**Ans No. 1**

**(A)**

**Fill in the blanks in each of the following statements:**

1. If a class declares constructors, the compiler will not create a **default constructor** for that class.
2. The public methods of a class are also known as the class’s **interface** or **API.**
3. Lists and tables of values can be stored in **arrays** and **collections**
4. The number used to refer to a particular array element is called the element’s **index**.
5. A variable known only within the method in which it’s declared is called a **local variable.**
6. It’s possible to have several methods with the same name that each operate on different types or numbers of arguments. This feature is called method **overloading**.
7. Typically, **for** statements are used for counter-controlled repetition and **while** statements for sentinel-controlled repetition.
8. Methods that perform common tasks and do not require objects are called **static methods**

**(B)**

**Write a Java statement or a set of Java statements to accomplish each of the following tasks:**

**a) Sum the odd integers between 1 and 99, using a for statement. Assume that the integer variables sum and count have been declared.**

**Ans:**

**public class SampleProgram {**

**public static void main(String[] args) {**

**int** sum = 0;

**for** (int count = 1; count <= 99; count += 2)

{

sum += count;

}

**System.out.println**("The sum of the odd integers between 1 and 99 is " + sum);

**}**

**}**

**b) Print the integers from 1 to 20, using a while loop and the counter variable i. Assume that the variable i has been declared, but not initialized. Print only five integers per line.**

**Ans:**

**public class SampleProgram** {

**public static void main(String[] args)** {

**int** i = 1;

**int** count = 0;

**while** (i <= 20)

{

**System.out.print**(i + " ");

i++;

count++;

**if** (count == 5)

{

**System.out.println**();

count = 0;

}

}

}

}

**(C) Repeat part (c), using a for statement.**

**Ans:**

**public class SampleProgram** {

**public static void main(String[] args)** {

**int** i = 1;

**for** (i = 1; i <= 20; i++)

{

**System.out.print**(i + " ");

**if** (i % 5 == 0)

{

**System.out.println**();

}

}

}

}

**(C)**

**(i) What gives Java its ‘write once and run anywhere’ nature?**

**Ans:**

Java’s ‘write once and run anywhere’ nature is based on its use of the Java Virtual Machine (JVM). The JVM is a software layer that acts as an interpreter between the Java bytecode and the underlying hardware and operating system. The Java bytecode is a standard format that is generated by compiling the Java source code. The bytecode is platform-independent, meaning that it can run on any device that has a compatible JVM installed. The JVM translates the bytecode into instructions that are understood by the specific hardware and operating system, thus enabling Java applications to run on different platforms without any modifications.

**ii) What happens at runtime during Java compilation?**

**Ans:**

During Java compilation, the source code is converted into bytecode, which is a platform-independent format that can run on any device that has a compatible Java Virtual Machine (JVM) installed. The JVM is a software layer that acts as an interpreter between the bytecode and the underlying hardware and operating system**. At runtime, the JVM performs the following steps to execute the bytecode:**

* **Class loading**: The JVM loads the bytecode of the main class and any other classes referenced in the program into the memory using a class loader. A class loader is an object that creates a namespace of class bodies that are identified by a string name. There are two types of class loaders: primordial and non-primordial. The primordial class loader is embedded into all the JVMs and is the default class loader. A non-primordial class loader is a user-defined class loader that can customize the class loading process.
* **Bytecode verification**: The JVM verifies that the bytecode is valid and does not violate any security or integrity rules. The bytecode verifier checks that the instructions are well-formed, do not access illegal data, do not overflow or underflow the stack, do not cause memory leaks, and do not perform any harmful actions.

**iii) Can you save a Java source file by another name than the class name?**

**Ans:**

It depends on whether the class is declared as public or not. If the class is declared as public, then the Java source file name must be the same as the public class name, and it must have a .java extension. For example, if the public class name is Rectangle, then the source file name must be Rectangle.java. This is because a public class can be accessed by any other class in any package, and the compiler needs to locate the source file by matching the class name and the file name.

However, if the class is not declared as public, then the Java source file name can be different from the class name, and it can still have a .java extension. For example, if the class name is Helper, then the source file name can be anything, such as GeeksforGeeks.java. This is because a non-public class can only be accessed by other classes in the same package, and the compiler does not need to match the class name and the file name.

Therefore, you can save a Java source file by another name than the class name only if the class is not declared as public. Otherwise, you will get a compilation error.

