. **Timed Waiting**: The thread is waiting for a specific amount of time.
6. **Terminated**: The thread has finished executing.

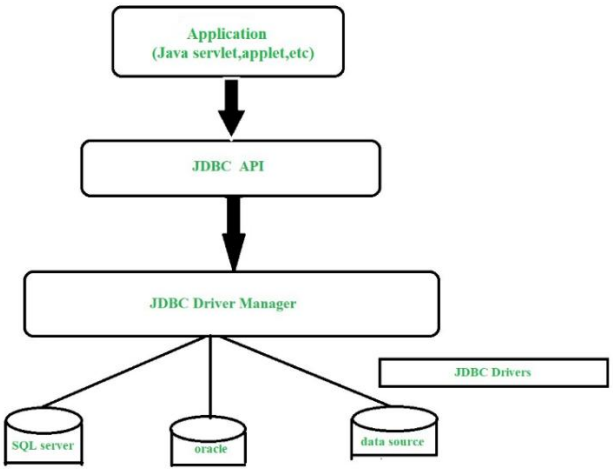
Module 3 : AWT, SWING AND JDBC(java Database Connectivity)

Q1] Design a simple GUI application using AWT and Swing components.

```
import javax.swing.*;

class Example extends JFrame {
   JLabel l1, l2;
   JTextField t1, t2;
   JButton btn;

   // Constructor
   Example() {
      // Set layout to null to use absolute positioning
      this.setLayout(null);

      // Initialize components
      l1 = new JLabel("Enter Name:");
      l2 = new JLabel("Enter Password:");
      t1 = new JTextField();
      t2 = new JTextField();
      btn = new JButton("Submit");

      // Set bounds for components
      l1.setBounds(20, 50, 100, 30);
      t1.setBounds(150, 50, 120, 30);
      l2.setBounds(20, 100, 100, 30);
      t2.setBounds(150, 100, 120, 30);
      btn.setBounds(100, 150, 100, 30);

      // Add components to the frame
      this.add(l1);
      this.add(t1);
      this.add(l2);
      this.add(t2);
      this.add(btn);
   }
}

class demo {
   public static void main(String[] args) {
      // Create an instance of the Example class
      Example obj = new Example();

      // Set properties for the JFrame
      obj.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        obj.setSize(400, 300); // Set the frame size
        obj.setVisible(true); // Make it visible
        obj.setTitle("Login Form"); // Set the title
        obj.setLocationRelativeTo(null); // Center the frame on the screen
    }
}
```

Q2] Describe the JDBC architecture and API. How does JDBC facilitate database connectivity?



JDBC Architecture

Java Database Connectivity (JDBC) is a standard Java API that allows applications to interact with relational databases. It serves as a bridge between Java applications and the database, enabling seamless data access and manipulation. The JDBC architecture is composed of two key layers:

---

1. JDBC API (Application Layer):

- Provides a set of classes and interfaces (e.g., Connection, Statement, ResultSet) for Java developers to perform database operations such as querying, updating, and managing data.
- Acts as an abstraction layer, making the code database-independent

2. JDBC Driver API (Driver Layer):

- Provides the implementation to connect to a specific database using its protocol.

Four types of JDBC drivers exist:

1. **Type-1 Driver (JDBC-ODBC Bridge):**
   The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls.
2. **Type-2 Driver (Native API):** The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.
3. **Type-3 Driver(Network Protocol Driver):**

- Uses middleware to convert JDBC calls into database-specific calls.
- Provides better performance and flexibility.

4. **Type-4 Driver (Thin Driver) :**
   - Directly converts JDBC calls to database-specific protocol without middleware.
   - Platform-independent and widely used.

How JDBC Facilitates Database Connectivity:

- **Establishes a Connection**: JDBC connects Java applications to a database using a specific driver and credentials.
- **Executes SQL Queries**: You can send SQL commands (e.g., SELECT, INSERT) to the database using JDBC.
- **Fetches Data**: The results of the query are retrieved in a ResultSet for further processing.
- **Database Independence**: JDBC allows you to switch between different databases easily by just changing the database driver.

Q3] Explain the process of executing SQL queries in Java using JDBC. Illustrate with an example program.

