MALLA REDDY UNIVERSITY

School of Engineering - AI&ML

B.Tech. II Year - II Semester

Question Bank -Schema of Evaluation

Subject Code: **MR22-1CS0203** Subject Name: **Java Programming**

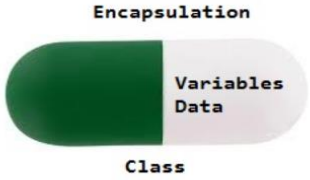
UNIT-I

Q. No.	Question
1	Explain following features of OOP with suitable examples:

a) **Encapsulation**

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is that it is a protective shield that prevents the data from being accessed by the code outside this shield.

- Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of its own class in which it is declared.
- As in encapsulation, the data in a class is hidden from other classes using the data hiding concept which is achieved by making the members or methods of a class private, and the class is exposed to the end-user or the world without providing any details behind implementation using the abstraction concept, so it is also known as a combination of data-hiding and abstraction.
- Encapsulation can be achieved by declaring all the variables in the class as **private** and writing **public** methods in the class to set and get the values of variables.
- It is more defined with the setter and getter method.



Advantages of Encapsulation

Data Hiding: it is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding. The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is

storing values in the variables. The user will only know that we are passing the values to a setter method and variables are getting initialized with that value.

Increased Flexibility: We can make the variables of the class read-only or write-only depending on our requirements. If we wish to make the variables read-only then we have to omit the setter methods like setName(), setAge(), etc. from the above program or if we wish to make the variables write-only then we have to omit the get methods like getName(), getAge(), etc. from the above program

Reusability: Encapsulation also improves the re-usability and is easy to change with new requirements.

Testing code is easy: Encapsulated code is easy to test for unit testing.

Freedom to programmer in implementing the details of the system: This is one of the major advantages of encapsulation that it gives the programmer freedom in implementing the details of a system. The only constraint on the programmer is to maintain the abstract interface that outsiders see.

Disadvantages of Encapsulation in Java

- Can lead to increased complexity, especially if not used properly.
- Can make it more difficult to understand how the system works.
- May limit the flexibility of the implementation.

Implementation of Java Encapsulation

```
class Person {
    // Encapsulating the name and age only approachable and used using  methods defined
    private String name;
    private int age;
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
     public int getAge() { return age; }
     public void setAge(int age) { this.age = age; }
}
 // Driver Class
public class Main {
    // main function
    public static void main(String[] args)
    {
```

```
        // person object created
        Person person = new Person();
        person.setName("John");
        person.setAge(30);
         // Using methods to get the values from the variables
        System.out.println("Name: " + person.getName());
        System.out.println("Age: " + person.getAge());
    }
}
```

Output

Name: John

Age: 30

b) **Abstraction**

Abstraction in Java is the process in which we only show essential details/functionality to the user. The non-essential implementation details are not displayed to the user.

Simple Example to understand Abstraction:

Television remote control is an excellent example of abstraction. It simplifies the interaction with a TV by hiding the complexity behind simple buttons and symbols, making it easy without needing to understand the technical details of how the TV functions.

- The abstraction is achieved by **interfaces** and **abstract classes**. We can achieve 100% abstraction using interfaces.
- Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

Abstract classes and Abstract methods

- An abstract class is a class that is declared with an **abstract** keyword.
- An abstract method is a method that is declared **without implementation**.
- An abstract class may or may not have all abstract methods. Some of them can be concrete methods
- A method-defined abstract must always be redefined in the subclass, thus making overriding compulsory or making the subclass itself abstract.

- Any class that contains one or more abstract methods must also be declared with an abstract keyword.
- There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the new operator.
- An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

Abstraction Example

```
abstract class Animal {
   private String name;
  public Animal(String name) { this.name = name; }
   public abstract void makeSound();
   public String getName() { return name; }
}
// Abstracted class
class Dog extends Animal {
   public Dog(String name) { super(name); }
   public void makeSound()
   {
      System.out.println(getName() + " barks");
   }
}
// Abstracted class
class Cat extends Animal {
   public Cat(String name) { super(name); }
   public void makeSound()
   {
      System.out.println(getName() + " meows");
   }
}
// Driver Class
public class AbstractionExample {
   // Main Function
   public static void main(String[] args)
```

```
    {
        Animal myDog = new Dog("Buddy");
        Animal myCat = new Cat("Fluffy");

        myDog.makeSound();
        myCat.makeSound();
    }
}
```

Output

Buddy barks

Fluffy meows

Explanation of the above Java program:

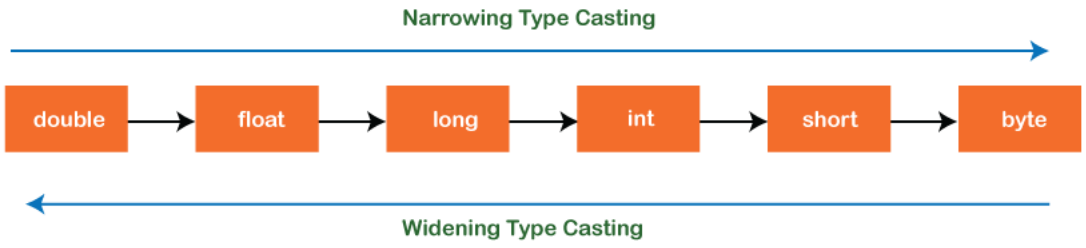
This code defines an Animal abstract class with an abstract method makeSound(). The Dog and Cat classes extend Animal and implement the makeSound() method. The main() method creates instances of Dog and Cat and calls the makeSound() method on them.

This demonstrates the abstraction concept in Java, where we define a template for a class (in this case Animal), but leave the implementation of certain methods to be defined by subclasses (in this case makeSound()).

2 Infer Type Casting. List and explain types of type casting methods with suitable example.

The type casting is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer.

Definition: Convert a value from one data type to another data type is known as type casting.



Types of Type Casting

- Widening Type Casting
- Narrowing Type Casting

Widening Type Casting

Converting a lower data type into a higher one is called widening type casting. It is also known as

implicit conversion or casting down. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

- Both data types must be compatible with each other.
- The target type must be larger than the source type.

byte -> short -> char -> int -> long -> float -> double

WideningTypeCastingExample.java

```
public class WideningTypeCastingExample
{
public static void main(String[] args)
{
int x = 7;
//automatically converts the integer type into long type
long y = x;
//automatically converts the long type into float type
float z = y;
System.out.println("Before conversion, int value "+x);
System.out.println("After conversion, long value "+y);
System.out.println("After conversion, float value "+z);
}
}
```

Output

```
Before conversion, the value is: 7
After conversion, the long value is: 7
After conversion, the float value is: 7.0
```

Narrowing Type Casting

Converting a higher data type into a lower one is called narrowing type casting. It is also known as explicit conversion or casting up. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

double -> float -> long -> int -> char -> short -> byte

NarrowingTypeCastingExample.java

```
public class NarrowingTypeCastingExample
{
public static void main(String args[])
```

```
{
double d = 166.66;
//converting double data type into long data type
long l = (long)d;
//converting long data type into int data type
int i = (int)l;
System.out.println("Before conversion: "+d);
//fractional part lost
System.out.println("After conversion into long type: "+l);
//fractional part lost
System.out.println("After conversion into int type: "+i);
}
}
```

Output

```
Before conversion: 166.66
After conversion into long type: 166
After conversion into int type: 166
```

3 Write a short note on control statements in java with example.

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

Decision Making statements

if statements

switch statement

Loop statements

do while loop

while loop

for loop

for-each loop

Jump statements

break statement

continue statement

Decision-Making statements:

The decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

If Statement:

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

- Simple if statement
- if-else statement
- if-else-if ladder
- Nested if-statement

Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement is given below.

```
if(condition) {
statement 1; //executes when condition is true
}
```

Example:

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y > 20) {
System.out.println("x + y is greater than 20");
}
}
}
```

Output:

```
x + y is greater than 20
```

if-else statement

The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
if(condition) {
statement 1; //executes when condition is true
}
else{
statement 2; //executes when condition is false
}
```

Example:

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y < 10) {
System.out.println("x + y is less than      10");
}   else {
System.out.println("x + y is greater than 20");
}
}
}
```

Output:

```
x + y is greater than 20
```

if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

Syntax of if-else-if statement is given below.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
}
```

```
else if(condition 2) {
statement 2; //executes when condition 2 is true
}
else {
statement 2; //executes when all the conditions are false
}
```

Example:

```
public class Student {
public static void main(String[] args) {
String city = "Delhi";
if(city == "Meerut") {
System.out.println("city is meerut");
}else if (city == "Noida") {
System.out.println("city is noida");
}else if(city == "Agra") {
System.out.println("city is agra");
}else {
System.out.println(city);
}
}
}
```

Output:

Delhi

Nested if-statement

In nested if-statements, the if statement can contain a if or if-else statement inside another if or else-if statement.

Syntax of Nested if-statement is given below.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
if(condition 2) {
statement 2; //executes when condition 2 is true
}
else{
```

```
statement 2; //executes when condition 2 is false
}
}
```

Example:

```
public class Student {
public static void main(String[] args) {
String address = "Delhi, India";
 if(address.endsWith("India")) {
if(address.contains("Meerut")) {
System.out.println("Your city is Meerut");
}else if(address.contains("Noida")) {
System.out.println("Your city is Noida");
}else {
System.out.println(address.split(",")[0]);
}
}else {
System.out.println("You are not living in India");
}    }    }
```

Output:

Delhi

Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied.
- It is optional, if not used, next case is executed.

- While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

The syntax to use the switch statement is given below.

```
switch (expression){
  case value1:
   statement1;
   break;
  .
  .
  case valueN:
   statementN;
   break;
  default:
   default statement;
}
```

Example:

```
public class Student implements Cloneable {
public static void main(String[] args) {
int num = 2;
switch (num){
case 0:
System.out.println("number is 0");
break;
case 1:
System.out.println("number is 1");
break;
default:
System.out.println(num);
}
}
}
```

Output:

2

4 Illustrate following types of Constructors:

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, **memory for the object** is allocated in the memory.

- It is a special type of method which is used to initialize the object.
- Every time an object is created using the **new()** keyword, at least one constructor is called.
- It calls a default constructor if there is no constructor available the class. In such case, Java compiler provides a default constructor by default.

Rules for creating Java constructor

There are three rules defined for the constructor.

- Constructor name must be the same as its class name
- A Constructor must have no explicit return type
- A Java constructor cannot be abstract, static, final, and synchronize

Key Points:

- It is called constructor because it constructs the values at the time of object creation.
- It is not necessary to write a constructor for a class.
- It is because java compiler creates a default constructor if your class doesn't have any.
- We can use access modifiers while declaring a constructor.
- It controls the object creation.
- In other words, we can have private, protected, public or default constructor in Java.

a)**Default Constructor**

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

<class_name>(){}

Example:

```
class Bike1{
//creating a default constructor
Bike1(){System.out.println("Bike is created");}
//main method
public static void main(String args[]){
//calling a default constructor
Bike1 b=new Bike1();
}
}
```

Output:

Bike is created

If there is no constructor in a class, compiler automatically creates a default constructor.

b)Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

The parameterized constructor is used to provide different values to distinct objects.

Example:

```
class Student{
   int id;
   String name;
   //creating a parameterized constructor
   Student(int i,String n){
   id = i;
   name = n;
   }
   //method to display the values
   void display(){System.out.println(id+" "+name);}
   public static void main(String args[]){
   //creating objects and passing values
   Student s1 = new Student(111,"Karan");
   Student s2 = new Student(222,"Aryan");
   //calling method to display the values of object
   s1.display();
   s2.display();
  }
}
```

Output:

111 Karan

222 Aryan

5 Summarize following access specifies in Java:

The access modifiers in Java specify the accessibility or scope of a field, method, constructor, or class. There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc.

There are four types of Java access modifiers:

- **Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
- **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
- **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
- **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

a)Public

The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.

Example:

```
//save by A.java
package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
//save by B.java
 package mypack;
import pack.*;
  class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Output: Hello

b)Default

If you don't use any modifier, it is treated as default by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example:

```
//save by A.java
package pack;
class A{
  void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

The scope of class A and its method msg() is default so it cannot be accessed from outside the package.

c)Protected

- The protected access modifier is accessible within package and outside the package but through inheritance only.
- The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.
- It provides more accessibility than the default modifer.

Example:

```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
class B extends A{
  public static void main(String args[]){
   B obj = new B();
```

```
   obj.msg();
  } }
```

Output:Hello

d)Private

The private access modifier is accessible only within the class.

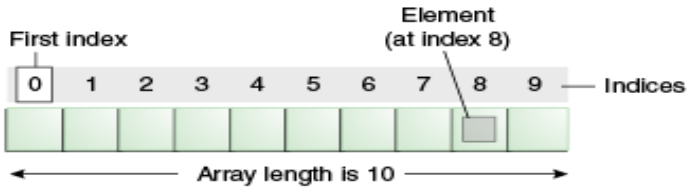
Example:

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}
 public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

6 How do you declare and access the array in java? Give an example.

Definition: An **array** is a collection of similar type of elements which has contiguous memory location.

- Java array is an object which contains elements of a similar data type.
- The elements of an array are stored in a contiguous memory location.
- It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.
- Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.
- In Java, array is an object of a dynamically generated class.
- Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces.



Advantages:

- ✓ **Code Optimization**: It makes the code optimized; we can retrieve or sort the data efficiently.
- ✓ **Random access:** We can get any data located at an index position.

Disadvantages:

- ✓ **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

There are two types of array.

- ➢ Single Dimensional Array
- ➢ Multidimensional Array

Single Dimensional Array

Syntax to Declare an Array

dataType[] arr; (or)

dataType []arr; (or)

dataType arr[];

Instantiation of an Array in Java

arrayRefVar=new datatype[size];

Example:

```
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Output:

10

20

70

40

50

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

int a[]={33,3,4,5};//declaration, instantiation and initialization

Example:

```
class Testarray1{
public static void main(String args[]){
int a[]={33,3,4,5};//declaration, instantiation and initialization
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

For-each Loop for Java Array

We can also print the Java array using for-each loop. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, and then executes the body of the loop.

The syntax of the for-each loop is given below:

```
for(data_type variable:array){
//body of the loop
}
```

Multidimensional Array:

Syntax to Declare Multidimensional Array in Java

dataType[][] arrayRefVar; (or)

dataType [][]arrayRefVar; (or)

dataType arrayRefVar[][]; (or)

dataType []arrayRefVar[];

Example to instantiate Multidimensional Array in Java

int[][] arr=new int[3][3];//3 row and 3 column

7 Brief the following concepts in Java:

a) **Constructor overloading**

In Java, we can overload constructors like methods. The constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task.

Example:

```
public class Student {
//instance variables of the class
int id;
String name;
Student(){
System.out.println("this a default constructor");
}
Student(int i, String n){
id = i;
name = n;
}
public static void main(String[] args) {
//object creation
Student s = new Student();
System.out.println("\nDefault Constructor values: \n");
System.out.println("Student Id : "+s.id + "\nStudent Name : "+s.name);
System.out.println("\nParameterized Constructor values: \n");
Student student = new Student(10, "David");
System.out.println("Student Id : "+student.id + "\nStudent Name : "+student.name);
}
}
```

Output:

```
this a default constructor
Default Constructor values:
Student Id : 0
Student Name : null
Parameterized Constructor values:
```

Student Id : 10

Student Name : David

b) **Constructor overriding**

In Java, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, the same parameters or signature, and the same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.