**iv) Can you have multiple classes in a java source file?**

**Ans:**

Yes, you can have multiple classes in a java source file, but there are some rules and limitations that you need to follow. Here are some of the main points:

* You can only have **one public class** per source file, and the name of the source file must be the same as the name of the public class. For example, if you have a public class named HelloWorld, then the source file name must be HelloWorld.java.
* You can have **multiple non-public classes** per source file, and the name of the source file can be different from the name of any of the non-public classes. For example, you can have a source file named GeeksforGeeks.java that contains a non-public class named Helper and another non-public class named Geek.
* The purpose of having multiple classes in one source file is to **bundle related support functionality** (internal data structures, support classes, etc) together with the main public class. This can improve the readability and maintainability of your code, as well as hide some implementation details from other classes.
* However, there are some drawbacks and challenges of having multiple classes in one source file. For instance, you may face some **compilation errors** if you try to refer to a non-public class from another source file without also referring to the public class in the same file. This is because the compiler may not be able to locate the source file that contains the non-public class. Another issue is that having multiple classes in one source file may make your code **less modular** and **harder to reuse**, as you cannot easily separate or import individual classes.

**(D)**

**Write a Java program to create and display unique three-digit number using 1, 2, 3, 4. Also count how many three-digit numbers are there.**

**Ans:**

**public class UniqueNumbers {**

**public static void main(String[] args) {**

**int[]** digits = {1, 2, 3, 4};

**int** count = 0;

**for (int** i = 0; i < digits.length; i++)

{

**for (int** j = 0; j < digits.length; j++)

{

**if** (j == i) **continue;**

**for (int** k = 0; k < digits.length; k++)

{

**if** (k == i || k == j)

**continue;**

**int** number = digits[i] \* 100 + digits[j] \* 10 + digits[k];

**System.out.println**(number);

count++;

**}**

**}**

**}**

**System.out.println**("Total number of unique numbers: " + count);

**}**

**}**

**Ans No. 2**

**(A)**

**What are the various access specifiers in Java? Write an example of public access modifier.**

**Ans:**

The access specifiers in Java are keywords that define the access scope of a class, method, constructor, or field. There are four types of access specifiers in Java: **public**, **private**, **protected**, and **default**.

The public access modifier makes a class, method, constructor, or field accessible from any other class or package. It is the least restrictive access modifier. For example:

**public class** PublicClass

{

**public int** x = 10;

**public** PublicClass()

{

publicMethod();

}

**public void** publicMethod()

{

**System.out.println**("This is a public method.");

}

}

In this example, the class PublicClass, its field x, its constructor, and its method publicMethod are all declared with the public modifier. This means that they can be accessed from any other class or package. For instance, we can create an object of PublicClass and access its members from another package:

**package com.example.another;**

**import com.example.PublicClass;** // Import the public class

**public class** **AnotherClass** {

**public static void main**(String[] args) {

**PublicClass** pc = new **PublicClass**(); // Create an object of the public class

**System.out.println**(pc.x); // Access the public field

**pc.publicMethod**(); // Call the public method

}

}

**(B)**

**Write the rules of Constructor. What is the purpose of a default constructor? Explain with example.**

**Ans:**

Some rules of constructor are:

* A constructor is a special method that is used to initialize objects of a class.
* A constructor must have the same name as the class name and no return type.
* A constructor can be overloaded with different parameters to create different types of objects.
* A constructor can use the **this** keyword to refer to the current object or invoke another constructor of the same class.
* A constructor can use the **super** keyword to invoke the constructor of the superclass.

The purpose of a default constructor is to provide a default initialization for the object’s attributes. A default constructor is a constructor that has no parameters and an empty body. If a class does not define any constructor, the compiler automatically generates a default constructor for the class. For example:

**public class** **Person** {

**String** name;

**int** age;

}

// The compiler will generate a default constructor like this

**public Person**() {

super(); // calls the constructor of Object class

this.name = null; // default value for String

this.age = 0; // default value for int

}

**(C)**

**i) What is the output of the following Java program?**

**public class Test**

{

Test(**int** a, **int** b)

{

**System.out.println**(“a = “+a+” b = “+b); }

Test(**int** a, **float** b)

{ **System.out.println**(“a = “+a+ “ b = “+b);

}

**public static void main** (String args[])

{

**byte** a = 10;

**byte** b = 15;

Test test = new Test(a,b);

} }

**Ans:**

The output of the following Java program is:

**a = 10 b = 15**

This is because the program invokes the constructor **Test(int a, int b)** with the arguments**a** and **b** of type **byte**. The byte values are automatically promoted to int values when passed to the constructor. The constructor then prints the values of **a** and **b** as int values. The constructor **Test(int a, float b)** is not invoked because there is no implicit conversion from byte to float.