Steps to Execute SQL Queries:

1. **Load the JDBC Driver:** Load the JDBC driver class to establish a connection with the database.

   ```
   Class.forName("org.postgresql.Driver");
   ```

2. **Establish a Database Connection:** Use the DriverManager to obtain a Connection object.

   ```
   Connection con =
   DriverManager.getConnection("jdbc:postgresql://localhost:5432/DatabaseName",
   "username", "password");
   ```

3. **Create a Statement Object:** Create a Statement or PreparedStatement object to execute SQL queries.

   ```
   Statement stmt = con.createStatement();
   ```

4. **Execute SQL Queries:** Use the statement object to execute queries:
   - For DDL/DML queries (e.g., INSERT, UPDATE, DELETE), use `executeUpdate()`.
   - For SELECT queries, use `executeQuery()`.
5. **Process the Result:** For executeQuery(), iterate through the ResultSet to retrieve data.

   ```
   ResultSet rs = stmt.executeQuery("SELECT * FROM tablename");
   while (rs.next()) {
     System.out.println("ColumnValue: " + rs.getString("columnName"));
   }
   ```

6. **Close Resources:** Always close the ResultSet, Statement, and Connection objects to free resources.

   ```
   rs.close();
   stmt.close();
   con.close();
   ```

Example Program :

Below is an example program that demonstrates executing a SQL query using JDBC:

```
import java.sql.*;

public class JdbcExample {
   public static void main(String[] args) {
      // Database URL, username, and password
      String url = "jdbc:postgresql://localhost:5432/BankMate";
      String username = "postgres";
      String password = "postgres";

      // SQL query
      String query = "SELECT * FROM dbuser";

      Connection con = null;
      Statement stmt = null;
      ResultSet rs = null;

      try {
         // Step 1: Load JDBC driver
         Class.forName("org.postgresql.Driver");

         // Step 2: Establish connection
         con = DriverManager.getConnection(url, username, password);
         System.out.println("Connected to the database!");

         // Step 3: Create statement
         stmt = con.createStatement();

         // Step 4: Execute query
         rs = stmt.executeQuery(query);

         // Step 5: Process the result
         System.out.println("ID | First Name | Last Name | Email");
         while (rs.next()) {
            int id = rs.getInt("dbuser");
            String firstName = rs.getString("ufname");
            String lastName = rs.getString("ulname");
            String email = rs.getString("uemail");
            System.out.println(id + " | " + firstName + " | " + lastName + " | " + email);
         }
      } catch (ClassNotFoundException e) {
         System.err.println("PostgreSQL JDBC Driver not found: " + e.getMessage());
      } catch (SQLException e) {
         System.err.println("Database error: " + e.getMessage());
      } finally {
         try {
            // Step 6: Close resources
            if (rs != null) rs.close();
            if (stmt != null) stmt.close();
```

```
            if (con != null) con.close();
            System.out.println("Database resources closed.");
        } catch (SQLException e) {
            System.err.println("Error closing resources: " + e.getMessage());
        }
    }
  }
}
```

Q4] Discuss the role of PreparedStatement and CallableStatement in JDBC. How do they differ from Statement?

Module 4 :

Q1] What is session management in Java web applications? Explain various techniques used for session management.

Session : Session is used to save user information . It starts from the instance the user logs into the application and remains till the user logs out of the application or shuts down the machine. In both cases, the session values are deleted automatically

Session Management : Session management in Java web applications refers to the process of maintaining a user's state (data) across multiple requests and responses in a stateless HTTP protocol. It ensures that the server can recognize the same user during a single session

Why is Session Management Needed?

- HTTP is **stateless**, meaning each request is treated as a new one, and the server doesn't automatically remember the user.
- Session management helps maintain user-specific data (e.g., login status, shopping cart) .

Techniques for Session Management

1. **Cookies**
    - Cookies are small pieces of data stored on the client's browser.
    - Cookies can store session identifiers or user data.
2. **URL Rewriting**
    - Appending the session information to the URL as a query parameter(eg. ? sessionID=12345)
    - Works even if cookies are disabled.
3. **httpSession(Server-Side Sessions)**
    - A server-side mechanism that associates user data with a unique session id
    - Can store large and complex data
4. **Database-Based Session Management**
    - Storing session data in a database instead of server memory.
    - The ID is shared with the client via cookies

Q2] Introduce Java Server Pages (JSP). How is JSP different from servlets?