```
class Parent {
    Parent() {
        System.out.println("Parent class constructor");
    }
}
class Child extends Parent {
    Child() {
        // Calling parent class constructor explicitly
        super();
        System.out.println("Child class constructor");
    }
    Child(String message) {
        // Calling parameterized parent class constructor explicitly
        super();
        System.out.println(message);
    }
}
public class Main {
    public static void main(String[] args) {
        // Creating objects of Child class
        Child child1 = new Child();
        Child child2 = new Child("Constructor overriding example");
    }
}
```

In this example:

- There are two classes: Parent and Child.
- Child extends Parent, so Child is a subclass of Parent.
- Both Parent and Child have constructors.
- In the Child class, we define two constructors: one without parameters and one with a String parameter.
- In both constructors of the Child class, we explicitly call the constructor of the Parent class using the **super()** keyword. This is necessary because, by default, if you don't specify, the Java compiler will automatically insert a call to the default constructor of the superclass.
- In the main method of the Main class, we create objects of the Child class to demonstrate constructor overriding.

8 Write a Java Program to check whether a number is Prime or not

```
import java.util.Scanner;
public class PrimeChecker {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number to check if it's prime: ");
        int number = scanner.nextInt();

        if (isPrime(number)) {
            System.out.println(number + " is a prime number.");
        } else {
            System.out.println(number + " is not a prime number.");
        }
            scanner.close();
    }
    // Function to check if a number is prime
    public static boolean isPrime(int num) {
        if (num <= 1) {
            return false;
        }
        if (num <= 3) {
            return true;
        }
```

```
        if (num % 2 == 0 || num % 3 == 0) {
            return false;
        }
          // Check divisibility by numbers of the form 6k ± 1
        for (int i = 5; i * i <= num; i += 6) {
            if (num % i == 0 || num % (i + 2) == 0) {
                return false;
            }
        }
        return true;
    }}
```

Explanation:

- We prompt the user to enter a number to check if it's prime.
- The isPrime function takes an integer as input and returns true if it's a prime number, otherwise returns false.
- We first handle special cases where numbers less than or equal to 1 are not prime.
- We then handle cases where the number is 2 or 3, as they are prime.
- We then check divisibility by 2 and 3 to exclude multiples of these numbers.
- After that, we only check divisibility by numbers of the form 6k ± 1, as all other factors will be covered by these. This optimization reduces the number of iterations required.
- If the number is divisible by any of the tested numbers, we return false, indicating that it's not prime. Otherwise, we return true.

9 Write a Java Program to print Fibonacci Series with recursion

```
import java.util.Scanner;
public class FibonacciSeries {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of terms in the Fibonacci series: ");
        int n = scanner.nextInt();
        scanner.close();
        System.out.println("Fibonacci series:");
        for (int i = 0; i < n; i++) {
            System.out.print(fibonacci(i) + " ");
```

```
    }
  }
  // Recursive function to calculate the nth Fibonacci number
  public static int fibonacci(int n) {
    if (n <= 1) {
      return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
  }}
```

Explanation:

- We prompt the user to enter the number of terms they want in the Fibonacci series.
- We then call the **fibonacci** method for each term from 0 to n-1 and print the result.
- The **fibonacci** method is a recursive function that calculates the nth Fibonacci number.
- It checks if the given index is less than or equal to 1. If so, it returns the index itself because the Fibonacci series starts with 0 and 1.
- If the index is greater than 1, it recursively calls itself to calculate the sum of the previous two terms in the Fibonacci series.

10 Explore any four methods of a String class.

Strings are the type of objects that can store the character of values and in Java, every character is stored in 16 bits i,e using UTF 16-bit encoding. A string acts the same as an array of characters in Java.

Example:

String name = "CSE-AIML";

Ways of Creating a String

There are two ways to create a string in Java:

- String Literal- *String demo = "CSE-AIML";*
- Using new Keyword- *String demo1 = new String ("CSE-AIML");*

The java.lang.String class provides a lot of built-in methods that are used to manipulate string in Java. By the help of these methods, we can perform operations on String objects such as trimming, concatenating, converting, comparing, replacing strings etc.

Java String is a powerful concept because everything is treated as a String if you submit any form in window based, web based or mobile application.

Java String toUpperCase() and toLowerCase() method

The Java String toUpperCase() method converts this String into uppercase letter and String

toLowerCase() method into lowercase letter.

Example:

```
public class Stringoperation1
{
public static void main(String ar[])
{
String s="Sachin";
System.out.println(s.toUpperCase());//SACHIN
System.out.println(s.toLowerCase());//sachin
System.out.println(s);//Sachin(no change in original)
}
}
```

Output:

SACHIN

sachin

Sachin

String trim() method:

The String class trim() method eliminates white spaces before and after the String.

Example:

```
public class Stringoperation2
{
public static void main(String ar[])
{
String s="  Sachin  ";
System.out.println(s);//  Sachin
System.out.println(s.trim());//Sachin
}  }
```

Output:

Sachin

Sachin

String startsWith() and endsWith() method:

The method startsWith() checks whether the String starts with the letters passed as arguments and endsWith() method checks whether the String ends with the letters passed as arguments.

Example:

```
public class Stringoperation3
{
public static void main(String ar[])
{
String s="Sachin";
 System.out.println(s.startsWith("Sa"));//true
 System.out.println(s.endsWith("n"));//true
}  }
```

Output:

```
true
true
```

String charAt() Method:

The String class charAt() method returns a character at specified index.

Example:

```
public class Stringoperation4
{
public static void main(String ar[])
{
String s="Sachin";
System.out.println(s.charAt(0));//S
System.out.println(s.charAt(3));//h
}  }
```

Output:

```
S
h
```

String length() Method:

The String class length() method returns length of the specified String.

Example:

```
public class Stringoperation5
{
public static void main(String ar[])
{
```

```
String s="Sachin";
System.out.println(s.length());//6
}
}
```

Output:

```
6
```

S.No **Question**

11 Illustrate following types of inheritance with example:

Inheritance is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class. In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition, we can add new fields and methods to our current class as well.

The **extends** keyword is used for inheritance in Java. The extends keyword indicates we are derived from an existing class. In other words, "extends" refers to increased functionality.

Syntax :

```
class DerivedClass extends BaseClass
{
  //methods and fields
}
```

Usage:

Code Reusability: The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.

Method Overriding: Method Overriding is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.

Abstraction: The concept of abstract where we do not have to provide all details is achieved through inheritance. Abstraction only shows the functionality to the user.

a) **Multi-level**

When there is a chain of inheritance, it is known as multilevel inheritance.

Example:

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
```

```
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

Output:

```
weeping...
barking...
eating...
```

b) **Hybrid**

In general, the meaning of hybrid (mixture) is made of more than one thing. In Java, the hybrid inheritance is the composition of two or more types of inheritance. The main purpose of using hybrid inheritance is to modularize the code into well-defined classes. It also provides the code reusability.

Example 1:

```
class HumanBody
{
public void displayHuman()
{   System.out.println("Method defined inside HumanBody class");  }
}
interface Male
{  public void show();  }
interface Female
{  public void show();  }
public class Child extends HumanBody implements Male, Female
{
public void show()
{
System.out.println("Implementation of show() method defined in interfaces Male and Female");
}
public void displayChild()
```

```
{
System.out.println("Method defined inside Child class");
}
public static void main(String args[])
{
Child obj = new Child();
System.out.println("Implementation of Hybrid Inheritance in Java");
obj.show();
obj.displayChild();
}
}
```

Output:

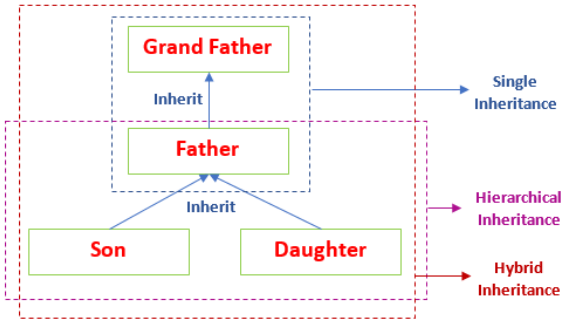
```
Implementation of Hybrid Inheritance in Java
Implementation of show() method defined in interfaces Male and Female
Method defined inside Child class
```

In this program, we have taken the example of Human body that performs different functionalities like eating, walking, talking, dancing etc. Human body may be either Male or Female. So, Male and Female are the two interfaces of the HumanBody class. Both the child class inherits the functionalities of the HumanBody class that represents the single Inheritance.

Suppose, Male and Female may have child. So, the Child class also inherits the functionalities of the Male, Female interfaces. It represents the multiple inheritance.

The composition of single and multiple inheritance represents the hybrid inheritance.

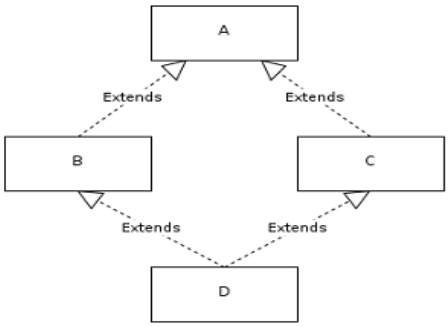
Example 2:



12 Justify "why Java does not support multiple Inheritance?"

Multiple inheritance is supported by C++, Python, and a few other languages, but not by Java. To prevent the **ambiguity** produced by multiple inheritance, Java does not allow it. The diamond problem, which happens in multiple inheritance, is one example of such a problem.

The Diamond Problem:



We have two classes B and C inheriting from A. Assume that B and C are overriding an inherited method and they provide their own implementation. Now D inherits from both B and C doing multiple inheritance. D should inherit that overridden method, which overridden method will be used? Will it be from B or C? Here we have an ambiguity.

Simplicity:

Dealing with the complexity caused by multiple inheritance is quite difficult. It causes issues during various operations like casting and constructor chaining, and all the above-mentioned reason is that there are very few circumstances in which multiple inheritance is required, thus it is preferable to avoid it to keep things simple and easy.

How to Achieve Multiple Inheritance in Java:

The problem arises when methods with similar signatures exist in both subclasses and superclasses. The compiler is unable to determine which class method should be called first or given priority when calling the method.

In Java, we can achieve multiple inheritance through the concept of **interface.** An interface is like a class that has variables and methods, however, unlike a class, the methods in an interface are abstract by default. Multiple inheritance through interface occurs in Java when a class implements multiple interfaces or when an interface extends multiple interfaces.

Example:

```
interface Dog {
void bark();
}
interface Cat {
void meow();
}
class Animal implements Dog, Cat {
public void bark() {
```

```
System.out.println("Dog is barking");
}
public void meow() {
System.out.println("Cat is meowing");
}
}
class Main {
public static void main(String args[]) {
Animal a = new Animal();
a.bark();
a.meow();
}
}
```

Output:

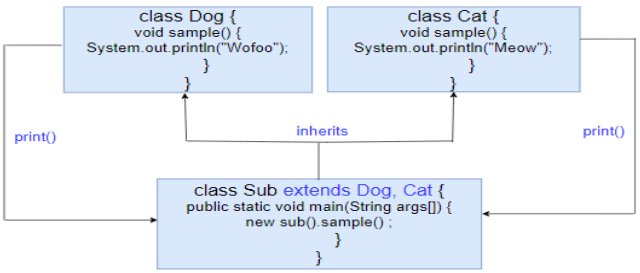
Dog is barking

Cat is meowing

Explanation:

Each interface, Dog and Cat, has one abstract method, i.e., bark() and meow(), respectively. The Animal class implements the interfaces Dog and Cat.

The main() method of the Demo class creates an object of the Animal class. Then, the bark() and meow() functions are called.



13 Write example programs for using following keywords in Java:

a) **"super" keyword**

The **super keyword** in Java is a **reference variable** which is used to refer immediate parent class object.

Whenever we create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

- super can be used to refer immediate parent class instance variable.
- super can be used to invoke immediate parent class method.
- super() can be used to invoke immediate parent class constructor.

super is used to refer immediate parent class instance variable.

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

Output:

```
black
white
```

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

Example:

```
class Animal{
void eat(){System.out.println("eating...");}
}
```

```
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}
```

Output:

```
eating...
barking...
```

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor.

Example:

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
```

```
Dog d=new Dog();
}}
```

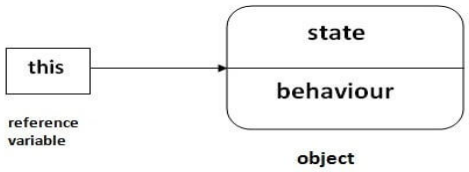
Output:

```
animal is created
dog is created
```

super() is added in each class constructor automatically by compiler if there is no super() or this().

b) **"this" key word**

There can be a lot of usage of Java this keyword. In Java, this is a reference variable that refers to the current object.