**(ii)**

**class Test**

{

**int** i; }

**public class** Main

{

**public static void main** (String args[])

{

Test test = new Test();

**System.out.println**(test.i);

} }

**Ans:**

The output of the following Java program is:

**0**

This is because the program creates an object of the class**Test** and prints the value of its field **i.** The field**i** is of type **in**t and has a default value of **0** if it is not initialized. The class **Test** does not have a constructor that can initialize the field**i**. Therefore, the value of**i** remains**0** for the object **test.**

**(iii)**

**class** Test

{

**public static void main** (String args[])

{

for(**int i**=0; 0; i++)

{

**System.out.println**(“Hello PSTU CSE”);

} } }

**Ans:**

The code that you have written has a syntax error. The condition of the for loop should be a boolean expression that evaluates to either true or false. However, you have written 0 as the condition, which is an int value and not a boolean value. Therefore, the compiler will not be able to execute the for loop and will give an error message. You need to change the condition to a valid boolean expression.

**(D)**

**Write a Java program to print a pyramid using star pattern. Number of rows input from keyboard.**

**Ans:**

I can write a Java program to print a pyramid using star pattern. The number of rows will be input from the keyboard. Here is the code:

**import java.util.Scanner**; // Import the Scanner class

**public class** Pyramid {

**public static void main**(String[] args) {

Scanner sc = new Scanner(System.in); // Create a Scanner object

**System.out.print**("Enter the number of rows: "); // Prompt the user for input

**int** n = sc.nextInt(); // Read the input as an integer

sc.close(); // Close the scanner

**for** (int i = 1; i <= n; i++) { // Loop from 1 to n

**for** (int j = 1; j <= n - i; j++) { // Loop to print spaces

**System.out.print**(" ");

}

**for** (**int** k = 1; k <= 2 \* i - 1; k++) { // Loop to print stars

**System.out.print**("\*");

}

**System.out.println**(); // Move to the next line

}

}

}

**Ans No. 3**

**(A)**

**What is the static variable? Explain a java program with and without static variable.**

**Ans:**

A static variable is a variable that belongs to the class and not to any specific instance of the class. A static variable is initialized only once at the start of the execution and shared among all objects of the class. A static variable can be accessed directly by the class name without creating an object of the class.

**Here is an example of a Java program with a static variable:**

// A class with a static variable

**public class** Counter {

// A static variable to count the number of objects created

**public static int** count = 0;

// A constructor that increments the static variable

**public** Counter() {

count++;

}

// A method that prints the value of the static variable

**public void** showCount() {

**System.out.println**("The number of objects created so far is " + count);

}

}

// A class to test the static variable

**public class** Test {

**public static void main**(String[] args) {

// Create three objects of Counter class

Counter c1 = new Counter();

Counter c2 = new Counter();

Counter c3 = new Counter();

// Call the showCount method on each object

c1.showCount(); // output: The number of objects created so far is 3

c2.showCount(); // output: The number of objects created so far is 3

c3.showCount(); // output: The number of objects created so far is 3

// Access the static variable directly by the class name

**System.out.println**("The value of count is " + Counter.count); // output: The value of count is 3

}

}

In this example, the class Counter has a static variable count that keeps track of how many objects of the class are created. The constructor of the class increments the static variable every time a new object is created. The method showCount prints the value of the static variable. The main method creates three objects of Counter class and calls the showCount method on each object. The output shows that the value of the static variable is 3 for all objects, because there is only one copy of the variable in memory. The main method also accesses the static variable directly by using the class name Counter.count.

**Here is an example of a Java program without a static variable:**

// A class without a static variable

**public class** Counter {

// A non-static variable to count the number of objects created

**public int** count = 0;

// A constructor that increments the non-static variable

**public** Counter() {

count++;

}

// A method that prints the value of the non-static variable

**public void** showCount() {

**System.out.println**("The number of objects created by this object is " + count);

}

}

// A class to test the non-static variable

**public class** Test {

**public static void main**(String[] args) {

// Create three objects of Counter class

Counter c1 = new Counter();

Counter c2 = new Counter();

Counter c3 = new Counter();

// Call the showCount method on each object

c1.showCount(); // output: The number of objects created by this object is 1

c2.showCount(); // output: The number of objects created by this object is 1

c3.showCount(); // output: The number of objects created by this object is 1

// Cannot access the non-static variable directly by the class name

**System.out.println**("The value of count is " + Counter.count); // compile-time error: non-static variable count cannot be referenced from a static context

}

}

In this example, the class Counter has a non-static variable count that keeps track of how many objects are created by each object. The constructor of the class increments the non-static variable every time a new object is created. The method showCount prints the value of the non-static variable. The main method creates three objects of Counter class and calls the showCount method on each object. The output shows that the value of the non-static variable is 1 for each object, because each object has its own copy of the variable in memory. The main method cannot access the non-static variable directly by using the class name Counter.count, because it is not a class-level variable.