Java Server Pages (JSP) is a server-side technology that enables developers to create dynamic web content. It is part of the Java EE platform and simplifies the creation of web pages by embedding Java code directly into HTML. JSP files have the extension .jsp and are compiled into servlets (Java classes) by the server during execution

Key Features

Ease of Use: JSP allows embedding Java code within HTML using special tags like <% %>

Reusability: Developers can use custom tags and JavaBeans to modularize the code.

Aspect	Servlets	JSP
Nature	Purely Java code encapsulated in a class.	HTML-based with embedded Java code.
Ease of Development	More complex for creating dynamic web pages, as HTML needs to be written using `out.print()` statements.	Easier for creating web pages since Java code can be directly embedded in HTML.
Syntax	Java-centric syntax (requires explicit handling of HTML and UI generation).	HTML-centric syntax with Java snippets embedded.
Compilation	Manually written and compiled Java classes.	Automatically compiled into servlets by the server.
Separation of Concerns	Not inherently designed for separating business logic and presentation.	Encourages separation by using JavaBeans and custom tags for logic.
Use Case	Best for handling complex backend processing and request/response manipulation.	Best for designing the front-end and dynamic content.
Performance	Slightly better performance as they are directly written as Java classes.	Initial compilation overhead as JSPs are first converted to servlets.
Custom Tags Support	No native support for custom tags.	Supports custom tags and tag libraries for reusable components.
File Extension	`.java`	`.jsp`

Q3] What are JSP tags? Discuss the role of scriptlets and expression language in JSP .

JSP tags are special constructs in Java Server Pages (JSP) used to embed Java code and perform specific operations within an HTML document. These tags enable developers to add dynamic behavior to web pages. JSP tags fall into three main categories:

1. **Directive Tags**: Define global settings for the JSP page.
    - Syntax: <%@ directive attribute="value" %>
    - <%@ include file="header.jsp" %>: Includes a static file at compile time.
2. **Declaration Tags**: Define methods or variables that can be used throughout the JSP page.
    - Syntax: `<%! Code %>`
    - Example: <%! int counter = 0; %>
3. **Scriptlet Tags**: Embed Java code directly into a JSP page.
    - Syntax: <% code %>
    - Example: <% out.println("Hello, User!"); %>

Role of Scriptlets in JSP

Scriptlets (`<% code %>`) are used to embed raw Java code within a JSP file. They allow developers to manipulate request data, session objects, or other server-side resources directly.

Example :

```
<%
   String name = request.getParameter("username");
   if (name == null) {
      name = "Guest";
   }
%>
<p>Welcome, <%= name %>!</p>
```

Key Points:

- Allows embedding of complex Java logic.
- Often used for processing user requests and generating dynamic content.
- Overuse can make JSP pages harder to maintain and less readable.

Role of Expression Language (EL) in JSP

Expression Language (EL) is a simplified syntax introduced in JSP 2.0 to replace scriptlets for accessing Java objects and attributes. It enhances readability and maintainability by reducing the need for raw Java code in JSP files.

Syntax:

- ${expression}: Evaluates the expression and outputs the result.

Example :

```
<p>Welcome, ${requestScope.username != null ? requestScope.username : 'Guest'}!</p>
```

Q4] . Explain the Model-View-Controller (MVC) architecture in Java web applications. Provide an example

The **Model-View-Controller (MVC)** architecture is a design pattern used to separate an application into three interconnected components. This separation helps manage complexity and makes the application easier to maintain and scale.

Components of MVC

1. **Model:**
   - Represents the application's data and business logic.
   - Responsible for retrieving, updating, and managing the data.
   - Directly interacts with the database or other data sources.

   Example :

   In a bookstore application, the Controller would handle actions such as searching for a book, adding a book to the cart, or checking out.
2. **View:**
   - Responsible for the presentation layer.
   - Displays data to the user.
   - Does not contain business logic; it relies on data passed from the controller.