Usage of Java this keyword

- this can be used to refer current class instance variable.
- this can be used to invoke current class method (implicitly)
- this() can be used to invoke current class constructor.
- this can be passed as an argument in the method call.
- this can be passed as argument in the constructor call.
- this can be used to return the current class instance from the method.

this: to refer current class instance variable

The **this** keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

without this keyword:

Example:

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
rollno=rollno;
name=name;
fee=fee;
}
```

```
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis1{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

```
0 null 0.0
0 null 0.0
```

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

with this keyword:

Example:

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
```

}}

Output:

111 ankit 5000.0

112 sumit 6000.0

If local variables(formal arguments) and instance variables are different

this keyword not required:

Example:

```
class Student{
int rollno;
String name;
float fee;
Student(int r,String n,float f){
rollno=r;
name=n;
fee=f;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class TestThis3{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

111 ankit 5000.0

112 sumit 6000.0

this: to invoke current class method:

Example:

```
class A{
void m(){System.out.println("hello m");}
```

```
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
}
}
class TestThis4{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

Output:

```
hello n
hello m
```

14 Compare method overloading and method overriding in Java.

The differences between Method Overloading and Method Overriding in Java are as follows:

Method Overloading	**Method Overriding**
Method overloading is a compile-time polymorphism.	Method overriding is a run-time polymorphism.
Method overloading helps to increase the readability of the program.	Method overriding is used to grant the specific implementation of the method which is already provided by its parent class or superclass.
It occurs within the class.	It is performed in two classes with inheritance relationships.
Method overloading may or may not require inheritance.	Method overriding always needs inheritance.
In method overloading, methods must have the same name and different signatures.	In method overriding, methods must have the same name and same signature.

can or can not be the same, but we just have to change the parameter.	be the same or co-variant.
Static binding is being used for overloaded methods.	Dynamic binding is being used for overriding methods.
Poor Performance due to compile time polymorphism.	It gives better performance. The reason behind this is that the binding of overridden methods is being done at runtime.
Private and final methods can be overloaded.	Private and final methods can't be overridden.
The argument list should be different while doing method overloading.	The argument list should be the same in method overriding.

Example of Method Overloading:

```
import java.io.*;
class MethodOverloadingEx {
   static int add(int a, int b) { return a + b; }
   static int add(int a, int b, int c)
   {
      return a + b + c;
   }
   // Main Function
   public static void main(String args[])
   {
      System.out.println("add() with 2 parameters");
      // Calling function with 2 parameters
     System.out.println(add(4, 6));
      System.out.println("add() with 3 parameters");
      // Calling function with 3 Parameters
      System.out.println(add(4, 6, 7));
   }
}
```

Example of Method Overriding:

```
import java.io.*;
// Base Class
class Animal {
   void eat()
   {
      System.out.println("eat() method of base class");
      System.out.println("eating.");
   }
}
// Inherited Class
class Dog extends Animal {
   void eat()
   {
      System.out.println("eat() method of derived class");
      System.out.println("Dog is eating.");
   }
}
// Driver Class
class MethodOverridingEx {
   // Main Function
   public static void main(String args[])
   {
      Dog d1 = new Dog();
      Animal a1 = new Animal();
      d1.eat();
      a1.eat();
        // Polymorphism: Animal reference pointing to Dog object
      Animal animal = new Dog();
            // Calls the eat() method of Dog class
      animal.eat();
   }
}
```

15 Elucidate following types of polymorphism with suitable examples:

Polymorphism is derived from two Greek words, “poly” and “morph”, which mean “many” and “forms”, respectively. Hence, polymorphism meaning in Java refers to the ability of objects to take on many forms. In other words, it allows different objects to respond to the same message or method call in multiple ways.

Polymorphism allows coders to write code that can work with objects of multiple classes in a generic way without knowing the specific class of each object.

a) **Compile-time polymorphism**

Compile-time polymorphism, sometimes referred to as **static polymorphism** or **early binding**, is the capacity of a programming language to decide which method or function to use based on the quantity, kind, and sequence of inputs at compile-time. Method overloading, which enables the coexistence of many methods with the same name but distinct parameter lists within a class, enables Java to accomplish compile-time polymorphism.

Method Overloading

In Java, compile-time polymorphism is mostly achieved by method overloading. Programmers can use it to create numerous methods with the same name but different parameters that are all included within the same class. Depending on the arguments used during the method call, the compiler chooses the right method to call. Developers can create more expressive and flexible code by offering various implementations of a method with varied argument kinds or quantities.

Syntax and Usage

In order to overload a method in Java, multiple methods with the same name but distinct argument lists must be created. Either the parameter lists should contain a distinct number of parameters or a variety of parameters. The method that is called is chosen without regard to the return type.

Consider the following example:

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

In the above example, we have defined two add methods within the Calculator class. The first method takes two integers as arguments and returns their sum as an integer. The second method

takes two doubles as arguments and returns their sum as a double.

During compilation, the Java compiler analyzes the argument types used in the method call and determines the appropriate method to invoke. For instance:

```
Calculator calc = new Calculator();
int sum1 = calc.add(5, 10);          // Invokes the first add method
double sum2 = calc.add(2.5, 3.7);    // Invokes the second add method
```

Example:

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
    public double add(double a, double b) {
        return a + b;
    }
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int sum1 = calc.add(5, 10);
        System.out.println("Sum of 5 and 10 (integers): " + sum1);
        double sum2 = calc.add(2.5, 3.7);
        System.out.println("Sum of 2.5 and 3.7 (doubles): " + sum2);
    }
}
```

Output:

```
Sum of 5 and 10 (integers): 15
Sum of 2.5 and 3.7 (doubles): 6.2
```

b) **Run-time polymorphism**

Runtime polymorphism or **Dynamic Method** Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time. In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Method overriding is an example of runtime polymorphism. In method overriding, a subclass overrides a method with the same signature as that of in its superclass. During compile time, the check is made on the reference type. However, in the runtime, JVM figures out the object type

and would run the method that belongs to that particular object.

Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:

```
class A{}
class B extends A{}
A a=new B();//upcasting
```

For upcasting, we can use the reference variable of class type or an interface type. For Example:

```
interface I{}
class A{}
class B extends A implements I{}
```

Example:

```
class Bike{
  void run(){System.out.println("running");}
}
class Splendor extends Bike{
  void run(){System.out.println("running safely with 60km");}

  public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run();
  }
}
```

Output:

running safely with 60km.

16 How do you declare and implement an abstract class in Java?

Abstract Classes

Sometimes, a class that you define represents an abstract concept and as such, should not be instantiated. Take, for example, food in the real world. Have you ever seen an instance of food? No. What you see instead are instances of carrot, apple, and (my favorite), chocolate. Food represents the abstract concept of things that we can eat. It doesn't make sense for an instance of food to exist.

Similarly in object-oriented programming, you may want to model abstract concepts but you don't want to be able to create an instance of it. For example, the Number class in the java.lang package

represents the abstract concept of numbers. It makes sense to model numbers in a program, but it doesn't make sense to create a generic number object. Instead, the Number class makes sense only as a superclass to classes like Integer and Float which implement specific kinds of numbers. Classes such as Number, which implement abstract concepts and should not be instantiated, are called abstract classes. An abstract class is a class that can only be subclassed--it cannot be instantiated.

To declare that your class is an abstract class, use the **keyword abstract** before the class keyword in your class declaration:

```
abstract class Number {
   . . .
}
```

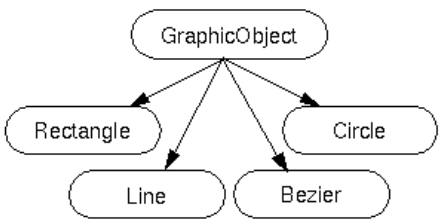
If you attempt to instantiate an abstract class, the compiler will display an error similar to the following and refuse to compile your program:

```
AbstractTest.java:6: class AbstractTest is an abstract class. It can't be instantiated.
      new AbstractTest();
      ^
1 error
```

Abstract Methods

An abstract class may contain abstract methods, that is, methods with no implementation. In this way, an abstract class can define a complete programming interface thereby providing its subclasses with the method declarations for all of the methods necessary to implement that programming interface. However, the abstract class can leave some or all of the implementation details of those methods up to its subclasses.

Let's look at an example of when you might want to create an abstract class with an abstract method in it. In an object-oriented drawing application, you can draw circles, rectangles, lines, Beziers, and so on. Each of these graphic objects share certain states (position, bounding box) and behavior (move, resize, draw). You can take advantage of these similarities and declare them all to inherit from the same parent object--GraphicObject.



However, the graphic objects are also substantially different in many ways: drawing a circle is quite different from drawing a rectangle. The graphics objects cannot share these types of states or

behavior. On the other hand, all GraphicObject's must know how to draw themselves; they just differ in how they are drawn. This is a perfect situation for an abstract superclass.

First you would declare an abstract class, GraphicObject, to provide member variables and methods that were wholly shared by all subclasses such as the current position and the moveTo() method. GraphicObject would also declare abstract methods for methods, such as draw(), that needed to be implemented by all subclasses, but implemented in entirely different ways (no suitable default implementation in the superclass makes sense). The GraphicObject class would look something like this:

```
abstract class GraphicObject {
   int x, y;
   . . .
   void moveTo(int newX, int newY) {
      . . .
   }
   abstract void draw();
}
```

Each non-abstract subclass of GraphicObject, such as Circle and Rectangle, would have to provide an implementation for the draw() method.

```
class Circle extends GraphicObject {
   void draw() {
      . . .
   }
}
class Rectangle extends GraphicObject {
   void draw() {
      . . .
   }
}
```

An abstract class is not required to have an abstract method in it. But any class that has an abstract method in it or that does not provide an implementation for any abstract methods declared in its superclasses must be declared as an abstract class.

17 Write a Java Program to demonstrate the following:

a) **usage of abstract classes**

```
// Abstract class
abstract class Shape {
    // Abstract method
    abstract double area();
    // Concrete method
    void display() {
        System.out.println("This is a shape.");
    } }
// Concrete subclass 1
class Rectangle extends Shape {
    double length;
    double width;
    // Constructor
    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
    // Implementing abstract method
    @Override
    double area() {
        return length * width;
    }
}
// Concrete subclass 2
class Circle extends Shape {
    double radius;
    // Constructor
    Circle(double radius) {
        this.radius = radius;
    }
    // Implementing abstract method
    @Override
    double area() {
```

```
        return Math.PI * radius * radius;
    }
}
public class Main {
    public static void main(String[] args) {
        // Creating objects of concrete subclasses
        Rectangle rectangle = new Rectangle(5, 4);
        Circle circle = new Circle(3);
        // Calling methods
        rectangle.display();
        System.out.println("Area of rectangle: " + rectangle.area());

        circle.display();
        System.out.println("Area of circle: " + circle.area());
    }
}
```

Explanation:

- We define an abstract class Shape with an abstract method area() and a concrete method display().
- We create two concrete subclasses: Rectangle and Circle, each extending the abstract class Shape.
- Each subclass implements the area() method according to its specific shape.
- In the main method, we create objects of the concrete subclasses and demonstrate polymorphism by calling methods on them.
- The display() method is called directly from the abstract class, while the area() method is called through polymorphism.

b) **usage of abstract methods**

```
// Abstract class
abstract class Animal {
    // Abstract method
    abstract void makeSound();
    // Concrete method
    void sleep() {
```

```
        System.out.println("Zzz");
    }}
// Concrete subclass 1
class Dog extends Animal {
    // Implementing abstract method
    @Override
    void makeSound() {
        System.out.println("Woof");
    } }
// Concrete subclass 2
class Cat extends Animal {
    // Implementing abstract method
    @Override
    void makeSound() {
        System.out.println("Meow");
    } }
public class Main {
    public static void main(String[] args) {
        // Creating objects of concrete subclasses
        Dog dog = new Dog();
        Cat cat = new Cat();
        // Calling methods
        dog.makeSound();
        dog.sleep();
        cat.makeSound();
        cat.sleep();
    } }
```

Explanation:

- We define an abstract class Animal with an abstract method makeSound() and a concrete method sleep().
- We create two concrete subclasses: Dog and Cat, each extending the abstract class Animal.
- Each subclass implements the makeSound() method according to the sound it makes.
- In the main method, we create objects of the concrete subclasses and call their methods.

- The makeSound() method is called through polymorphism, where the actual method called depends on the type of the object.

18 Outline the usage in following in Java:

In Java, the final keyword can be used to indicate that something cannot be changed. It can be used in several contexts, such as to declare a variable as a constant, to declare a method as final, or to declare a class as final.

The final Keyword in Java behaves differently when used with classes, methods, and variables.

Points to remember about Final Keyword in Java:

- Once any data member (a variable, method, or class) gets declared as final, it can only be assigned once.
- The final variable cannot be reinitialized with another value.
- A final method cannot be overridden by another method.
- A final class cannot be extended or inherited by another child class.

a) final variables

When a final keyword is used with a variable, it makes the variable constant. Hence, once assigned, the value of the variable cannot be changed.

Example:

```
public class MyClass {
  public static void main(String[] args) {
    final int num = 10;
    System.out.println(num); // prints 10
    // the following line will cause an error because num is final
    num = 20;
  }
}
```

Explanation:

In the above example, num is a final variable that is initialized with the value 10. If you try to change the value of num later in the code, you will get an error.

Correct Way to Implement Final Variable in Java:

```
class Circle {
  public static void main(String[] args) {
    // Constant variable to store the value of pi
    static final double PI = 3.14;
    // Radius of the circle
```

```
    int radius = 4;
    // Calculate and display the area of the circle
    System.out.println("Area of circle: " + (PI * radius * radius));
  }
}
```

Output:

Area of circle: 50.24

b) **final methods**

The final keyword can also be used to declare a method as final. A final method cannot be overridden by a subclass.

Example:

```
public class MyClass {
  public final void myMethod() {
    // method implementation
  }
}
```

In this example, myMethod is a final method that cannot be overridden by a subclass.

```
public class MySubclass extends MyClass {
  // the following line will cause an error because myMethod is a final method
  public void myMethod() {
    // new implementation of myMethod
  }
}
```

The subclass will not be able to override the myMethod method because it is declared as final in the superclass.

Declaring a method as final can be useful if you want to prevent subclasses from changing the behavior of the method. It can also improve the performance of the program because the Java virtual machine (JVM) can optimize the method since it knows that the method cannot be overridden.

Correct Way to Implement Final Methods in Java:

Example:

```
class Animal {
  // Method to display the general characteristics of an animal
  final void displayCharacteristics() {
```

```
    // Constant variables to store the number of legs, ears, eyes, and whether the animal has a tail
    static final int NUM_LEGS = 4;
    static final int NUM_EARS = 2;
    static final int NUM_EYES = 2;
    static final int HAS_TAIL = 1;
    System.out.println("General characteristics of an animal are: ");
    System.out.println("Legs: " + NUM_LEGS);
    System.out.println("Eyes: " + NUM_EYES);
    System.out.println("Ears: " + NUM_EARS);
    System.out.println("Tail: " + (HAS_TAIL == 1 ? "Yes" : "No"));
  }
}
class Wolf extends Animal {
  // Method to display the additional characteristics of a wolf
  final void displayAdditionalCharacteristics() {
    System.out.println();
    System.out.println("Additional characteristics:");
    System.out.println("Sound: howl");
  }
  public static void main(String[] args) {
    Wolf w = new Wolf();
    w.displayCharacteristics();
    w.displayAdditionalCharacteristics();
  }
}
```

Output:

```
General characteristics of an animal are:
Legs: 4
Eyes: 2
Ears: 2
Tail: Yes
Additional characteristics:
Sound: howl
```

c) **final class**

Classes followed by the final keyword in Java cannot be inherited by any subclass (or child class).

Example:

```
public final class MyClass {
  // class implementation
}
```

In this example, MyClass is a final class that cannot be subclassed.

```
public class MySubclass extends MyClass {
  // the following line will cause an error because MyClass is a final class
}
```

The MySubclass class will not be able to extend the MyClass class because it is declared as final. Declaring a class as final can be useful if you want to prevent other classes from creating subclasses of the class.