**(B)**

**(i) What is the difference between static (class) method and instance method?**

**Ans:**

The difference between static (class) method and instance method is that:

* A static method is a method that belongs to the class and can be called without creating an object of the class. A static method is declared with the static keyword and can access only static variables and static methods of the class. A static method can be invoked by using the class name, such as ClassName.staticMethod().
* An instance method is a method that belongs to an object of the class and can access the instance variables and instance methods of the object. An instance method is not declared with the static keyword and can also access the static variables and static methods of the class. An instance method can be invoked by using the object reference, such as objectName.instanceMethod().

**(ii) What are the main uses of this keyword?**

**Ans:**

The main uses of this keyword in Java are:

* To refer to the current class instance variable when there is a conflict with a local variable or a parameter. For example, this.x = x; assigns the value of the parameter x to the instance variable x of the current object.
* To invoke the current class method, either explicitly or implicitly. For example, this.method(); calls the method method() of the current object.
* To invoke the current class constructor from another constructor of the same class. For example, this(10); calls the constructor that takes an int parameter from another constructor.
* To pass the current object as an argument in a method call or a constructor call. For example, obj.method(this); passes the current object as a parameter to the method method() of another object obj.
* To return the current object from a method. For example, return this; returns the current object as the result of the method.

**(C)**

**Define Object and Class. Write Object and Class Example: main outside the class and main within the class.**

**Ans:**

An object is an instance of a class that has state and behavior. State refers to the values of the attributes or fields of the object, and behavior refers to the actions or methods that the object can perform. For example, a car is an object that has state (such as color, model, speed, etc.) and behavior (such as start, stop, accelerate, brake, etc.).

A class is a blueprint or template that defines the common state and behavior of a group of objects. A class specifies the fields and methods that the objects of that class will have. A class can also have constructors, blocks, and nested classes or interfaces.

Here are two examples of how to write a Java program with a class and an object. One example has the main method inside the class, and the other example has the main method outside the class.

**Object and Class Example: main within the class**

// A class named Student with two fields: id and name

**public class** Student {

// Declare the fields

**int** id;

**String** name;

// Define the main method inside the class

**public static void main**(String[] args) {

// Create an object of Student class using new keyword

Student s1 = new Student();

// Assign values to the fields of the object

s1.id = 101;

s1.name = "Alice";

// Print the values of the fields of the object

**System.out.println**("Student id: " + s1.id);

**System.out.println**("Student name: " + s1.name);

}

}

**Object and Class Example: main outside the class**

// A class named Student with two fields: id and name

**public class** Student {

// Declare the fields

**int** id;

**String** name;

}

// Another class named Main with the main method

**public class** Main {

// Define the main method outside the Student class

**public static void main**(String[] args) {

// Create an object of Student class using new keyword

Student s1 = new Student();

// Assign values to the fields of the object

s1.id = 101;

s1.name = "Alice";

// Print the values of the fields of the object

**System.out.println**("Student id: " + s1.id);

**System.out.println**("Student name: " + s1.name);

}

}

**(D)**

**Write a Java program to sort an array of given integers using the Bubble sorting Algorithm.**

**Original Array**: [7, -5, 3, 2, 1, 0, 45]

**Sorted Array:** [-5, 0, 1, 2, 3, 7, 45]

**Ans:**

I can write a Java program to sort an array of given integers using the Bubble sorting Algorithm. The Bubble sorting Algorithm is a simple sorting algorithm that compares adjacent elements in an array and swaps them if they are in the wrong order. The algorithm repeats this process until the array is sorted.