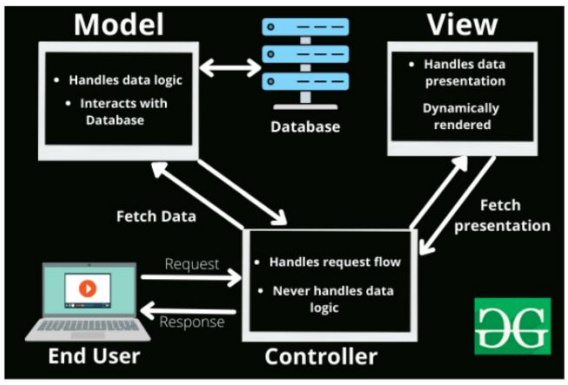
- Example: JSP pages, HTML, or Thymeleaf templates.

  Example: In a bookstore application, the View would display the list of books, book details, and provide input fields for searching or filtering books.

3. **Controller:**

- Acts as an intermediary between the Model and View.
- Handles user input, processes it, and decides which view to display.
- Responsible for invoking appropriate methods in the Model.
- Example: Servlets, Spring Controllers, or Struts Actions.

  Example: In a bookstore application, the Model would handle data related to books, such as the book title, author, price, and stock level.



MVC Architecture Design

Module 5 : Spring

Q1] What is dependency injection (DI)? Discuss its types and significance in Spring.

Answer:

Dependency Injection (DI) is a technique in programming where a class is **given (injected)** the objects it needs (its dependencies) from the outside, instead of **creating them itself**.

Example

Suppose class One needs the object of class Two to instantiate or operate a method, then class One is said to be **dependent** on class Two. Now though it might appear okay to depend on a module on the other, in the real world, this could lead to a lot of problems, including system failure. Hence such dependencies need to be avoided. Spring IOC resolves such dependencies with Dependency Injection, which makes the code **easier to test and reuse**

Type

1 . Constructor Injection Using XML

- Inject dependencies through the constructor.

Example

Example class

```
public class Engine {
   public void start() {
      System.out.println("Engine started!");
   }
}

public class Car {
   private Engine engine;

   public Car(Engine engine) { // Constructor Injection
      this.engine = engine;
   }

   public void drive() {
      engine.start();
      System.out.println("Car is driving!");
   }
}
```

Xml configuration

```
<beans …. >
<bean id=”engine” class=”engine”/>
<bean id=”car” class=”car”>
          <constructor-arg ref=”engine”>
```

```
</bean>
</bean>
```

Main class

```
Public class Main{
        Public static void main(String[] args){
                ApplicationContext context =new ClassPathXmlApplicationContext(“beans.xml”);
                Car car=(car)context.getBean(“car”);
                Car.drive();
}
}
```

2] Settle Injection

Inject dependencies using setter methods.

Example

Example class

```
Class engine{
        Public void start(){
                S.o.p(“Engine is started”);
}}
Class car{
        Private engine engine;
        Void setEngine(engine engine){
                This.engine=engine;
        }
        Void drive(){
                Engine.start();
                System.out.println(“Car id driving”);
        }
}
```

Beans.xml

```
        <bean …. >
                <bean id=”engine” class=”engine”/>
                <bean id=”car” class=”car”>
                        <property name=”engine” ref=”engine”>
                </bean>
        </bean>
```

Main.java

```
Class Main{
        Public static void main(String[] args){
                ApplicationContext context=new ClassPathXmlApplicationContext(“beans.xml”);
                Car obj=(car) context.getBean(“car”);
                Obj.drive();
        }
}
```

Q2] Explain the concept of inversion of control (IoC) in Spring. How is it implemented?

Spring IoC (Inversion of Control) Container is the core of Spring Framework. It creates the objects, configures and assembles their dependencies, manages their entire life cycle.

Example

Think of IoC as ordering food at a restaurant:

- **Without IoC**: You go into the kitchen, cook your food, and serve it yourself. (You manage everything.)
- **With IoC**: You tell the waiter (the container) what you need, and they bring it to you. (The container manages everything.)

Why IoC Is Useful

1. **Less Work**: You don't have to write code for creating or managing objects.
2. **Loose Coupling**: Your code doesn't depend on specific implementations, making it flexible and easier to change.
3. **Focus on Logic**: You can focus on solving problems while the framework handles the setup.

