UTNIT-5

Q.1 Explain the different states in lifecycle of Thread?

New (born) state
Example: Creating a new thread using the Thread class constructor.
Use case: Creating a new thread to perform a background task while the main Thread continues with other operations
Runnable state
Example: After calling the start() method on a thread, it enters the runnable state.
Use case: Multiple threads competing for CPU time to perform their tasks concurrently.
Running state
Example: When a thread executes its code inside the run() method.
Use case: A thread executing a complex computation or performing a time-consuming task.
Blocked state:
Example: When a thread tries to access a synchronized block or method, but another thread already holds the lock.
Use case: Multiple threads accessing a shared resource can only be obtained by a single Thread, such as a database or a file.
Waiting state:
Example: Using the wait method inside a synchronized block, a thread can wait until another thread calls the notify() or notify All() methods to wake it up.
Use case: Implementing the producer-consumer pattern, where a thread waits for a specific condition to be met before continuing its execution
Timed waiting state:
Example: Using methods like sleep (milliseconds) or join (milliseconds) causes a thread to enter the timed waiting state for the specified duration.
Use case: Adding delays between consecutive actions or waiting for the completion of other threads before proceeding.
Terminated state:
Example: When the run () method finishes its execution or when the stop() method is called on the Thread.
Use case: Completing a task or explicitly stopping a thread's execution.

These examples demonstrate how threads can handle various scenarios in Java applications, allowing for concurrent and parallel execution of tasks. Understanding the different thread states helps in designing efficient and responsive multithreaded applications. Moreover, access modifiers in Java are also important to understand. Since these are used in multithreading programming, knowledge of them is important.

Q.2 What are the benefits of Multithreading?

Responsiveness
Since a process has multiple threads, so different threads do different tasks.
To understand how multithreading affects responsiveness, let us take an example. Consider a process that includes task A, task B, and task C. Thread 1 is performing task A, thread 2 is performing task B, and thread 3 is performing task C. If for some reason, thread 2 is blocked or waiting for a long time, in such a case, multithreading helps increase responsiveness. This is achieved since thread 1, and thread 3 can continue doing their work, increasing responsiveness.

Faster context switching
Context switching is a process that allowes the threads to be switched from the CPU so that another thread could be executed and save the state of the current thread so that the current thread can be continued from where it left.
The context switching time is less between two threads than between two processes. This is the reason why multithreading enables faster context switching.

Resource sharing
The threads of a process share the memory and the resources of that process among themselves. The code, data, files, and other operating system resources are the same for the threads of a process. When there is sharing of code and data, then within the same address space, the application can have multiple threads of activity.
Consider the example of a text editor that auto-corrects the spelling while one is typing. It also indents the line properly. Multiple threads share the same doc file. Hence they share the same memory and work on a common file

Cost-effective

The jobs of allocating memory and other resources for process creation are costly in terms of space and time. Every process requires its own memory. However, in the case of multithreading, since the threads share the resources(code, data, etc.) of a process, it is very cost-effective to create multiple threads. Multithreading is comparatively faster than creating multiple processes. Also, the resources required are less. These two reasons encourage one to go for multithreading.

Easy communication

Since the threads share the resources of a process, they can communicate with each other very fast.

Proper utilization of multiprocessor architecture

Let us consider a situation where we have four CPUs (processors). If we have a process with a single thread, we can only use one of the CPUs without utilizing the other three.

Economy

Multithreading allows for the efficient use of resources, enabling multiple tasks to share the same resources (such as CPU time and memory) without the need for separate processes. This results in reduced overhead and costs associated with managing multiple independent processes.

Scalability

Multithreading enhances scalability by allowing a program to take advantage of multiple CPU cores or processors. Tasks can be divided into threads that can execute concurrently, enabling better utilization of available hardware resources and improving overall system performance as workload increases.

Utilizing Microprocessor Architecture

Multithreading exploits the parallelism inherent in modern microprocessor architectures. By dividing tasks into smaller threads, the processor can execute multiple threads simultaneously or switch between them rapidly, maximizing throughput and efficiency.