**Here is the code:**

// A method that implements the Bubble sorting Algorithm

**public static void** bubbleSort(**int**[] arr) {

**int** n = arr.length;

**for** (**int** i = 0; i < n - 1; i++) {

**for** **(int** j = 0; j < n - i - 1; j++) {

**if** (arr[j] > arr[j + 1]) {

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

// A method that prints an array

**public static void** printArray(int[] arr) {

**for** (**int** element : arr) {

**System.out.print**(element + " ");

}

**System.out.println**();

}

// The main method

**public static void main**(String[] args) {

**int**[] arr = {7, -5, 3, 2, 1, 0, 45};

**System.out.print**("Original Array: ");

printArray(arr);

bubbleSort(arr);

**System.out.print**("Sorted Array: ");

printArray(arr);

}

**Ans No. 4**

**(A)**

**Differentiate between the throw and throws keyword.**

**Ans:**

The throw and throws keywords are both related to exception handling in Java, but they have different meanings and uses. Here are some of the main differences between them:

* The throw keyword is used to explicitly throw an exception from a method or a block of code. For example, **throw new ArithmeticException("/ by zero");** will throw an ArithmeticException with the message “/ by zero”. The throw keyword can be used to throw both checked and unchecked exceptions, but it is mainly used to throw custom exceptions.
* The throws keyword is used to declare the list of possible exceptions that a method might throw. For example, **public static int divideNum(int m, int n) throws ArithmeticException** declares that the method divideNum might throw an ArithmeticException. The throws keyword can only be used with method signatures, and it can declare both checked and unchecked exceptions. However, it is mainly used to declare checked exceptions that are not handled within the method.
* The throw keyword is followed by an instance of an exception class, such as **throw new NullPointerException("demo");.** The throws keyword is followed by one or more class names of exception classes, separated by commas, such as **throws IOException, SQLException;.**
* The throw keyword stops the execution of the program immediately after the exception is thrown, unless it is caught by a try-catch block. The throws keyword does not stop the execution of the program, but it indicates that the caller of the method should handle the exception or propagate it further.
* The throw keyword is used to create a new exception object and throw it. The throws keyword is used to rethrow an existing exception object that was thrown by another method or by the Java runtime system.

**(B)**

**“Aggregation represents HAS-A relationship.”-explain with example.**

**Ans:**

Aggregation represents HAS-A relationship means that one object has a reference to another object as a part of its state. For example, a car has an engine, a person has a name, a book has an author, etc. Aggregation implies that the two objects are not dependent on each other’s existence and can have their own lifecycles. For example, if we destroy the car, the engine still exists; if we change the name of the person, the person still exists; if we delete the book, the author still exists.

One of the benefits of using aggregation is to achieve code reusability. For example, we can define a class for Engine and use it as a field in different classes, such as Car, Truck, Motorcycle, etc. We can also access the methods and attributes of the aggregated object through the reference. For example, we can call engine.start() or engine.stop() from the car object.

**Here is an example of aggregation in Java code:**

// A class for Engine

**public class** Engine {

**private String** type;

**public** Engine(String type) {

this.type = type;

}

**public void** showType() {

**System.out.println**("The engine type is " + type);

}

**public void** start() {

**System.out.println**("The engine is started");

}

**public void** stop() {

**System.out.println**("The engine is stopped");

}

}

// A class for Car that has an Engine as a field

**public class** Car {

**private String** model;

**private String** color;

**private Engine** engine;

**public** Car(String model, String color, Engine engine) {

this.model = model;

this.color = color;

this.engine = engine;

}

// A method that prints the car details

**public void** showDetails() {

**System.out.println**("The car model is " + model);

**System.out.println**("The car color is " + color);

engine.showType();

}

// A method that drives the car

**public void** drive() {

**System.out.println**("The car is driving");

engine.start();

}

// A method that parks the car

**public void** park() {

**System.out.println**("The car is parked");

engine.stop();

}

}

// A class to test the aggregation relationship

**public class** Test {

**public static void main**(String[] args) {

Engine e1 = new Engine("V8");

Car c1 = new Car("Ford Mustang", "red", e1);

c1.showDetails();

c1.drive();

c1.park();

}

}

**(C)**

**Is it possible to make any class read-only or write-only in java? How?**

**Ans:**

It is not possible to make any class read-only or write-only in Java, because there is no such keyword or modifier that can enforce this restriction. However, there are some ways to design a class that can achieve a similar effect by using other features of the language, such as access modifiers, final keyword, getters and setters, and immutability.

A read-only class is a class that does not allow any changes to its internal state after it is created. This means that the class has only getter methods and no setter methods, and all its fields are either private or final or both. A read-only class is also known as an immutable class, because its objects cannot be modified once they are initialized. An example of a read-only class is the String class in Java.

A write-only class is a class that does not allow any access to its internal state except for setting it. This means that the class has only setter methods and no getter methods, and all its fields are private. A write-only class is also known as an opaque class, because its objects hide their internal state from the outside world. An example of a write-only class is the PasswordField class in JavaFX.