Implementation

Exaample

Example class

```
public class Engine {
   public void start() {
      System.out.println("Engine started!");
   }
}

public class Car {
   private Engine engine;

   public void setEngine(Engine engine) {
      this.engine = engine;  // Spring will inject Engine
   }

   public void drive() {
      engine.start();
      System.out.println("Car is driving!");
   }
}
```

Configuration bean in java class

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
@Configuration
public class AppConfig {

   @Bean
   public Engine engine() {
      return new Engine();  // Spring creates the Engine
   }

   @Bean
   public Car car() {
      Car car = new Car();
      car.setEngine(engine());  // Spring injects Engine into Car
      return car;
   }
}
```

Use the object

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {
   public static void main(String[] args) {
      // Initialize Spring container
      AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

      // Get the Car bean (object) that Spring created and injected
      Car car = context.getBean(Car.class);
      car.drive();  // Use the Car

      context.close();
   }
}
```

In Simple Words

1. **You define the objects (beans)**, but don't create them.
2. **You tell Spring how these objects are related** (like Car needs Engine).
3. **Spring creates and manages them** for you.
4. You **use the objects that Spring provides** with all dependencies set up.

Q3] What are Spring Beans? Discuss the bean lifecycle in the Spring framework.

What Are Spring Beans?

In the Spring framework, a **Bean** is simply an object that is managed by the Spring **IoC (Inversion of Control) Container**. Spring Beans are the building blocks of any Spring application. These are the objects that are created, configured, and managed by the Spring container.

- A Spring Bean can be any Java object, like a service, controller, or utility class.

- The Spring container takes care of **instantiating** the object, **injecting dependencies**, and managing its lifecycle.

Bean Lifecycle in Spring

The lifecycle of a Spring Bean refers to the various stages the bean goes through from its creation to its destruction. Here's a simplified explanation:

1. **Bean Instantiation**: The Spring container creates the bean using a constructor or factory method.
2. **Dependency Injection**: The container injects any dependencies (other beans) into the bean using constructor injection, setter injection, or field injection.
3. **Bean Initialization**:
    - If the bean implements InitializingBean or has a custom init-method, Spring will call the initialization method after dependencies are set.
4. **Bean Use**: The bean is used by the application for its functionality.
5. **Bean Destruction**:
    - When the application context is closed, Spring will call the destroy-method or the @PreDestroy annotation to clean up resources.

4] Write a program to demonstrate dependency injection in Spring using XML configuration

Code above code is valid (Engine and car)

Module 6 : Hibernate Framework (Java)

Q1] what is hibernate ? Explain its key features and benefits

Answer:

What is Hibernate?

Hibernate is a **Java framework** used to interact with databases. It simplifies the process of storing, retrieving, and managing data in a database by converting Java objects into database tables and vice versa. Hibernate helps developers write less SQL code and focus on Java programming.

Key Features of Hibernate

1. **Object-Relational Mapping (ORM):**
   - Hibernate maps Java objects to database tables using XML files or annotations. Each entity class corresponds to a table, and class fields map to table columns.
2. **Hibernate Query Language (HQL):**
   - A database-independent query language to perform operations on the data
3. **Lazy and Eager Fetching:**
   - Hibernate supports both lazy loading (loading data only when needed) and eager loading (loading data immediately).
4. **Automatic Table Generation:**
   - Hibernate can automatically generate database tables based on the entity classes, reducing the need for manual database schema creation.
5. **Support for Multiple Database**
   - Hibernate can work with various databases (MySQL, PostgreSQL, Oracle, etc.) by simply changing the configuration, without altering the application code
6. **Integration with Frameworks:**
   - Hibernate can be easily integrated with popular frameworks like Spring, enabling a more comprehensive development environment.

Benefits of Using Hibernate

1. **Simplifies Database Interaction:**
   - Developers can focus on writing Java code without worrying about SQL queries.
2. **Portability:**
   - Hibernate is database-independent, making it easier to switch databases without significant code changes.
3. **Improved Productivity:**
   - Automatic table generation, built-in transaction management, and HQL save development time.
4. **Performance Optimization:**
   - Features like caching and lazy loading improve performance by reducing database load.
5. **Scalability:**
   - Hibernate is designed to handle complex database schemas and large-scale applications efficiently.
6. **Maintenance and Debugging:**
   - Since Hibernate abstracts most SQL operations, maintaining and debugging applications becomes easier.