Correct Way to Implement Final Class in Java:

Example:

```
final class Day {
  // Constant variables to store the number of minutes and hours in a day
  static final int MINUTES_PER_DAY = 1440;
  static final int HOURS_PER_DAY = 24;
  // Method to display the number of hours and minutes in a day
  static void displayHoursAndMinutes() {
    System.out.println("Number of hours in a day: " + HOURS_PER_DAY);
    System.out.println("Number of minutes in a day: " + MINUTES_PER_DAY);
  }
}
class Main {
  // Main method
  public static void main(String[] args) {
    // Call the displayHoursAndMinutes method
    Day.displayHoursAndMinutes();
  }
}
```

Output:

```
Number of hours in a day: 24
Number of minutes in a day: 1440
```

Difference between Final, Finalize, and Finally in Java:

Attributes	Final Keyword	Finally Block	Finalize Method
Definition	The **final keyword** is used as an access modifier to restrict alteration/changes on classes, variables, and methods.	**Finally** is a code block in Exception Handling in Java. This block carries the statements which get executed irrespective of exception occurrence.	**Finalize** is the method in Java that is used to perform memory clean-up before the object got collected by JVM garbage collection.
Applicability	The final keyword is a concept of inheritance and it is used with the classes, methods, and variables.	The finally block comes with a try and catch block in exception handling.	finalize() method is used with the objects to clean the memory space.
Functionality	The final variable cannot be reinitialized with another value. And a final method cannot be overridden by another method. Moreover, a final class cannot be extended or inherited by another child class.	Finally block runs the essential code irrespective of exception occurrence.	finalize method clean the object before its destruction.
Execution	The final method only executes, when called.	Finally block is executed as soon as the try-catch block is executed. Its execution is not dependent on the exception.	finalize method executes just before the object is destroyed by the JVM Garbage collector.

19 Discuss about the implementation of interfaces in Java.

An interface is a reference type, similar to a class that can contain only constants, method signatures, default methods, static methods, and nested types. It represents a contract between the interface and the class that implements it.

An interface is implemented by a class. The class that implements an interface must provide code for all the methods defined in the interface; otherwise, it must be defined as an abstract class.

The class uses a keyword implements to implement an interface. A class can implement any number of interfaces. When a class wants to implement more than one interface, we use the implements keyword is followed by a comma-separated list of the interfaces implemented by the class.

The following is the syntax for defineing a class that implements an interface.

Syntax

```
class className implements InterfaceName{
   ...
   boby-of-the-class
```

```
    ...
}
```

Example:

```
interface Human {
        void learn(String str);
        void work();
        int duration = 10;
}
class Programmer implements Human{
        public void learn(String str) {
                System.out.println("Learn using " + str);
        }
        public void work() {
                System.out.println("Develop applications");
        }
}
public class HumanTest {
        public static void main(String[] args) {
                Programmer trainee = new Programmer();
                trainee.learn("coding");
                trainee.work();
        }
}
```

In the above code defines an interface Human that contains two abstract methods learn(), work() and one constant duration. The class Programmer implements the interface. As it implementing the Human interface it must provide the body of all the methods those defined in the Human interface.

Implementing multiple Interfaces

When a class wants to implement more than one interface, we use the implements keyword is followed by a comma-separated list of the interfaces implemented by the class.

The following is the syntax for defineing a class that implements multiple interfaces.

Syntax

```
class className implements InterfaceName1, InterfaceName2, ...{
```

```
    ...
    boby-of-the-class
    ...
}
```

Extending an Interface

Similar to classes, interfaces can extend other interfaces. The extends keyword is used for extending interfaces. For example,

```
interface Line {
  // members of Line interface
}
// extending interface
interface Polygon extends Line {
  // members of Polygon interface
  // members of Line interface
}
```

20 Explain how interface is used to achieve multiple Inheritances in Java with suitable example.

An interface contains variables and methods like a class but the methods in an interface are abstract by default unlike a class. Multiple inheritance by interface occurs if a class implements multiple interfaces or also if an interface itself extends multiple interfaces.

Example:

```
interface AnimalEat {
  void eat();
}
interface AnimalTravel {
  void travel();
}
class Animal implements AnimalEat, AnimalTravel {
  public void eat() {
    System.out.println("Animal is eating");
  }
  public void travel() {
    System.out.println("Animal is travelling");
  }
```

```
}
public class Demo {
  public static void main(String args[]) {
    Animal a = new Animal();
    a.eat();
    a.travel();
  }
}
```

Output

```
Animal is eating
Animal is travelling
```

The interface AnimalEat and AnimalTravel have one abstract method each i.e. eat() and travel(). The class Animal implements the interfaces AnimalEat and AnimalTravel. A code snippet which demonstrates this is as follows:

```
interface AnimalEat {
  void eat();
}
interface AnimalTravel {
  void travel();
}
class Animal implements AnimalEat, AnimalTravel {
  public void eat() {
    System.out.println("Animal is eating");
  }
  public void travel() {
    System.out.println("Animal is travelling");
  }
}
```

In the method main() in class Demo, an object a of class Animal is created. Then the methods eat() and travel() are called. A code snippet which demonstrates this is as follows:

```
public class Demo {
  public static void main(String args[]) {
    Animal a = new Animal();
```

```
        a.eat();
        a.travel();
    }
}
```

Q21. Demonstrate how to create and access user defined package in Java? Explain with example.

Ans: Following steps are used to create and access user defined package in Java

1. **Create a Package Directory Structure**: In Java, packages are represented by directories. You can create a directory structure that mirrors your package names. For example, if your package name is **com.example**, you would create a directory named **com** and then a subdirectory named **example** inside it.

```
myproject
|
└── src
    └── com
        └── example
```

2. **Write Your Java Classes**: Inside the package directory, you can write your Java classes. Let's create a simple class named **HelloWorld**:

```
package com.example;
public class HelloWorld {
    public static void sayHello() {
        System.out.println("Hello, world!");
    }
}
```

3. **Compile Your Code**: You need to compile your Java code. Ensure that your terminal is at the root of your project directory and execute the following command:

```
Javac  src/com/example/HelloWorld.java
```

This command compiles the **HelloWorld.java** file and generates a **HelloWorld.class** file in the same directory.

4. **Accessing the Package from Another Class**: Now, let's create another class outside the **com.example** package that uses the **HelloWorld** class.

```
// Main.java
import com.example.HelloWorld;
public class Main {
    public static void main(String[] args) {
        HelloWorld.sayHello();
```

```
 }
}
```

5. **Compile and Run the Main Class**: Compile the **Main.java** file using the following command:

```
javac Main.java
```

Then, execute the **Main** class:

This should output: **Hello, world!**.

Q22. Exemplify the ways to access package from another package with an example.

Ans: Let's create a simple example to demonstrate how to access a package from another package in Java.

Suppose we have two packages: **com.example.package1** and **com.example.package2**. We'll create some classes in each package and demonstrate how they can interact with each other.

1. **Create Package Structure**:

```
myproject
|
└── src
    ├── com
    |   └── example
    |       ├── package1
    |       └── package2
    |
    └── Main.java
```

2. **Write Classes in package1**:

```
// MyClass1.java
package com.example.package1;

public class MyClass1 {
    public static void method1() {
        System.out.println("Method 1 from MyClass1 in package1");
    }
}
```

3. **Write Classes in package2**:

```
// MyClass2.java
package com.example.package2;
import com.example.package1.MyClass1;
public class MyClass2 {
    public static void method2() {
        System.out.println("Method 2 from MyClass2 in package2");
        MyClass1.method1(); // Accessing method1 from MyClass1 in package1
    }
}
```

4. Create a Main Class to Run:

```
// Main.java
import com.example.package2.MyClass2;

public class Main {
    public static void main(String[] args) {
        MyClass2.method2(); // Accessing method2 from MyClass2 in package2
    }
}
```

5. **Compile and Run**:

Compile all the **.java** files:

```
javac src/com/example/package1/MyClass1.java
javac src/com/example/package2/MyClass2.java
javac Main.java
```

Run the **Main** class:

```
java Main
```

6. **Output**:

```
Method 2 from MyClass2 in package2
Method 1 from MyClass1 in package1
```

This demonstrates how you can access classes and methods from one package (**package1**) in another package (**package2**). In **MyClass2** within **package2**, we import **MyClass1** from **package1** and then access its **method1**.

23. Illustrate the following access modifiers are used in relation to packages in Java

a)Protected (4M)

b)Public (4M)

Ans: In Java, access modifiers like `protected` and `public` are used to control the accessibility of classes, methods, and fields. Let's illustrate how they work in relation to packages:

a) `protected` Access Modifier:

The `protected` access modifier makes a member (method or field) accessible within its own package and by subclasses in other packages.

```java
// MyClass1.java in package com.example.package1
package com.example.package1;

public class MyClass1 {
    protected void method1() {
        System.out.println("Method 1 from MyClass1 in package1");
    }
}

// MyClass2.java in package com.example.package2
package com.example.package2;
import com.example.package1.MyClass1;
public class MyClass2 extends MyClass1 {
    public void method2() {
        System.out.println("Method 2 from MyClass2 in package2");
        method1();
// Accessing method1 from MyClass1 in package1 because MyClass2 is a subclass of MyClass1
    }
}
```

b) `public` Access Modifier:

The `public` access modifier makes a member accessible from any other class in any package.

// PublicClass.java in package com.example.package1

```
package com.example.package1;
public class PublicClass {
    public void publicMethod() {
        System.out.println("This is a public method in PublicClass");
    }
}

// Main.java in the default package
public class Main {
    public static void main(String[] args) {
        com.example.package1.PublicClass obj = new com.example.package1.PublicClass();
        obj.publicMethod(); // Accessing publicMethod from PublicClass in package1
    }
}
```

In both cases, the `protected` and `public` members can be accessed from classes outside their package, demonstrating the visibility scopes defined by the access modifiers.

24. Implement a Java program to perform employee payroll processing using packages.

Ans: Sure! Let's create a simple Java program to perform employee payroll processing using packages. We'll organize the code into packages to demonstrate how packages can be used for better code organization.

Package Structure:

```
myproject
|
└── src
    ├── com
    |   └── payroll
    |       ├── Employee.java
    |       └── Payroll.java
    |
    └── Main.java
```

Employee Class:

```
// Employee.java in package com.payroll
package com.payroll;
public class Employee {
   private String name;
   private double salary;
   public Employee(String name, double salary) {
      this.name = name;
      this.salary = salary;
   }
   public String getName() {
      return name;
   }

   public double getSalary() {
      return salary;
   }
}
```

Payroll Class:

```
// Payroll.java in package com.payroll
package com.payroll;

public class Payroll {
   public static double calculateSalary(Employee employee) {
      // Simple calculation for demonstration purposes
      double salary = employee.getSalary();
      return salary;
   }
}
```

Main Class:

```
// Main.java in the default package
import com.payroll.Employee;
import com.payroll.Payroll;

public class Main {
    public static void main(String[] args) {
        // Creating Employee objects
        Employee emp1 = new Employee("John Doe", 50000);
        Employee emp2 = new Employee("Jane Smith", 60000);

        // Calculating and displaying salary
        double salary1 = Payroll.calculateSalary(emp1);
        double salary2 = Payroll.calculateSalary(emp2);

        System.out.println(emp1.getName() + "'s salary: $" + salary1);
        System.out.println(emp2.getName() + "'s salary: $" + salary2);
    }
}
```

Explanation:

- We have two classes, `Employee` and `Payroll`, organized into the `com.payroll` package.
- The `Employee` class represents an employee with a name and a salary.
- The `Payroll` class contains a method `calculateSalary` that calculates the salary of an employee.
- In the `Main` class, we import `Employee` and `Payroll` from the `com.payroll` package and create instances of `Employee`.
- We then use the `Payroll` class to calculate and display the salaries of the employees.

Compile and run the `Main` class, and you'll see the output displaying the salaries of the employees based on the calculation performed by the `Payroll` class. This example demonstrates how packages can be used to organize related classes and improve code maintainability.

25. Summarize the following exceptions types in Java.

a) checked exceptions (4M)

b) unchecked exceptions (4M)

Ans: Sure, let's summarize the two main types of exceptions in Java:

a) Checked Exceptions:

Checked exceptions are those exceptions that are checked at compile-time by the compiler. This means that the compiler checks if the code that can potentially throw a checked exception is properly handling it using either a try-catch block or by declaring the exception using the `throws` keyword. If a checked exception is not handled properly, the code will not compile.

Common examples of checked exceptions in Java include:

- `IOException`: This exception is thrown when an I/O operation fails.
- `SQLException`: This exception is thrown when there is an error in executing SQL statements.
- `ClassNotFoundException`: This exception is thrown when a class is not found by the JVM at runtime.

b) Unchecked Exceptions:

Unchecked exceptions, also known as runtime exceptions, are not checked at compile-time by the compiler. These exceptions occur at runtime and are usually caused by programming errors or unexpected conditions in the code. Unlike checked exceptions, there is no requirement to handle or declare unchecked exceptions.

Common examples of unchecked exceptions in Java include:

- `NullPointerException`: This exception is thrown when a null reference is accessed.
- `ArrayIndexOutOfBoundsException`: This exception is thrown when an invalid index is used to access an array.
- `ArithmeticException`: This exception is thrown when an arithmetic operation results in an error, such as division by zero.

In summary, checked exceptions are checked at compile-time and must be handled or declared, while unchecked exceptions occur at runtime and do not require handling or declaration. Both types of exceptions play crucial roles in Java exception handling, helping developers write more robust and reliable code

26. Outline the purpose and functionality of the try, catch, and finally blocks.

Ans: In Java, exception handling is crucial for writing robust and reliable code. The `try`, `catch`, and `finally` blocks are key components of Java's exception handling mechanism.

1. `try` Block:

The `try` block is used to enclose the code that may throw exceptions. It allows you to define a block of code where you anticipate potential exceptions to occur.

```
try {
   // Code that may throw exceptions
} catch (Exception e) {
   // Exception handling code
}
```

- **Purpose**: The primary purpose of the `try` block is to identify code that may throw exceptions.
- **Functionality**: When an exception occurs within the `try` block, the control is transferred to the corresponding `catch` block.

2. `catch` Block:

The `catch` block is used to handle exceptions that are thrown within the corresponding `try` block. It catches the exceptions and provides code to handle them gracefully.

```
try {
   // Code that may throw exceptions
} catch (ExceptionType e) {
   // Exception handling code
}
```

- **Purpose**: The purpose of the `catch` block is to handle specific types of exceptions or to provide a generic exception handling mechanism.

- **Functionality**: When an exception occurs within the `try` block, if the type of the thrown exception matches the type specified in the `catch` block, the control is transferred to that `catch` block for handling the exception.