**Here are some examples of how to create a read-only and a write-only class in Java:**

**A read-only class for Point:**

**public final class** Point {

**private** final int x;

**private** final int y;

**public** Point(int x, int y) {

this.x = x;

this.y = y;

}

**public int** getX() {

**return** x;

}

**public int** getY() {

**return** y;

}

}

**A write-only class for Logger:**

**public class** Logger {

**private** **String** message;

**public void** setMessage(String message) {

this.message = message;

// Write the message to a file or console or database

**System.out.println**(message);

}

}

**(D)**

**What is the use of instance initializer block while we can directly assign a value in instance data member?**

**Ans:**

An instance initializer block is a block of code that is executed when an object of a class is created. It is used to initialize the instance variables of the object with some values that are not known at compile time or that require some logic to be calculated. For example, we can use an instance initializer block to assign a random number or a current date or time to an instance variable.

We can directly assign a value to an instance data member, but only if the value is a constant or a literal that is known at compile time. For example, we can directly assign **int x = 10**; or **String name = "Alice";** to an instance data member. However, if the value depends on some runtime condition or calculation, we cannot directly assign it to an instance data member. For example, we cannot directly assign **int y = x + 5;** or **String** **date = new Date().toString();** to an instance data member, because the values of **x** and **new Date()** are not known at compile time.

**Here is an example of how to use an instance initializer block to assign a value to an instance data member:**

// A class for Employee

**public class** Employee {

**private int** id;

{

id = (**int**) (Math.random() \* 1000) + 1;

}

**public int** getId() {

**return** id;

}

}

// A class to test the instance initializer block

**public class** Test {

**public static void main**(String[] args) {

Employee e1 = new Employee();

Employee e2 = new Employee();

// Print the ids of the objects

**System.out.println**("The id of e1 is " + e1.getId());

**System.out.println**("The id of e2 is " + e2.getId());

}

}

**(E)**

**How can you achieve abstraction in java?**

**Ans:**

Abstraction in Java is a concept that allows us to hide the implementation details of a class or an interface and only show the essential features to the users. Abstraction helps us to achieve simplicity, modularity, and security in our code. There are two ways to achieve abstraction in Java: abstract classes and interfaces.

* An abstract class is a class that is declared with the abstract keyword and cannot be instantiated. An abstract class can have both abstract and concrete methods. An abstract method is a method that is declared without a body and must be overridden by the subclasses. A concrete method is a method that has a body and can be inherited by the subclasses. An abstract class can also have fields, constructors, blocks, and nested classes.
* An interface is a collection of abstract methods and constants that are declared with the interface keyword. An interface cannot have any concrete methods, fields, constructors, blocks, or nested classes. An interface can be implemented by one or more classes using the implements keyword. A class that implements an interface must provide the body for all the abstract methods of the interface.

**Ans No. 5**

**(A)**

**Write a java program for demonstrating several thread states.**

**Ans:**

I can write a Java program for demonstrating several thread states. A thread is a lightweight sub-process that can execute a task concurrently with other threads. A thread can have one of the following states:

* **New**: The thread is created but not yet started.
* **Runnable**: The thread is ready to run and waiting for the CPU time.
* **Running**: The thread is executing its task.
* **Blocked**: The thread is waiting for a monitor lock to enter a synchronized block or method.
* **Waiting**: The thread is waiting indefinitely for another thread to perform a specific action, such as notifying or joining.
* **Timed waiting**: The thread is waiting for a specified amount of time for another thread to perform a specific action, such as sleeping or waiting with a timeout.
* **Terminated**: The thread has completed its task or exited abnormally.

**Here is the code:**

// A class that implements Runnable interface

**public class** MyThread implements Runnable {

**private String** name;

**public** MyThread(String name) {

this.name = name;

}

**public void** printState() {

**System.out.println**(name + " is " + Thread.currentThread().getState());

}

// Override the run method of Runnable interface

**public void** run() {

printState();

**for** (**int** i = 1; i <= 5; i++) {

**System.out.println**("Iteration " + i + " by " + name);

try {

Thread.sleep(500);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

printState();

}

}

// A class to test the thread states

**public class** Test {

**public static void main**(String[] args) throws InterruptedException {

MyThread t1 = new MyThread("Thread-1");

MyThread t2 = new MyThread("Thread-2");

Thread th1 = new Thread(t1);