Minimized System Resource Usage

Multithreading minimizes system resource usage by allowing threads to share resources such as memory and I/O devices. This reduces duplication of resources and overhead associated with creating and managing separate processes for each task, leading to more efficient resource utilization.

Concurrency Enhancement

Multithreading enhances concurrency by allowing multiple threads to execute concurrently within the same process. This enables tasks to overlap in execution, leading to improved responsiveness, better throughput, and enhanced overall system performance.

Minimized Time for Context Switching

Context switching refers to the process of saving and restoring the state of a thread or process when switching between tasks. Multithreading minimizes the time required for context switching compared to switching between separate processes, as threads within the same process share the same memory space and can switch more quickly. This results in reduced overhead and improved system responsiveness.

Q.3 What are the differences between Multithreading and Multitasking?

S.NO	Multitasking	Multithreading
1.	In multitasking, users are allowed to perform many tasks by CPU.	While in multithreading, many threads are created from a process through which computer power is increased.
2.	Multitasking involves often CPU switching between the tasks.	While in multithreading also, CPU switching is often involved between the threads.
3.	In multitasking, the processes share separate memory.	While in multithreading, processes are allocated the same memory.
4.	The multitasking component involves multiprocessing.	While the multithreading component does not involve multiprocessing.

5.	In multitasking, the CPU is provided in order to execute many tasks at a time.	While in multithreading also, a CPU is provided in order to execute many threads from a process at a time.
6.	In multitasking, processes don't share the same resources, each process is allocated separate resources.	While in multithreading, each process shares the same resources.
7.	Multitasking is slow compared to multithreading.	While multithreading is faster.
8.	In multitasking, termination of a process takes more time.	While in multithreading, termination of thread takes less time.
9.	Isolation and memory protection exist in multitasking.	Isolation and memory protection does not exist in multithreading.
10.	It helps in developing efficient programs.	It helps in developing efficient operating systems.
11.	Involves running multiple independent processes or tasks	Involves dividing a single process into multiple threads that can execute concurrently
12.	Multiple processes or tasks run simultaneously, sharing the same processor and resources	Multiple threads within a single process share the same memory space and resources
13.	Each process or task has its own memory space and resources	Threads share the same memory space and resources of the parent process
14.	Used to manage multiple processes and improve system efficiency	Used to manage multiple processes and improve system efficiency
15.	Examples: running multiple applications on a computer, running multiple servers on a network	Examples: splitting a video encoding task into multiple threads, implementing a responsive user interface in an application

Q.4 Define thread in java?

A thread in Java is the direction or path that is taken while a program is being executed. Generally, all the programs have at least one thread, known as the main thread, that is provided by the JVM or Java Virtual Machine at the starting of the program's execution. At this point, when the main thread is provided, the main () method is invoked by the main thread.

Thread in Java enables concurrent execution, dividing tasks for improved performance. It's essential for handling operations like I/O and network communication efficiently. Understanding threads is crucial for responsive Java applications.

Q.5 List the states of thread life cycle.

1. New State
2. Runnable State
3. Blocked State
4. Waiting State
5. Timed Waiting State
6. Terminated State

Q.6 Explain different methods supported by Thread class in java

Q.7 Briefly explain the advantages about synchronization in thread.

Data Consistency: Synchronization ensures that shared data is accessed in a controlled manner, preventing data corruption. This is crucial when multiple threads attempt to read and write shared variables.
Avoiding Race Conditions: By controlling the order of execution, synchronization helps avoid race conditions, where the outcome depends on the sequence or timing of uncontrollable events.

Atomicity: Synchronization ensures that complex operations on shared data are completed without interruption, maintaining atomicity and preventing partial updates that could lead to inconsistent states.
Coordination Between Threads: It allows threads to coordinate with each other, ensuring that resources are accessed in a well-defined sequence, which is particularly important in producer-consumer scenarios or when dealing with finite resources.
Deadlock Prevention: Proper synchronization mechanisms can help design systems that avoid deadlocks, where two or more threads are waiting indefinitely for resources held by each other.
Fair Resource Allocation: Synchronization mechanisms like semaphores and monitors can help ensure that resources are allocated fairly among threads, preventing resource starvation.