3. `finally` Block:

The `finally` block is used to define code that needs to be executed regardless of whether an exception occurs or not. It is often used for cleanup operations such as closing resources.

```
try {
   // Code that may throw exceptions
} catch (Exception e) {
   // Exception handling code
} finally {
   // Cleanup or finalization code
}
```

- **Purpose**: The purpose of the `finally` block is to ensure that certain code is executed regardless of whether an exception occurs or not.
- **Functionality**: The `finally` block is executed after the `try` block, and if an exception is caught, after the corresponding `catch` block. It executes even if an exception is thrown or if there is an early return statement in the `try` block.

In summary, `try` blocks identify code that may throw exceptions, `catch` blocks handle those exceptions, and `finally` blocks ensure that certain code is executed regardless of exceptions. Together, they provide a robust mechanism for handling exceptions in Java programs.

27. Elucidate the role of the following keywords in Java exception handling.

a) Throw (4M)

b) Throws (4M)

Ans: Both `throw` and `throws` are keywords in Java used in exception handling, but they serve different purposes. Let's delve into each:

a) `throw` Keyword:

The `throw` keyword is used to explicitly throw an exception within a method or a block of code. When an exceptional situation occurs and the programmer wants to manually trigger an exception, they use the `throw` keyword followed by an instance of the desired exception class.

```
public void divide(int dividend, int divisor) {
   if (divisor == 0) {
      throw new ArithmeticException("Divisor cannot be zero");
   }
   int result = dividend / divisor;
   System.out.println("Result: " + result);
}
```

- **Role**: The role of the `throw` keyword is to throw an exception object explicitly.
- **Functionality**: When the `throw` statement is encountered, the Java runtime system immediately stops executing the current block of code and looks for an appropriate exception handler (a `catch` block) to handle the thrown exception.

b) `throws` Keyword:

The `throws` keyword is used in method declarations to specify that the method may throw certain types of exceptions. When a method is capable of causing an exception that it does not handle itself, it can declare this behavior using the `throws` clause.

```
public void readFile(String fileName) throws FileNotFoundException {
   FileReader fileReader = new FileReader(fileName);
   // Other code to read the file
}
```

- **Role**: The role of the `throws` keyword is to declare that a method might throw certain exceptions.

- **Functionality**: When a method is declared with a `throws` clause, it indicates that the responsibility of handling the specified exceptions lies with the caller of the method. The caller can either handle the exceptions using a `try-catch` block or propagate them further up the call stack.

In summary, `throw` is used to manually throw exceptions within code blocks, while `throws` is used to declare that a method may throw certain types of exceptions, transferring the responsibility of handling those exceptions to the caller of the method.

28. Elucidate the following types of exception handling with example

a) IndexOutof BoundsException (4M)

b) NumberFormatException (4M)

Ans: Let's explore each type of exception handling with examples:

a) IndexOutOfBoundsException:

`IndexOutOfBoundsException` occurs when trying to access an index that is outside the bounds of an array or a collection. This typically happens when accessing an element at an invalid index.

Example:

```
public class IndexOutOfBoundsExceptionExample {
    public static void main(String[] args) {
        int[] numbers = {10, 20, 30};

        try {
            // Accessing an element at an invalid index
            int value = numbers[3]; // This will throw IndexOutOfBoundsException
            System.out.println("Value: " + value);
        } catch (IndexOutOfBoundsException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

Output:

Exception caught: Index 3 out of bounds for length 3

In this example, we try to access the element at index 3 of the `numbers` array, which only has elements at indices 0, 1, and 2. This results in an `IndexOutOfBoundsException`, which is caught in the `catch` block, and an appropriate error message is printed.

b) NumberFormatException:

`NumberFormatException` occurs when trying to convert a string to a numeric type (like `int`, `double`, etc.), but the string does not represent a valid numeric value.

Example:

```
public class NumberFormatExceptionExample {
    public static void main(String[] args) {
        String str = "abc";

        try {
            // Attempting to convert a non-numeric string to an integer
            int number = Integer.parseInt(str); // This will throw NumberFormatException
            System.out.println("Number: " + number);
        } catch (NumberFormatException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

Output:

Exception caught: For input string: "abc"

In this example, we try to parse the string "abc" into an integer using `Integer.parseInt()`, which results in a `NumberFormatException` because "abc" is not a valid integer representation. The exception is caught in the `catch` block, and an appropriate error message is printed.

These examples illustrate how to handle `IndexOutOfBoundsException` and `NumberFormatException` using `try-catch` blocks, enabling graceful handling of exceptional conditions in Java programs.

29. Explain how user-defined exceptions are implemented in Java.

Ans: User-defined exceptions in Java are custom exceptions created by the programmer to represent specific error conditions that may arise in their application. These exceptions extend either directly from the `Exception` class or one of its subclasses. By creating custom exceptions, developers can provide more meaningful error messages and handle exceptional situations more effectively.

Here's a general outline of how user-defined exceptions are implemented in Java:

1. **Create a Custom Exception Class**:

Define a new class that extends either `Exception` or one of its subclasses (such as `RuntimeException` or any other subclass of `Exception`). This class represents the custom exception.

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

2. **Throwing the Custom Exception**:

Use the `throw` keyword to throw an instance of the custom exception class whenever the exceptional condition occurs in your code.

```
public class MyClass {
    public void performOperation() throws CustomException {
        // Check for some condition
        if (/* condition */) {
            throw new CustomException("Custom error message");
        }
        // Other code
```

```
    }
  }
```

3. **Handling the Custom Exception**:

In the calling code, catch the custom exception using a `try-catch` block and handle it appropriately.

```
  public class Main {
    public static void main(String[] args) {
      MyClass obj = new MyClass();
      try {
        obj.performOperation();
      } catch (CustomException e) {
        // Handle the custom exception
        System.out.println("Custom exception caught: " + e.getMessage());
      }
    }
  }
```

By following these steps, you can implement user-defined exceptions in Java. This approach allows you to create custom exceptions tailored to the specific needs of your application, providing clearer error messages and more precise error handling.

30. Elaborate the best practices for exception handling in Java.

Ans: Exception handling is a critical aspect of writing robust and reliable Java code. Here are some best practices to follow for effective exception handling in Java:

1. **Use Specific Exceptions**: Catch specific exceptions rather than catching general exceptions like `Exception` or `Throwable`. This allows you to handle different types of exceptions differently and provides better clarity in your code.

2. **Handle Exceptions Appropriately**: Handle exceptions at an appropriate level of abstraction. Don't catch exceptions if you can't handle them properly. Instead, let them propagate to higher levels where they can be handled effectively.

3. **Avoid Catching Throwable**: Avoid catching `Throwable` unless absolutely necessary. Catching `Throwable` can mask errors like `OutOfMemoryError` or other critical errors.

4. **Use Try-With-Resources**: When working with resources like streams, connections, or files, use try-with-resources (introduced in Java 7). This ensures that resources are closed properly, even if an exception occurs.

```
try (FileInputStream fis = new FileInputStream("file.txt")) {
    // Code that uses fis
} catch (IOException e) {
    // Exception handling
}
```

5. **Log Exceptions**: Always log exceptions and error messages using a logging framework like Log4j or java.util.logging. Logging exceptions helps in debugging and troubleshooting issues in production environments.

6. **Provide Descriptive Error Messages**: When throwing exceptions or logging them, provide descriptive error messages that explain the cause of the exception and guide developers or users on how to resolve the issue.

7. **Avoid Swallowing Exceptions**: Avoid swallowing exceptions by catching them and not doing anything with them. If you catch an exception and can't handle it, consider logging it or rethrowing it.

8. **Follow Java Naming Conventions**: Name custom exception classes appropriately and in line with Java naming conventions. For example, suffix custom exception classes with "Exception".

9. **Use Checked Exceptions Judiciously**: Use checked exceptions for exceptional conditions that can be reasonably anticipated and recovered from. If an exception is not recoverable or is unlikely to occur, use unchecked exceptions.

10. **Test Exception Handling Code**: Test exception handling code thoroughly to ensure that exceptions are caught and handled correctly in different scenarios.

11. **Document Exception Handling**: Document the exceptions thrown by methods using Javadoc comments. Describe the conditions under which each exception is thrown and provide guidance on handling them.

By following these best practices, you can write cleaner, more robust, and maintainable Java code with effective exception handling.

Q31. Illustrate the following ways of creating thread with an example.
(I)Using Thread class (4M)
(II)Using Runnable interface (4M).
Using Thread class:

Ans: The first way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run( ) method, which is the entry point for the new thread. As before, a call to start( ) begins execution of the new thread. Here is the preceding program rewritten to extend Thread:

```
// Create a second thread by extending Thread
class NewThread extends Thread {
NewThread() {
// Create a new, second thread
super("Demo Thread");
System.out.println("Child thread: " + this);
}
// This is the entry point for the second thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
class ExtendThread {
public static void main(String[] args) {
NewThread nt = new NewThread(); // create a new thread
nt.start(); // start the thread
try {
for(int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

This program generates the same output as the preceding version. As you can see, the child thread is created by instantiating an object of NewThread, which is derived from Thread.

Using Runnable interface:

Ans: The easiest way to create a thread is to create a class that implements the Runnable interface. Runnable abstracts a unit of executable code. You can construct a thread on any object that implements Runnable. To implement Runnable, a class need only implement a single method called run( ), which is declared like this:
public void run( )
Inside run( ), you will define the code that constitutes the new thread.

After the new thread is created, it will not start running until you call its start( ) method, which is declared within Thread. In essence, start( ) initiates a call to run( ). The start( ) method is shown here:
void start( )
Here is an example that creates a new thread and starts it running:

```
// Create a second thread.
class NewThread implements Runnable {
Thread t;
NewThread() {
// Create a new, second thread
t = new Thread(this, "Demo Thread");
System.out.println("Child thread: " + t);
}
// This is the entry point for the second thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}

class ThreadDemo {
public static void main(String[] args) {
NewThread nt = new NewThread(); // create a new thread
nt.t.start(); // Start the thread
try {
for(int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

Inside NewThread’s constructor, a new Thread object is created by the following

statement:
t = new Thread(this, "Demo Thread");

Q32. What is a thread? Explain the states of a thread with a neat diagram. (Thread Life Cycle)

Ans:

In computing, a thread refers to the smallest unit of execution within a process. Threads share the same memory space and resources of the process they belong to, enabling them to work concurrently.

The life cycle of a thread generally consists of several states through which it transitions during its execution. Here's a simplified explanation along with a diagram:

1. **New:** This is the initial state of a thread when it's created but not yet started. Resources like memory have been allocated, but the thread hasn't started executing yet.

2. **Runnable/Ready:** Once the thread's start method is called, it moves to the runnable state. In this state, the thread is eligible to run, but the scheduler has not yet selected it to run on the CPU.

3. **Running:** When the scheduler selects the thread for execution, it moves into the running state. In this state, the thread is actively executing its code on the CPU.

4. **Blocked/Waiting:** A thread can enter this state for various reasons, such as waiting for I/O operations to complete, acquiring a lock, or waiting for another thread to finish. While in this state, the thread cannot continue execution until the condition causing the blockage is resolved.

5. **Terminated/Dead:** This is the final state of a thread. It occurs when the thread completes its task or is explicitly terminated. In this state, the thread releases any resources it holds and its memory space is deallocated.

Here's a basic diagram illustrating the thread life cycle:

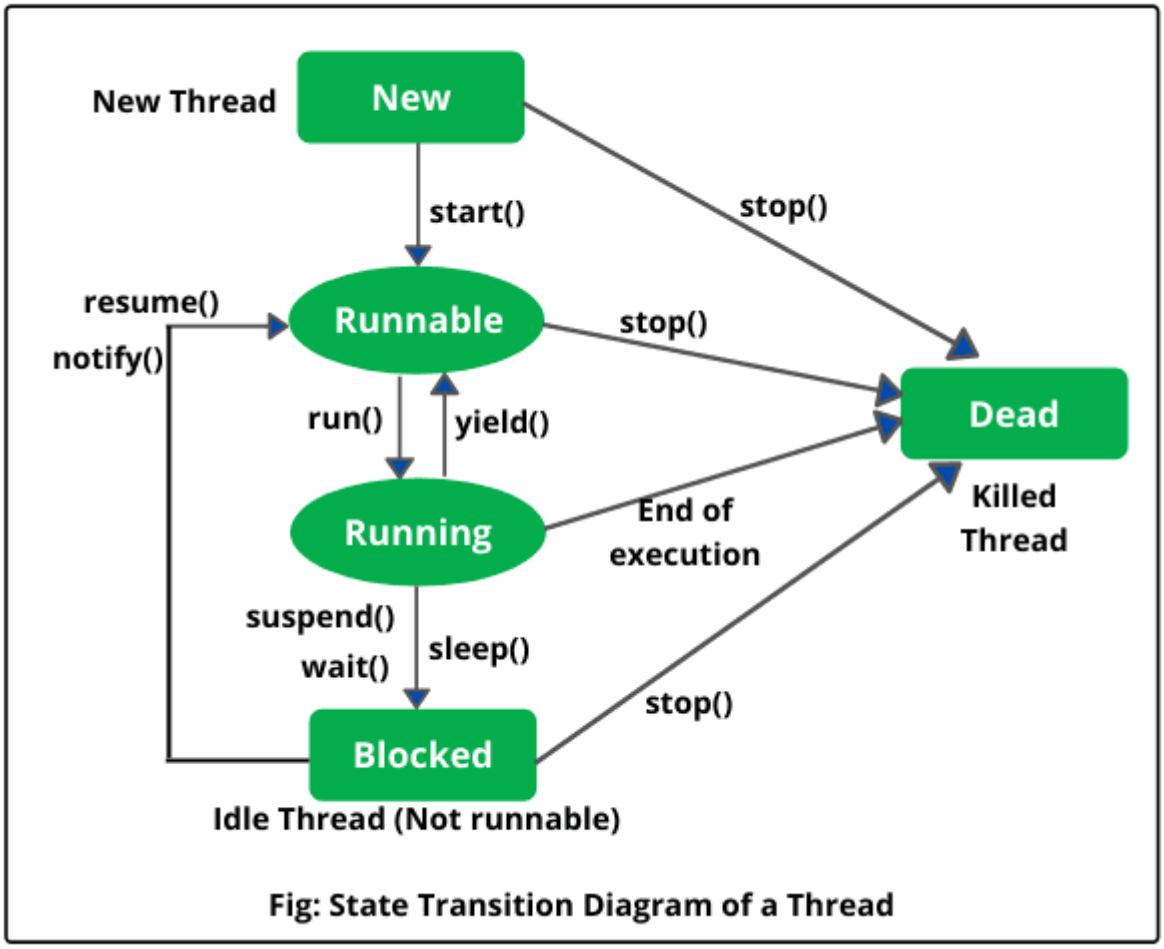


Fig: State Transition Diagram of a Thread

Q33. Develop a java program for producer and consumer problem using Threads

Ans:

The producer-consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem.