Thread th2 = new Thread(t2);

t1.printState(); // output: Thread-1 is NEW

t2.printState(); // output: Thread-2 is NEW

th1.start();

th2.start();

t1.printState(); // output: Thread-1 is RUNNABLE

t2.printState(); // output: Thread-2 is RUNNABLE

th1.join();

th2.join();

t1.printState(); // output: Thread-1 is TERMINATED

t2.printState(); // output: Thread-2 is TERMINATED

}

}

**(B)**

**“Java doesn’t allow the return type-based overloading, but JVM always allows return type-based overloading.”- justify the statement with example.**

**Ans:**

The statement means that Java does not allow two methods in the same class to have the same name and parameters but different return types, because it would cause ambiguity and confusion for the compiler and the programmer. However, the Java Virtual Machine (JVM), which is the runtime environment that executes the Java bytecode, does allow such methods to exist, because it uses a different mechanism to identify and invoke them.

**For example, consider the following Java code:**

// A class with two methods that have the same name and parameters but different return types

**public class** Test {

**public int** add(int a, int b) {

**return** a + b;

}

**public double** add(int a, int b) {

**return** a + b + 0.5;

}

// A main method that tries to call the methods

**public static void main**(String[] args) {

Test t = new Test();

int x = t.add(10, 20);

double y = t.add(10, 20);

}

}

This code will not compile in Java, because it violates the rule of return type-based overloading. The compiler will give an error message saying that the method add(int,int) is already defined in the class Test. The compiler cannot distinguish between the two methods based on their return types, because they are not part of the method signature.

However, if we look at the bytecode generated by the compiler for this class, we will see something like this:

**public int** add(int, int);

descriptor: (II)I

flags: ACC\_PUBLIC

Code:

stack=2, locals=3, args\_size=3

0: iload\_1

1: iload\_2

2: iadd

3: ireturn

**public double** add(int, int);

descriptor: (II)D

flags: ACC\_PUBLIC

Code:

stack=4, locals=3, args\_size=3

0: iload\_1

1: i2d

2: iload\_2

3: i2d

4: dadd

5: ldc2\_w #16 // double 0.5d

8: dadd

9: dreturn

**public static void main**(java.lang.String[]);

descriptor: ([Ljava/lang/String;)V

flags: ACC\_PUBLIC, ACC\_STATIC

Code:

stack=3, locals=4, args\_size=1

0: new #18 // class Test

3: dup

4: invokespecial #19 // Method "<init>":()V

7: astore\_1

8: aload\_1

9: bipush 10

11: bipush 20

13: invokevirtual #20 // Method add:(II)I or Method add:(II)D ?

...

We can see that the JVM uses a different way to identify and invoke the methods. It uses a descriptor that includes the return type as well as the parameter types of the method. For example, (II)I means a method that takes two ints as parameters and returns an int, and (II)D means a method that takes two ints as parameters and returns a double. The JVM can use these descriptors to find and execute the correct method at runtime.

However, this does not mean that we can write such methods in Java and expect them to work. The compiler will still give an error and prevent us from creating such ambiguous methods. The JVM allows return type-based overloading only for compatibility reasons with other languages that run on it.

**(C)**

**Multiple inheritance is not supported through class in java, but it is possible by an interface, why?**

**Ans:**

Multiple inheritance is a feature that allows a class to inherit from more than one superclass. Java does not support multiple inheritance through class, because it can cause ambiguity and complexity in the code. For example, if a class inherits from two superclass that have the same method name but different implementations, it is not clear which method the subclass should inherit or override. This is known as the diamond problem.

However, Java supports multiple inheritance through interface, because an interface is a collection of abstract methods and constants that do not have any implementation details. A class can implement multiple interfaces and provide the body for all the abstract methods. This does not cause any ambiguity or complexity, because the class has to define its own behavior for each method. An interface can also extend multiple interfaces and inherit their abstract methods.

**Here is an example of how to use multiple inheritance through interface in Java:**

**public** interface Swimmable {

**public void** swim();

}

**public** interface Flyable {

**public void** fly();

}

**public class** Duck implements Swimmable, Flyable {

**public void** swim() {

**System.out.println**("The duck is swimming");

}

**public void** fly() {

**System.out.println**("The duck is flying");

}

}

**public class** Test {

**public static void main**(String[] args) {

Duck d = new Duck();

d.swim(); // output: The duck is swimming

d.fly(); // output: The duck is flying

}

}

**(D)**

**Can we initialize blank final variable? How?**

**Ans:**

Yes, we can initialize a blank final variable in Java. A blank final variable is a final variable that is not initialized at the time of declaration. A final variable is a variable that cannot be changed once it is assigned a value. A blank final variable can be initialized only once, either in a constructor or in an instance initializer block.