Q2] define object-relational mapping . how does hibernate implement ORM?

Object-Relational Mapping (ORM) is a method of linking Java classes to database tables. It allows developers to use Java objects to interact with database data instead of writing SQL queries directly. For example, a Customer class in Java can represent a customer table in the database.

1. Entity Class (Mapping Java Class to a Database Table)

This class represents the database table.
java
Copy code

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;

@Entity  // Marks this class as an entity linked to a database table
public class Customer {

   @Id  // Marks this field as the primary key
   private int id;
   private String name;
   private String email;

   // Getters and Setters
   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 String getEmail() {
      return email;
   }
   public void setEmail(String email) {
      this.email = email;
   }
}
```

---

2. Hibernate Configuration File (hibernate.cfg.xml)

This file configures Hibernate to connect to the database.

xml
Copy code

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="hibernate.connection.driver_class">org.postgresql.Driver</property>
    <property
name="hibernate.connection.url">jdbc:postgresql://localhost:5432/mydatabase</property>
    <property name="hibernate.connection.username">postgres</property>
    <property name="hibernate.connection.password">postgres</property>

    <!-- Hibernate settings -->
    <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.hbm2ddl.auto">update</property>

    <!-- Mapping class -->
    <mapping class="Customer"/>
  </session-factory>
</hibernate-configuration>
```

---

3. Main Class (Performing ORM Operations)

This class shows how to save a Customer object into the database.

java
Copy code

```java
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class HibernateExample {
  public static void main(String[] args) {
    // Create Hibernate configuration and session factory
    Configuration config = new Configuration();
    config.configure("hibernate.cfg.xml"); // Load configuration
    SessionFactory sessionFactory = config.buildSessionFactory();

    // Open a session
    Session session = sessionFactory.openSession();

    // Start a transaction
    Transaction transaction = session.beginTransaction();

    // Create a new Customer object
```

```
        Customer customer = new Customer();
        customer.setId(1);
        customer.setName("John Doe");
        customer.setEmail("john.doe@example.com");

        // Save the Customer object to the database
        session.save(customer);

        // Commit the transaction and close the session
        transaction.commit();
        session.close();

        System.out.println("Customer saved successfully!");
    }
}
```

How Hibernate Implements ORM

1. **Entity Classes:**
    - Hibernate uses Java classes (like User, Product) to represent database tables, and class variables (like id, name) represent table columns.
2. **Mapping Annotations:**
    - Hibernate uses special annotations (e.g., @Entity, @Table) or XML files to map classes and fields to database tables and columns.
3. **Session Factory:**
    - Hibernate creates a session to handle all database operations (like save, update, delete) using Java objects.
4. **HQL (Hibernate Query Language):**
    - Hibernate provides an easy-to-use query language to fetch and manage data without worrying about SQL syntax.

Q3] Describe the architecture of Hibernate. Discuss its components

Q4] How do you configure Hibernate? Explain the steps for setting up Hibernate in a Java application.

Add Hibernate Library

Include Hibernate and your database driver libraries in your project.

If using Maven:

Add this to your pom.xml:

xml

Copy code

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.6.15.Final</version>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
```

```
    <version>42.5.0</version>
</dependency>
```

---

2. Create Hibernate Configuration File

Add a hibernate.cfg.xml file in the src/main/resources folder.
This file contains your database connection details:
xml
Copy code

```
<hibernate-configuration>
    <session-factory>
        <property
name="hibernate.connection.url">jdbc:postgresql://localhost:5432/your_database</property>
        <property name="hibernate.connection.username">postgres</property>
        <property name="hibernate.connection.password">postgres</property>
        <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</property>
        <property name="hibernate.hbm2ddl.auto">update</property>
        <mapping class="com.example.model.User" />
    </session-factory>
</hibernate-configuration>
```

---

3. Create a Simple Entity Class

Create a Java class for your database table. For example:
java
Copy code

```
import javax.persistence.*;

@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String username;
    private String password;

    // Getters and setters
}
```

---

4. Write a Utility Class

Create a helper class to set up Hibernate:
java
Copy code

```
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
    private static SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
```

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

---

5. Perform Database Operations

Use Hibernate to save data into the database:

java

Copy code