The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

The producer's job is to generate data, put it into the buffer, and start again.

At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

Problem

To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Solution

The producer is to either go to sleep or discard data if the buffer is full.

The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again.

In the same way, the consumer can go to sleep if it finds the buffer to be empty.

The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

Program:

```
import java.util.LinkedList;
public class ThreadExample {
    public static void main(String[] args) throws InterruptedException {
        final PC pc = new PC();

        Thread t1 = new Thread(() -> {
            try {
                pc.produce();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        Thread t2 = new Thread(() -> {
            try {
                pc.consume();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        t1.start();
        t2.start();
        t1.join();
        t2.join();
    }
    public static class PC {
        LinkedList<Integer> list = new LinkedList<>();
        int capacity = 2;
        public void produce() throws InterruptedException {
            int value = 0;
```

```
        while (true) {
            synchronized (this) {
                while (list.size() == capacity)
                    wait();

                System.out.println("Producer produced: " + value);
                list.add(value++);
                notify();
                Thread.sleep(1000);
            }
        }
    }
    public void consume() throws InterruptedException {
        while (true) {
            synchronized (this) {
                while (list.size() == 0)
                    wait();

                int val = list.removeFirst();
                System.out.println("Consumer consumed: " + val);
                notify();
                Thread.sleep(1000);
            }
        }
    }
}
```

Output:

Producer produced: 0

Producer produced: 1

Consumer consumed: 0

Producer produced: 2

Consumer consumed: 1

Producer produced: 3

Consumer consumed: 2

Producer produced: 4

Consumer consumed: 3

Producer produced: 5

Consumer consumed: 4

Q34. Compare and contrast between multi-processing and multi-threading.

Ans:

Multi-processing and multi-threading are both techniques used in concurrent programming to achieve parallelism and improve performance. However, they have distinct differences in their approach and usage:

1. **Definition**:
   - Multi-processing: Involves executing multiple processes simultaneously where each process has its own memory space, resources, and address space. Processes are independent of each other and communicate through inter-process communication mechanisms.
   - Multi-threading: Involves executing multiple threads within a single process. Threads share the same memory space and resources of the process they belong to, allowing them to share data and communicate more efficiently.

2. **Resource Overhead**:
   - Multi-processing: Generally has a higher overhead in terms of system resources (such as memory and CPU) because each process requires its own memory space and resources.
   - Multi-threading: Has lower overhead compared to multi-processing because threads within the same process share resources and memory.

3. **Communication**:
   - Multi-processing: Inter-process communication (IPC) mechanisms such as pipes, message queues, or shared memory are used for communication between processes.
   - Multi-threading: Threads within the same process can communicate directly by sharing variables and data structures, simplifying communication.

4. **Scalability**:
   - Multi-processing: Offers better scalability, especially on multi-core or multi-processor systems, as different processes can run on different CPU cores.

- Multi-threading: May have limitations in scalability due to factors like contention for shared resources within the process, but it's generally more lightweight and efficient for applications that require concurrent execution within a single process.

5. **Fault Isolation**:
- Multi-processing: Provides better fault isolation since each process runs in its own memory space. If one process crashes, it typically doesn't affect other processes.
- Multi-threading: May lack fault isolation since all threads within the same process share the same memory space. A bug or crash in one thread can potentially affect the entire process.

6. **Programming Complexity**:
- Multi-processing: Can be more complex to implement due to the need for explicit communication mechanisms between processes and potential synchronization issues.
- Multi-threading: Generally simpler to implement as threads share the same memory space and can communicate more easily, but it requires careful synchronization to avoid race conditions and other concurrency issues.

Q35. Write a Java program that implements a multi-thread application that has three threads.

Ans:

```
public class MultiThreadExample {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new MyRunnable("Thread 1"));
        Thread thread2 = new Thread(new MyRunnable("Thread 2"));
        Thread thread3 = new Thread(new MyRunnable("Thread 3"));

        thread1.start();
        thread2.start();
        thread3.start();
    }

    static class MyRunnable implements Runnable {
        private final String threadName;
```

```
    MyRunnable(String threadName) {
        this.threadName = threadName;
    }

    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(threadName + " - Iteration: " + i);
            try {
                // Sleep for a random duration to simulate some work
                Thread.sleep((long) (Math.random() * 1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println(threadName + " finished execution.");
    }
  }
}
```

This program creates three threads using the Thread class and assigns each thread a MyRunnable object as the target runnable. The MyRunnable class implements the Runnable interface, defining the task each thread will execute. In this case, it prints a message indicating the thread's name and iteration count.

Output:

```
Thread 1 - Iteration: 1
Thread 3 - Iteration: 1
Thread 2 - Iteration: 1
Thread 2 - Iteration: 2
Thread 1 - Iteration: 2
Thread 3 - Iteration: 2
Thread 1 - Iteration: 3
Thread 2 - Iteration: 3
```

Thread 3 - Iteration: 3

Thread 2 - Iteration: 4

Thread 3 - Iteration: 4

Thread 1 - Iteration: 4

Thread 2 - Iteration: 5

Thread 1 - Iteration: 5

Thread 3 - Iteration: 5

Thread 2 finished execution.

Thread 1 finished execution.

Thread 3 finished execution.

Q36. Explore different states in the life cycle of a Java applet.

Ans:

In Java, an applet is a special type of program that runs within a web browser. The life cycle of a Java applet involves several states as it is loaded, initialized, displayed, interacted with, and eventually destroyed. Here are the different states in the life cycle of a Java applet:

Initialization:

Loaded: The applet's class files are loaded into memory when the web page containing the applet is accessed.

Instantiation: An instance of the applet class is created using its constructor.

Start:

Init(): The init() method is called after the applet is instantiated. This method is used for one-time initialization tasks such as setting up GUI components, initializing variables, and loading resources.

Start(): The start() method is called after init() and signals that the applet is starting or restarting. This method is used to start any threads or animations that the applet may use.

Running:

Running State: The applet is in a running state when it is visible in the web browser and actively responding to user input or performing tasks.

User Interaction:

Mouse/Keyboard Events: During the running state, the applet can respond to user input events such as mouse clicks, keyboard presses, and mouse movements.

Pause/Stop:

Stop(): The stop() method is called when the applet is no longer visible in the web browser, such as when the user navigates away from the page containing the applet. This method is used to suspend any ongoing activities or threads.

LostFocus(): If the applet loses focus (e.g., the user switches to another tab or window), the lostFocus() method may be called, allowing the applet to pause or perform other actions.

Shutdown:

Destroy(): The destroy() method is called when the applet is about to be unloaded from memory. This method is used to perform cleanup tasks such as releasing resources, closing streams, and stopping any background threads.

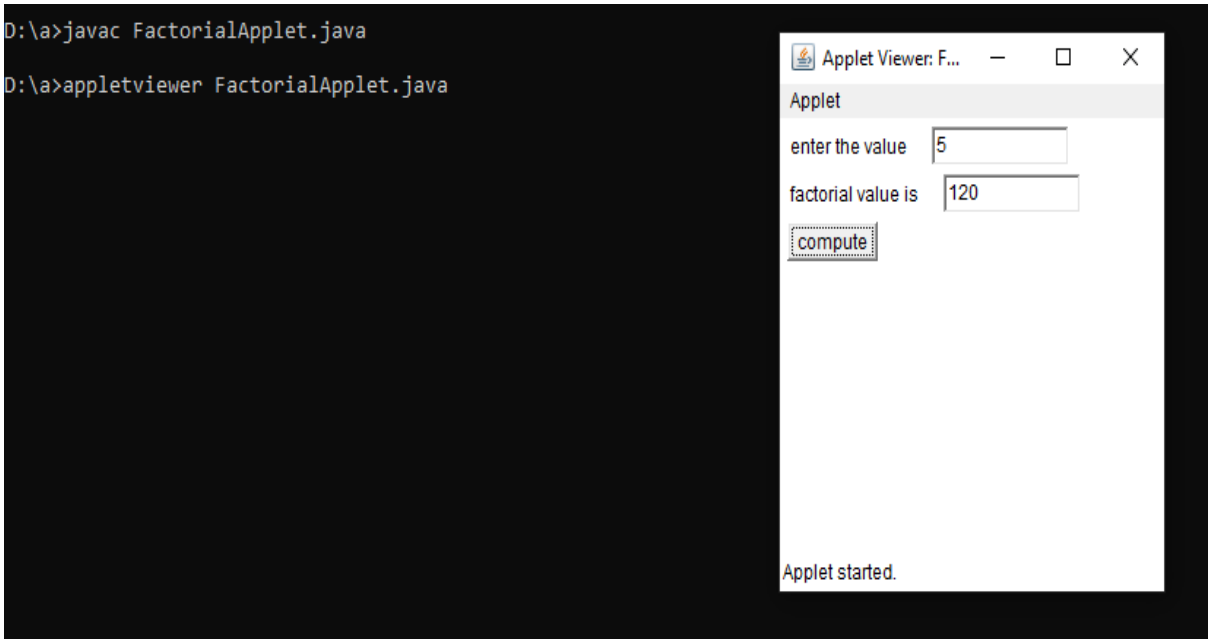
Q27. Implement a Java program compute factorial value using Applet

Ans:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="FactorialApplet" width=300 height=300>
</applet>
*/
public class FactorialApplet extends Applet implements ActionListener
{
      Label L1,L2;
      TextField T1,T2;
       Button B1;
        public void init()
       {
setLayout(new FlowLayout(FlowLayout.LEFT));
L1=new Label("enter the value");
add(L1);
```

```
T1=new TextField(10);
add(T1);
L2=new Label("factorial value is");
add(L2);
T2=new TextField(10);
add(T2);
B1=new Button("compute");
add(B1);
B1.addActionListener(this);
        }


public void actionPerformed(ActionEvent e)
{
if(e.getSource()==B1)
{
int value=Integer.parseInt(T1.getText());
int fact=factorial(value);
T2.setText(String.valueOf(fact));
}
}
 int factorial(int n)
 {
if(n==0)
return 1;
else
return n*factorial(n-1);
 }
}
```

Output:



Q38. Examine Java applets methods with example code.

Ans: Java applets are special Java programs that are designed to be embedded within HTML pages and run in web browsers. They have a specific lifecycle defined by methods that are called at different stages of their execution. Here are some of the key methods used in Java applets along with example code:

init(): This method is called when the applet is first initialized. It is typically used to perform one-time initialization tasks such as setting up GUI components and initializing variables.

```
import java.applet.Applet;
import java.awt.Label;

public class InitExample extends Applet {
    public void init() {
        Label label = new Label("Hello, World!");
        add(label);
    }
}
```

start(): This method is called when the applet is started or restarted. It is used to start any threads or animations that the applet may use.

```
import java.applet.Applet;
```

```
public class StartExample extends Applet {
    public void start() {
        // Start animation or other activities
    }
}
```

stop(): This method is called when the applet is stopped. It is typically used to suspend any ongoing activities or threads.

```
import java.applet.Applet;

public class StopExample extends Applet {
    public void stop() {
        // Suspend ongoing activities or threads
    }
}
```

destroy(): This method is called when the applet is about to be unloaded from memory. It is used to perform cleanup tasks such as releasing resources and stopping any background threads.

```
import java.applet.Applet;

public class DestroyExample extends Applet {
    public void destroy() {
        // Perform cleanup tasks
    }
}
```

paint(Graphics g): This method is called whenever the applet needs to be rendered or repainted. It is typically used to draw graphics and display visual elements on the applet's surface.

```
import java.applet.Applet;
import java.awt.Graphics;
public class PaintExample extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello, World!", 50, 50);
```

```
    }
}
```

These are some of the most commonly used methods in Java applets.

Q39. What are the advantages and disadvantages of using Java applets?

Ans:

Advantages:

Rich User Interfaces: Java applets can provide rich and interactive user interfaces with features like animations, multimedia, and advanced graphical capabilities using the Java AWT (Abstract Window Toolkit) or Swing APIs.

Cross-platform Compatibility: Java applets are platform-independent, meaning they can run on any system with a Java Virtual Machine (JVM) installed. This cross-platform compatibility ensures that applets behave consistently across different operating systems and web browsers.

Security: Applets run within a sandboxed environment provided by the JVM, which restricts their access to system resources. This security model helps prevent malicious applets from causing harm to the user's system.

Dynamic Content: Applets can dynamically update and modify their content without requiring the entire web page to be reloaded, providing a more seamless and responsive user experience.

Wide Range of APIs: Java provides a vast array of APIs for applet development, including networking, database access, multimedia, and more, allowing developers to create complex and feature-rich applications.

Disadvantages:

Browser Compatibility Issues: Java applets rely on browser support for the Java plugin, which has become increasingly limited in modern web browsers due to security concerns and performance issues. Many browsers have deprecated or completely removed support for Java applets.

Performance: Java applets can suffer from performance issues, especially when running complex animations or graphical effects. The overhead of running within the JVM and the need for browser plugins can lead to slower performance compared to other web technologies.

Security Concerns: While Java's sandboxed environment provides some level of security, it is not foolproof. There have been numerous security vulnerabilities in the Java plugin over the years, leading to concerns about the potential for malicious applets to exploit these vulnerabilities and compromise users' systems.

Limited Mobile Support: Java applets are not supported on mobile devices, as most mobile browsers do not have a Java plugin. With the shift towards mobile-first web development, this limitation further diminishes the relevance of Java applets in modern web development.

Complex Deployment: Deploying Java applets on web pages requires users to have the Java plugin installed and enabled in their browsers. This additional step can be cumbersome for both developers and end-users, especially considering the declining popularity of Java applets.

Q30. Write a Java program that demonstrates painting an Applet.

Ans:

```
import java.awt.event.*;

import java.applet.*;

/*

<applet code="Painting.class" width="300" height="300">

</applet>

*/

public class Painting extends Applet implements MouseMotionListener

{

   public void init ()