**Here is an example of how to initialize a blank final variable in Java:**

**public class** Student {

**private final int** id;

**public** Student(int id) {

this.id = id;

}

**public int** getId() {

**return** id;

}

}

**public class** Test {

**public static void main**(String[] args) {

Student s1 = new Student(101);

**System.out.println**("The id of s1 is " + s1.getId());

// s1.id = 102;

}

}

**Ans No. 6**

**(A)**

**What will be the output of the following Java programs?**

**(i)**

class Dog{

public static void main(String args[]){

Dog d=null;

 System.out.println(d instanceof Dog);

}  }

**Output:** False

**(ii)**

**class** A{

**protected void** msg(){**System.out.println**(“Hello java”);}}

**public class** Simple extends A{

**void** msg(){**System.out.println**(“Hello java”);}

**public static void main**(String args[]){

Simple obj=new Simple();

obj.msg();  }}

**Output:** Hello java

**(iii)**

**public class** JavaExceptionExample

{

**public static void main**(String args[]){

     try{

**int** data=100/0;   }catch(ArithmeticException e){**Syste**

**m.out.println**(e);}

**System.out.println**(“rest of the code...

“);

  }  }

**Output:** java.lang.ArithmeticException: / by zero

rest of the code...

**(iv)**

**class** Animal{

**void** eat(){System.out.println(&quot;eating...&quot;);}  }

**class** Dog extends Animal{

**void** bark(){System.out.println(&quot;barking...&quot;);}  }

**class** Cat extends Animal{

**void** meow(){System.out.println(&quot;meowing...&quot;);}  }

**class** TestInheritance3{

**public static void main**(String args[]){

Cat c=new Cat();

c.meow();

c.eat();

c.bark();}}

**Output:** meowing...

eating...

compile-time error: cannot find symbol

symbol: method bark()

location: variable c of type Cat

**(B)**

**How to access package from another package?**

**Ans:**

To access a package from another package in Java, we need to use the import statement. The import statement allows us to use the classes and interfaces of a package without specifying their fully qualified names. For example, if we want to use the ArrayList class of the java.util package, we can write:

**import** java.util.ArrayList; // Import the ArrayList class

// Or

import java.util.\*; // Import all the classes and interfaces of the java.util package

Then, we can create an object of the ArrayList class without using its full name:

ArrayList<String> list = new ArrayList<String>(); // Create an object of ArrayList class

If we do not use the import statement, we have to use the fully qualified name of the class or interface every time we use it. For example:

java.util.ArrayList<String> list = new java.util.ArrayList<String>(); // Create an object of ArrayList class using its full name

This can make the code longer and less readable. Therefore, it is recommended to use the import statement to access a package from another package.

**(C)**

**What is the purpose of join method?**

**Ans:**

The purpose of the join method in Java is to wait for a thread to finish its execution before continuing with the current thread. The join method is defined in the Thread class and can take an optional parameter that specifies the maximum time to wait in milliseconds. The join method can throw an InterruptedException if the current thread is interrupted while waiting for the other thread.

**(D)**

**How to perform two tasks by two threads?**

**Ans:**

To perform two tasks by two threads in Java, we can use the following steps:

* Define a class that implements the Runnable interface and override the run method to specify the task that the thread will execute.
* Create two objects of the class that implements Runnable and pass them as targets to two objects of Thread class.
* Start the two threads by calling the start method on them.
* The two threads will execute their tasks concurrently and independently.

**(E)**

**What is the Thread Scheduler and what is the difference between preemptive scheduling and time slicing?**

**Ans:**

The Thread Scheduler is the component of the Java Virtual Machine (JVM) that determines the execution order of multiple threads on a single processor (CPU). It decides which thread should run and for how long, based on the thread priority and the scheduling algorithm.

Preemptive scheduling and time slicing are two types of scheduling algorithms that the Thread Scheduler can use. The difference between them is:

* Preemptive scheduling: The highest priority thread executes until it enters the waiting or dead states or a higher priority thread comes into existence. This means that a running thread can be interrupted by another thread with a higher priority.
* Time slicing: A thread executes for a predefined slice of time and then reenters the pool of ready threads. This means that a running thread can be interrupted by another thread with the same priority after a certain amount of time.

Preemptive scheduling can improve the responsiveness of the system, but it can also cause starvation for lower priority threads. Time slicing can ensure fairness among threads, but it can also increase the overhead of context switching. The choice of the scheduling algorithm depends on the requirements and characteristics of the application.