```
import org.hibernate.Session;
import org.hibernate.Transaction;

public class Main {
    public static void main(String[] args) {
        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = session.beginTransaction();

        // Create a new user
        User user = new User();
        user.setUsername("john_doe");
        user.setPassword("password123");

        session.save(user);  // Save user to the database
        transaction.commit();
        session.close();
    }
}
```

---

6. Run the Application

- Hibernate will connect to the database, create tables (if hibernate.hbm2ddl.auto=update), and save the data.
- Check your database to see the table and data!

Key Points:

- **Dependencies**: Include Hibernate and database driver.
- **Configuration**: Use hibernate.cfg.xml for settings.
- **Entities**: Create Java classes that represent database tables.
- **Utility**: Use HibernateUtil to handle setup.
- **CRUD**: Use Hibernate sessions to perform operations.

Q5] Discuss CRUD operations in Hibernate. Illustrate with an example program.6

What are CRUD Operations?

1. **Create**: Add new data to the database.
2. **Read**: Fetch data from the database.
3. **Update**: Modify existing data in the database.
4. **Delete**: Remove data from the database.

Steps to Perform CRUD Operations

1. Create an Entity Class

This class represents a database table.

java

Copy code

```
import javax.persistence.*;

@Entity
public class User {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private int id;

   private String username;
   private String password;

   // Getters and Setters
   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }

   public String getUsername() {
      return username;
   }

   public void setUsername(String username) {
      this.username = username;
   }

   public String getPassword() {
      return password;
   }

   public void setPassword(String password) {
      this.password = password;
   }
}
```

---

2. Hibernate Configuration

Add a hibernate.cfg.xml file for database connection and settings.
xml
Copy code

```
<hibernate-configuration>
  <session-factory>
    <property
name="hibernate.connection.url">jdbc:postgresql://localhost:5432/your_database</property>
    <property name="hibernate.connection.username">postgres</property>
    <property name="hibernate.connection.password">postgres</property>
    <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</property>
    <property name="hibernate.hbm2ddl.auto">update</property>
    <mapping class="User" />
  </session-factory>
</hibernate-configuration>
```

---

3. Create Hibernate Utility
This class provides a reusable SessionFactory.
java
Copy code

```
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
  private static SessionFactory sessionFactory;

  static {
    sessionFactory = new Configuration().configure().buildSessionFactory();
  }

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

---

4. Perform CRUD Operations
Write a simple program to perform CRUD operations.
java
Copy code

```
import org.hibernate.Session;
import org.hibernate.Transaction;

public class HibernateCRUD {
  public static void main(String[] args) {
    // Create
    createUser("john_doe", "password123");

    // Read
    User user = getUserById(1);
```

```
    if (user != null) {
        System.out.println("User: " + user.getUsername());
    }
    // Update
    updateUser(1, "john_updated", "newpassword123");

    // Delete
    deleteUser(1);
}

// Create a new user
public static void createUser(String username, String password) {
    Session session = HibernateUtil.getSessionFactory().openSession();
    Transaction transaction = session.beginTransaction();

    User user = new User();
    user.setUsername(username);
    user.setPassword(password);

    session.save(user);
    transaction.commit();
    session.close();
    System.out.println("User created!");
}

// Read user by ID
public static User getUserById(int id) {
    Session session = HibernateUtil.getSessionFactory().openSession();
    User user = session.get(User.class, id);
    session.close();
    return user;
}

// Update user
public static void updateUser(int id, String newUsername, String newPassword) {
    Session session = HibernateUtil.getSessionFactory().openSession();
    Transaction transaction = session.beginTransaction();

    User user = session.get(User.class, id);
    if (user != null) {
        user.setUsername(newUsername);
        user.setPassword(newPassword);
        session.update(user);
        transaction.commit();
        System.out.println("User updated!");
    } else {
        System.out.println("User not found!");
    }
```

```
        session.close();
    }

    // Delete user
    public static void deleteUser(int id) {
        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = session.beginTransaction();

        User user = session.get(User.class, id);
        if (user != null) {
            session.delete(user);
            transaction.commit();
            System.out.println("User deleted!");
        } else {
            System.out.println("User not found!");
        }
        session.close();
    }
}
```