   {
```

```
        addMouseMotionListener (this);
        setBackground (Color.red);
    }
    public void mouseDragged (MouseEvent me)
    {
        Graphics g = getGraphics ();
        g.setColor (Color.white);
        g.fillOval (me.getX (), me.getY (), 5, 5);
    }
    public void mouseMoved (MouseEvent me)
    {
    }
}
```

Output:

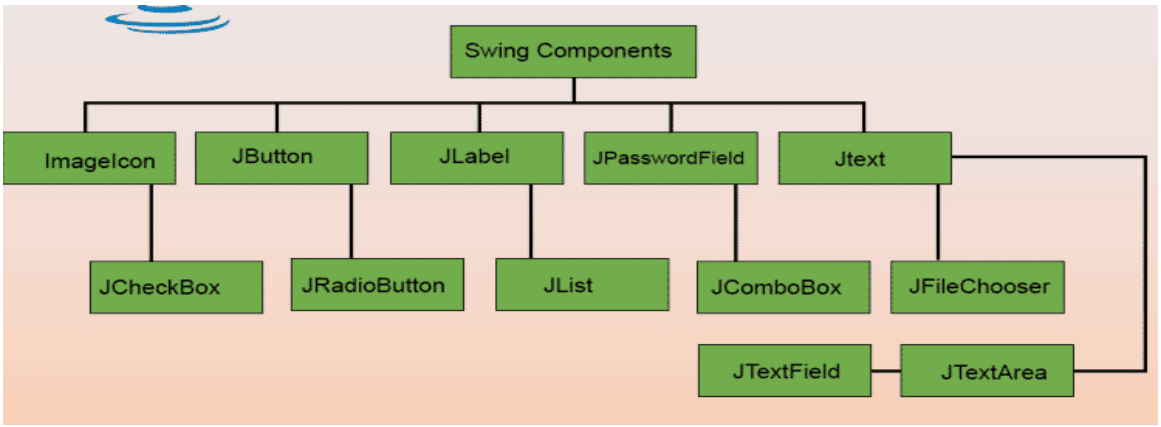
UNIT-V
Java Swing, Servlets & JDBC

Question & Answers

1) Illustrate all the key components of Java Swing and their functionalities?

Answer:

- Swing is the graphical user interface toolkit that is used for developing windows based java applications or programs. It is the successor of AWT, known as the Abstract window toolkit API for Java, and AWT components are mainly heavyweight.
- A component is an independent visual control and Java Swing Framework contains a large set of these components which provide rich functionalities and allow high level of customization. They all are derived from JComponent class.
- This class provides some common functionality like pluggable look and feel, support for accessibility, drag and drop, layout, etc.
- Swing has a big set of components that we can include in our programs and avail the rich functionalities using which we can develop highly customized and efficient GUI applications.
- **JComponent:** With the exception of top-level containers, all Swing components whose names begin with "J" descend from the JComponent class.
- For example, JPanel, JScrollPane, JButton, and JTable all inherit from JComponent. The JComponent class extends the Container class, which itself extends Component. The Component class includes everything from providing layout hints to supporting painting and events. The Container class has support for adding components to the container and laying them out.



Icons and Labels:

- Swing allows you to create labels that can contain text, images, or both.
- JLabel is a class of java Swing. JLabel is used to display a short string or an image icon.

- JLabel is inactive to input events such a mouse focus or keyboard focus. By default labels are vertically centred but the user can change the alignment of label.
  Example:
  ImageIcon icon = new ImageIcon("java-swing-tutorial.JPG","My Website");
  JLabel j1 = new JLabel("Image with Text", icon, JLabel.CENTER);
  JLabel j2 = new JLabel("Text Only Label");
  JLabel j3 = new JLabel(icon);

JTextfields:

- JTextField is a part of javax.swing package. The class JTextField is a component that allows editing of a single line of text. JTextField inherits the JTextComponent class and uses the interface SwingConstants.
- A user can input non-formatted text in the box. To initialize the text field, call its constructor and pass an optional integer parameter to it. This parameter sets the width of the box measured by the number of columns. It does not limit the number of characters that can be input in the box.
  Example:
  JTextField txtBox = new JTextField(16); It renders a text box of 16 column width.

JButton:

- JButton class in Java is used to create push buttons that can be used to perform any ActionEvent whenever it is clicked.
- In Order to achieve event action, the ActionListener interface needs to be implemented.
- The Buttons component in Swing is similar to that of the AWT button component except that it can contain text, image or both.
- The JButton class
- extends the JComponent class and can be used in a container.

Examples: JButton okBtn = new JButton(“Click Here");
This constructor returns a button with text Click Here on it

Check boxes:

- Check boxes are labelled options that allow the user to select multiple options at once. Thus, any, all, or none of a set of check boxes may be selected.
- JCheckBox renders a check-box with a label. The check-box has two states – on/off. When selected, the state is on and a small tick is displayed in the box.
  Example: CheckBox chkBox = new JCheckBox("Show Title", true);
  It returns a checkbox with the label Show Help. Notice the second parameter in the constructor.
  It is a boolean value that indicates the default state of the check-box. True means the check-box is defaulted to on state.

Radio buttons:

- Radio buttons provide a user with a way to select one of a number of labelled options displayed simultaneously in a window.
- Radio buttons are organized in a radio button group. Only one button in a group can be selected at any given time.
- When the user selects a new button, the previously selected button is automatically

deselected. One button is usually selected at program starup by default.

Example:

```
ButtonGroup radioGroup = new ButtonGroup();
JRadioButton rb1 = new JRadioButton("Easy", true);
JRadioButton rb2 = new JRadioButton("Medium");
JRadioButton rb3 = new JRadioButton("Hard"); r
adioGroup.add(rb1);
radioGroup.add(rb2);
radioGroup.add(rb3);
```

Combo boxes:

- Combo boxes are drop-down menus that can appear along with other widgets in the main area of a window. The user may select one option at a time, and it then remains selected.
- JCombobox class is used to render a dropdown of the list of options.

  Example:

  ```
  String[] proglang = { “Java", "Python", "c++", "PHP", "Perl" };
  JComboBox lang = new JComboBox(proglang);
  lang.setSelectedIndex(3);
  ```

  The default selected option can be specified through the setSelectedIndex method. The above code sets C++ as the default selected option.

Tabbed Panes:

- With the JTabbedPane class, you can have several components, such as panels, share the same space. The user chooses which component to view by selecting the tab corresponding to the desired component.
- TtabbedPane is very useful component that lets the user switch between tabs in an application. This is a highly useful utility as it lets the user browse more content without navigating to different pages.
- To create a tabbed pane, instantiate JTabbedPane, create the components you wish it to display, and then add the components to the tabbed pane using the addTab method.

  Example:

  ```
  JTabbedPane tabbedPane = new JTabbedPane();
  tabbedPane.addTab("Tab 1", new JPanel());
  tabbedPane.addTab("Tab 2", new JPanel());
  tabbedPane.addTab("Tab 3", new JPanel());
  tabbedPane.addTab("Tab 4", new JPanel());
  ```

Scroll Panes:

- A JscrollPane is used to make scrollable view of a component. When screen size is limited, we use a scroll pane to display a large component or a component whose size can change dynamically.
- It can have both vertical and horizontal scrollbars when needed.

  Example:

  ```
  JTextArea tArea = new JTextArea(10,10);
  JScrollPane scrollPane = new JScrollPane(tArea);
  panel.add(scrollPane);
  ```

Trees:

- JTree is a Swing component with which we can display hierarchical data. JTree is quite a complex component.
- A JTree has a 'root node' which is the top-most parent for all nodes in the tree. A node is an item in a tree.
- A node can have many children nodes. These children nodes themselves can have further children nodes. If a node doesn't have any children node, it is called a leaf node.
  Example:
  DefaultMutableTreeNode state=new DefaultMutableTreeNode("States");
  DefaultMutableTreeNode wb=new DefaultMutableTreeNode("West Bengal");
  DefaultMutableTreeNode del=new DefaultMutableTreeNode("Delhi");
  DefaultMutableTreeNode ap=new DefaultMutableTreeNode("Andhra Pradesh");
  DefaultMutableTreeNode tn=new DefaultMutableTreeNode("Tamil Nadu");
  state.add(wb); state.add(del); state.add(ap); state.add(tn);
  JTree jt=new JTree(state);
  f.add(jt);

Tables:

- The JTable class is a part of Java Swing Package and is generally used to display or edit two-dimensional data that is having both rows and columns.
- It is similar to a spreadsheet. This arranges data in a tabular form.
  Example:
  String[] columnNames = {"Name", "Salary"};
  Object[][] data = { {"Ramesh Raman", 5000} {"Shabbir Hussein", 7000} };
  JTable table = new JTable(data, columnNames);
  JScrollPane scrollPane = new JScrollPane(table);
  scrollPane.setSize(300, 300);
  table.setFillsViewportHeight(true);

2) **Compare and contrast between AWT and Swing in Java**

Answer:

S.NO	AWT	Swing
1.	Java AWT is an API to develop GUI applications in Java	Swing is a part of Java Foundation Classes and is used to create various applications.
2.	The components of Java AWT are heavy weighted.	The components of Java Swing are light weighted.
3.	Java AWT has comparatively less functionality as compared to Swing.	Java Swing has more functionality as compared to AWT.

S.NO	AWT	Swing
4.	The execution time of AWT is more than Swing.	The execution time of Swing is less than AWT.
5.	The components of Java AWT are platform dependent.	The components of Java Swing are platform independent.
6.	MVC pattern is not supported by AWT.	MVC pattern is supported by Swing.
7.	AWT provides comparatively less powerful components.	Swing provides more powerful components.
8	AWT components require java.awt package	Swing components requires javax.swing package
9	AWT is a thin layer of code on top of the operating system.	Swing is much larger swing also has very much richer functionality.
10	AWT stands for Abstract windows toolkit .	Swing is also called as JFC(java Foundation classes). It is part of oracle's JFC.
11	Using AWT , you have to implement a lot of things yourself .	Swing has them built in.

3) Implement a Java Program to display text in different fonts

Answer:

```
import javax.swing.*;
import java.awt.*;

public class DifferentFontsExample extends JFrame {
    public DifferentFontsExample() {
        setTitle("Different Fonts Example");
        setSize(600, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        add(new FontPanel());
    }
```

```
    public static void main(String[] args) {
            SwingUtilities.invokeLater(() -> {
                DifferentFontsExample frame = new DifferentFontsExample();
                frame.setVisible(true);
            });
        }
    }

    class FontPanel extends JPanel {
        @Override
        protected void paintComponent(Graphics g) {
            super.paintComponent(g);

            String text = "Hello, Java!";
            int x = 50;
            int y = 50;

            // Define different fonts
            Font font1 = new Font("Serif", Font.PLAIN, 24);
            Font font2 = new Font("SansSerif", Font.BOLD, 24);
            Font font3 = new Font("Monospaced", Font.ITALIC, 24);
            Font font4 = new Font("Dialog", Font.BOLD | Font.ITALIC, 24);

            // Set and draw text with different fonts
            g.setFont(font1);
            g.drawString(text, x, y);

            y += 50; // Move down for the next text
            g.setFont(font2);
            g.drawString(text, x, y);

            y += 50; // Move down for the next text
            g.setFont(font3);
            g.drawString(text, x, y);

            y += 50; // Move down for the next text
            g.setFont(font4);
            g.drawString(text, x, y);
        }
    }
```

4) Illustrate the following types of Containers in Java AWT

- Frame (4M)
- Panel (4M)

Answer:

Frame in Java AWT

A Frame in Java AWT is a top-level window with a title and a border that can act as the main window for an application. Frames are capable of containing other AWT components such as buttons, text fields, panels, and more. They are typically used to create the primary user interface for a standalone application. A Frame is a top-level window with a title and a border. It's like the main window of an application where you can add other components like buttons, labels, text fields, etc.

Example of a Frame:

```
import java.awt.*;
public class FrameExample {
    public static void main(String[] args) {
        // Create a new Frame
        Frame frame = new Frame("AWT Frame Example");
        // Set the size of the frame
        frame.setSize(400, 300);
        // Set the layout for the frame
        frame.setLayout(new FlowLayout());
        // Add a button to the frame
        Button button = new Button("Click Me");
        frame.add(button);
        // Add a label to the frame
        Label label = new Label("Hello, AWT!");
        frame.add(label);
        // Make the frame visible
        frame.setVisible(true);
        // Add a window listener to close the frame
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                frame.dispose();
            }
        });
    }
}
```

Panel in Java AWT:

A Panel in Java AWT is a container that can hold a group of components. Panels are often used as sub-containers to organize components within a larger container, such as a Frame or another Panel. They do not have borders or titles and are used primarily to manage layout and group related

components. A Panel is a container that can hold a group of components. It's used to organize components within a window (like a Frame).

Example of a Panel:

```
import java.awt.*;
public class PanelExample {
        public static void main(String[] args) {
            // Create a new Frame
            Frame frame = new Frame("AWT Panel Example");
            // Create a Panel
            Panel panel = new Panel();
            // Set the size of the frame
            frame.setSize(400, 300);
            // Set the layout for the panel
            panel.setLayout(new FlowLayout());
            // Add components to the panel
            panel.add(new Button("Button 1"));
            panel.add(new Button("Button 2"));
            panel.add(new Label("Label inside Panel"));
            // Add the panel to the frame
            frame.add(panel);
            // Make the frame visible
            frame.setVisible(true);
            // Add a window listener to close the frame
            frame.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                    frame.dispose();
                }
            });
        }
    }
```

5) Analyse the states in the life cycle of a Servlet.

Answer:

The life cycle is the process from the construction till the destruction of any object. A servlet alsofollows a certain life cycle. The life cycle of the servlet is managed by the servlet container.

The container performs the following steps:

1. **Loads Servlet Class**
2. **Creates instance of Servlet**
3. **Calls init( ) method**
4. **Calls service ( ) method**
5. **Calls destroy ( ) method**

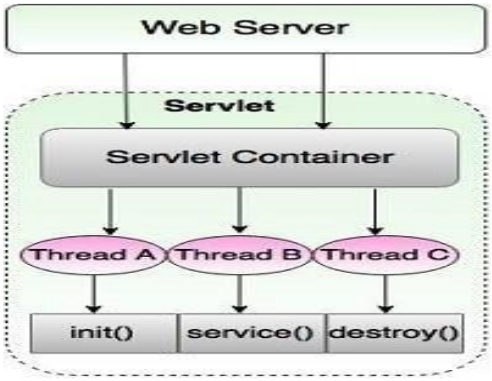


Fig: Servlet Life Cycle

1. Loads the Servlet Class

The **class loader** is responsible to load the Servlet class into the container. The servlet class isloaded when the first request comes from the web container.

2. Creates instance of Servlet

After loading of Servlet class into the container, the web container creates its instance. Theinstance of Servlet class is created only once in the servlet life cycle.

3. Calls the init( ) method

The **init( )** method performs some specific action constructor of a class. It is automatically calledby Servlet container. It is called only once through the complete servlet life cycle.

Syntax:

public void init(ServletConfig config) throws ServletException

4. Calls the service( ) method

The **service( )** method performs actual task of servlet container. The web container is responsibleto call the service method each time the request for servlet is received. The service( ) invokes
the **doGet( ), doPost( ), doPut ( ), doDelete( )** methods based on the HTTP request.

Syntax:

public void service(ServletRequest request, ServletResponse response) throws ServletException,IOException

5. Calls the destroy( ) method

The destroy( ) method is executed when the servlet container remove the servlet from

thecontainer. It is called only once at the end of the life cycle of servlet.

Syntax:
public void destroy( )

6) Distinguish between HttpServlet and GenericServlet.

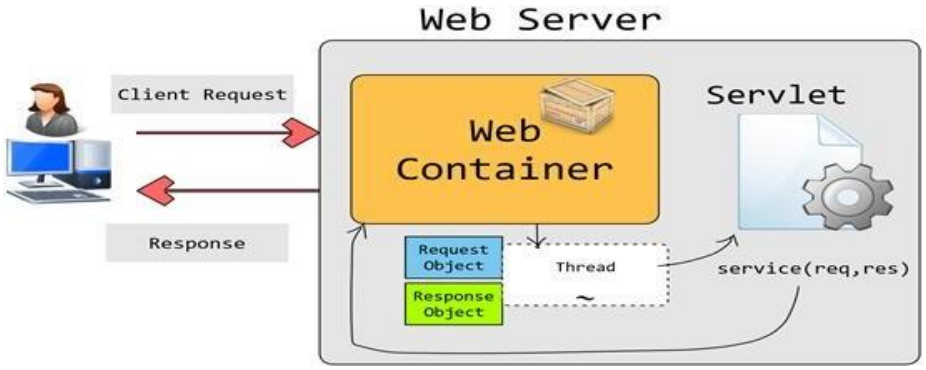
Answer:

Feature	HttpServlet	GenericServlet
Inheritance	Inherits from GenericServlet	Directly inherits from Servlet
Use Case	Designed for handling HTTP requests and responses	Can handle any type of protocol, not limited to HTTP
HTTP Methods	Provides methods for handling specific HTTP methods like doGet(), doPost(), doPut(), doDelete(), etc.	Does not provide specific methods for HTTP methods
Protocol Specific	HTTP-specific functionality	Protocol-independent functionality
Convenience	Offers convenience methods for dealing with HTTP headers, cookies, sessions, etc.	Requires manual handling for protocol-specific tasks
Initialization	Uses init() method for initialization with ServletConfig and can override to get ServletContext	Uses init() method, typically overridden to get ServletContext
Service Method	service(HttpServletRequest req, HttpServletResponse res)	service(ServletRequest req, ServletResponse res)
Request/Response	Works specifically with HttpServletRequest and HttpServletResponse	Works with ServletRequest and ServletResponse
Default Behaviour	Provides default implementations for various HTTP methods (e.g., doGet(), doPost() throwing 405 error if not overridden)	Requires overriding service() method to handle requests
Flexibility	Less flexible, specialized for web applications using HTTP	More flexible, suitable for a variety of protocols

7) Give a detailed note on the architecture of Java Servlet.

Answer:

Servlet Architecture is can be depicted from the image itself as provided below as follows:



Execution of Servlets basically involves the following steps:

1. The clients send the request to the webserver using any web browser.
2. The web server receives the request, the server verifies the request and transfers the request to the servlet container or web container.
3. The servlet container creates a request object, response object, and thread to handle the client request. The container stores the request and response objects in the thread object. For every client request, the container creates a pair of request, response, and thread objects. For example, if the servlet receives 2 client requests, the container will create 2 pairs of the request, response objects, and 2 threads. The multiple threads created by the container share the same OS-level process.
4. The thread will invoke the service() lifecycle method of the servlet by passing the request andthe response objects as the parameters. The business logic implemented in the service() methodwould be executed.
5. The container will provide the response object from the servlet to the web server.
6. The web server constructs the response data in the form of a web page and serves the page to the client browser.
7. The container will destroy the request, response, and thread.

8) How do Servlets handle the following?

- concurrent requests (4M)
- sessions (4M)

Answer:

Servlets handle concurrent requests and sessions using a combination of threading, synchronization, and session management mechanisms provided by the Servlet API. Following is the explanation of how these aspects are managed

Handling Concurrent Requests:

1. **Threading Model:**

   - Each servlet instance is typically shared among multiple threads. When a request arrives, the servlet container (such as Tomcat, Jetty, or others) assigns a new thread from its thread pool to handle the request. This means multiple threads can execute the service() method (and indirectly doGet(), doPost(), etc.) of the servlet concurrently.

2. **Synchronization:**

   - Since multiple threads may access shared resources concurrently, synchronization is crucial to prevent race conditions. Developers should ensure that any shared resources (like instance variables of the servlet or static variables) are properly synchronized.
   - Avoid using instance variables to store per-request information. Instead, use local variables within methods since each thread will have its own stack.

3. **Thread Safety:**

   - It's important to write thread-safe code in servlets. This often means avoiding shared mutable state or ensuring that access to shared resources is thread-safe, typically using synchronization blocks or concurrent data structures.

Handling Sessions:

1. **Session Management:**

   - HTTP is a stateless protocol, meaning each request is independent. To create a stateful experience, servlets use sessions to track user interactions across multiple requests.
   - A session is created by the servlet container when a user accesses the web application, and it can be identified using a session ID, which is typically stored in a cookie (JSESSIONID) or URL rewriting.

2. **Session Tracking:**

- **Cookies:** The most common method. The servlet container generates a unique session ID and sends it to the client as a cookie. On subsequent requests, the client sends this cookie back, allowing the server to associate the request with the session.
- **URL Rewriting:** If cookies are disabled, the session ID can be appended to URLs within the web application. The servlet container parses the session ID from the URL to associate the request with the correct session.

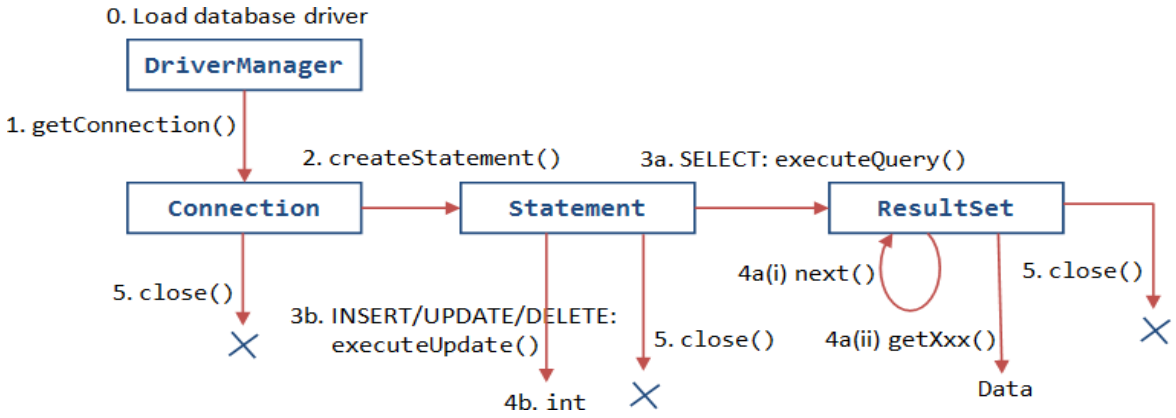
Example Code for Handling Sessions

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

@WebServlet("/SessionExample")
public class SessionExampleServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
        HttpSession session = request.getSession();
        // Set session attribute
        session.setAttribute("username", "john_doe");

        // Get session attribute
        String username = (String) session.getAttribute("username");

        response.getWriter().write("Hello, " + username);
    }
}
```

9) Summarize the basic steps involved in establishing a database connection using JDBC.

Answer:
Steps involved in establishing a database connection using JDBC

1. Import the Packages
2. Load the drivers using the forName() method
3. Register the drivers using DriverManager
4. Establish a connection using the Connection class object
5. Create a statement
6. Execute the query
7. Close the connections



Let us discuss these steps one by one:

Step 1: Import the Packages

```
import java.sql.*; // Importing database
import java.util.*; // Importing required classes
```

Step 2: Loading the drivers

- In order to begin with, you first need to load the driver before using it in the program. This is done by using Class.forName().
- The following example uses Class.forName() to load the Mysql driver as shown below as follows:

Class.forName("com.mysql.cj.jdbc.Driver");

Step 3: Register the drivers

- Registration is to be done once in your program.
- You can register a driver using DriverManager.registerDriver()
- DriverManager is a Java inbuilt class with a static member register. Here we call the constructor of the driver class at compile time.
- The following example uses DriverManager.registerDriver()to register the Oracle driver as shown below:

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver())
```

Step 4: Establish a connection using the Connection class object

- After loading the driver, establish connections as shown below as follows:

Connection con = DriverManager.getConnection(url,user,password)

user: Username from which your SQL command prompt can be accessed.

password: password from which the SQL command prompt can be accessed.

con: It is a reference to the Connection interface.

Url: Uniform Resource Locator which is created as shown below:

String url = " jdbc:oracle:thin:@localhost:1521:xe"

Step 5: Create a statement

- Once a connection is established you can interact with the database. The JDBCStatement, CallableStatement, and PreparedStatement interfaces define the methods that enable you to send SQL commands and receive data from your database. Use of JDBC Statement is as follows:

Statement st = con.createStatement();

Note: Here, con is a reference to Connection interface used in previous step.

Step 6: Execute the query

- Now comes the most important part i.e executing the query. The query here is an SQL Query. Now we know we can have multiple types of queries. Some of them are as follows:
  1. The query for updating/inserting a table in a database.
  2. The query for retrieving data.
- The **executeQuery()** method of the Statement interface is used to execute queries of retrieving values from the database. This method returns the object of ResultSet that can be used to get all the records of a table.
- The **executeUpdate(sql query)** method of the Statement interface is used to execute queries of updating/inserting.

Step 7: Closing the connections

- So finally we have sent the data to the specified location and now we are on the verge ofcompleting our task.
- By closing the connection, objects of Statement and ResultSet will be closed automatically.
- The close() method of the Connection interface is used to close the connection. It isshown below as follows:

con.close();

Example program for Connectivity between Java Program and Database

```
import java.io.*;
import java.sql.*;
class jdbcconnection {


   public static void main(String[] args) throws Exception
```

```
    {
        String url= "jdbc:mysql://localhost:3306/database_name"; // table
        detailsString username = "rootgfg"; // MySQL credentials
        String password = "gfg123";
        String query = "select *from students"; // query to be run
        Class.forName("com.mysql.cj.jdbc.Driver"); // Driver name
        Connection con = DriverManager.getConnection(url, username, password);
        System.out.println("Connection Established successfully");
        Statement st = con.createStatement();
        ResultSet rs = st.executeQuery(query); //
        Execute queryrs.next();
        String name = rs.getString("name"); // Retrieve name from db
        System.out.println(name); // Print result on console
        st.close(); // close
        statement
        con.close(); // close
        connection
        System.out.println("Connection Closed.....");
    }
}
```

10) How the architecture of JDBC is related with ODBC? Demonstrate with example.

Answer:

JDBC (Java Database Connectivity) and ODBC (Open Database Connectivity) are both APIs that allow applications to interact with a database. While JDBC is specific to Java, ODBC is language-agnostic. Despite their differences, the architecture of JDBC and ODBC has some conceptual similarities because both are designed to provide a standard interface for database connectivity, albeit in different environments. Let's explore how the architecture of JDBC relates to ODBC with an example.

Similarities in Architecture
Both JDBC and ODBC provide a means to:

- Establish a Connection to a database.
- Execute SQL Queries.
- Retrieve Results.
- Handle Transactions.
- Key Components

ODBC Architecture:

- **Application:** The client application using ODBC to interact with the database.
- **Driver Manager:** Manages communication between applications and database drivers.
- **Driver:** Translates ODBC function calls into specific calls for the database management system (DBMS).
- **Data Source:** Contains information required to connect to the database (like database name, directory, etc.).

JDBC Architecture:

- **Application:** The Java application using JDBC to interact with the database.
- **DriverManager:** Manages a list of database drivers and establishes a connection between the application and the driver.
- **Driver:** Interface that handles the communication with the database.
- **Connection:** A session with a specific database.
- **Statement:** An interface for executing SQL statements and returning results.
- **ResultSet:** Holds data retrieved from a database after executing a query.

Example Demonstration

ODBC Example:

A typical ODBC connection setup involves setting up a Data Source Name (DSN) and then connecting to the database using this DSN.

```
#include <windows.h>
#include <sql.h>
#include <sqlext.h>

void connectToDatabase() {
    SQLHENV env;
    SQLHDBC dbc;
    SQLHSTMT stmt;
    SQLRETURN ret;
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env);
    SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (void *)SQL_OV_ODBC3, 0);
    SQLAllocHandle(SQL_HANDLE_DBC, env, &dbc);
    SQLCHAR dsn[] = "DSN=mydsn;UID=myuser;PWD=mypassword;";

    ret = SQLDriverConnect(dbc, NULL, dsn, SQL_NTS, NULL, 0, NULL,
                SQL_DRIVER_COMPLETE);

    if (SQL_SUCCEEDED(ret)) {
        SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt);
        SQLExecDirect(stmt, (SQLCHAR *)"SELECT * FROM mytable;", SQL_NTS);
        // Fetch and display the results
        SQLFreeHandle(SQL_HANDLE_STMT, stmt);
    }


    SQLDisconnect(dbc);
```

```
    SQLFreeHandle(SQL_HANDLE_DBC, dbc);
    SQLFreeHandle(SQL_HANDLE_ENV, env);
}
```

JDBC Example:

JDBC requires adding the database driver to your classpath and then using JDBC API to connect and query the database.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class JDBCExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mydatabase";
        String user = "myuser";
        String password = "mypassword";
        try {
            Connection conn = DriverManager.getConnection(url, user, password);
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * FROM mytable");
            while (rs.next()) {
                System.out.println(rs.getString("column_name"));
            }
            rs.close();
            stmt.close();
            conn.close();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- **Connection Establishment:**

  ODBC: Uses SQLDriverConnect to establish a connection using a DSN.
  JDBC: Uses DriverManager.getConnection with a connection URL.

- **Executing SQL:**

  ODBC: Uses SQLExecDirect to execute a query.
  JDBC: Uses Statement.executeQuery to execute a query.

- **Handling Results:**

  ODBC: Handles results using SQLFetch (not shown in the example for brevity).
  JDBC: Handles results using ResultSet.

Bridging the Gap

JDBC-ODBC Bridge is a mechanism provided by JDBC to allow JDBC applications to access databases via the ODBC driver. This was part of early JDBC API implementations but is generally discouraged now in favour of native JDBC drivers due to performance and compatibility issues.

JDBC-ODBC Bridge Example:

```
import java.sql. Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class JDBCODBCBridgeExample {
    public static void main(String [] args) {
        String url = "jdbc:odbc:mydsn";
        String user = "myuser";
        String password = "mypassword";
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Connection conn = DriverManager.getConnection(url, user, password);
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * FROM mytable");
            while (rs.next()) {
                System.out.println(rs.getString("column_name"));
            }
            rs.close();
            stmt.close();
            conn.close();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

While ODBC is a language-independent API and JDBC is Java-specific, both provide a similar layered architecture for database connectivity. The JDBC-ODBC bridge served as a transitional tool, enabling JDBC applications to communicate with ODBC drivers, highlighting the conceptual relationship between the two APIs.