Q.8 With a simple program explain the multithreading concept in Java.

Multithreading in Java is a process of executing multiple threads simultaneously.
A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.
However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
Java Multithreading is mostly used in games, animation, etc.

```
class Hello extends Thread {
   public void run() {
      for(int i=1;i<=200;i++) {
         System.out.println("Hello");
      }
   }
}
class Hi extends Thread{
   public void run() {
      for(int i=1;i<=200;i++) {
         System.out.println("Hi");
      }
   }
}
public class Main {
   public static void main(String[] args) {
      Hello t1 = new Hello();
      Hi t2 = new Hi();
      t1.start();
      t2.start();
   }
}
```

OUTPUT

```
Hello
Hello
Hi
Hi
Hi
Hi
Hi
Hi
Hello
Hello
Hello
Hello
```

Q.9 Explain the creation of Thread with an example program.

Q.10 What is thread priority?

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled. You can get and set the priority of a Thread. Thread class provides methods and constants for working with the priorities of a Thread. Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

UNIT-4

Q.1 Write a java program to implement multilevel inheritance with 2 levels of hierarchy.
Q.2 What is single level inheritance? Explain with suitable example.

Single inheritance is the simplest type of inheritance in java. In this, a class inherits the properties from a single class. The class which inherits is called the derived class or child class or subclass, while the class from which the derived class inherits is called the base class or superclass or parent class. So, in single inheritance, we have only one derived class and one base class.

Example of Single Level Inheritance
In this example, we have two classes:
Animal: The base class.
Dog: The derived class that inherits from `Animal`.

Q.3 What is package? State how to create and access user defined package in Java.

Packages in Java are a mechanism to encapsulate a group of classes, interfaces, and sub-packages. In Java, it is used for making search/locating and usage of classes, interfaces, enumerations, and annotations easier. It can be considered data encapsulation also. In other words, we can say a package is a container of a group of related classes where some of the classes are accessible are exposed, and others are kept for internal purposes.

Creating and Accessing a User-Defined Package in Java

Step 1: Create a Package

Create the Package:

- Choose a directory structure for your package. For example, let's create a package named `mypackage`.
- Create a directory named **`mypackage`** to hold your classes.

Write Classes in the Package:

- Inside the **`mypackage`** directory, create a Java class file. For example, **`MyClass.java`**. Java

```
// File: mypackage/MyClass.java

package mypackage;


public class MyClass {
    public void displayMessage() {
        System.out.println("Hello from MyClass in mypackage");
    }
}
```

In the above code:

- The **`package`** statement at the top declares that **`MyClass`** is part of the **`mypackage`** package.

Step 2: Compile the Package

- Open a terminal or command prompt.

- Navigate to the directory containing the `mypackage` directory.
- Compile the `MyClass.java` file:

javac mypackage/MyClass.java

This will create a `MyClass.class` file inside the `mypackage` directory.

Step 3: Use the Package

1. **Create a Class to Use the Package**:
- In a directory outside of `mypackage`, create another Java file to use the class `MyClass`. For example, `Main.java`.

```
// File: Main.java

import mypackage.MyClass;


public class Main {
    public static void main(String[] args) {
        MyClass myObject = new MyClass();
        myObject.displayMessage();
    }
}
```

In the above code:

- The `import` statement is used to bring the `mypackage.MyClass` class into the `Main` class so it can be used.

2. Compile and Run the Program:

- Ensure you are in the directory containing `Main.java`.
- Compile `Main.java`:

sh

javac Main.java

- Run the compiled `Main` class:

sh

```
java Main
```

You should see the output:

Hello from MyClass in mypackage

Summary:
Package: A namespace for organizing classes and interfaces in a logical manner.
Creating a Package: Use the `package` keyword at the top of your Java file.
Compiling a Package: Use `javac` to compile your package.
Using a Package: Use the `import` statement to import your user-defined package into other classes.

Q.4 What is meant by interface? State its need and write syntax and features of interface

The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not the method body. It is used to achieve abstraction and multiple inheritances in Java using Interface. In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body. Java Interface also **represents the IS-A relationship**.

❖ Need for Interfaces

Multiple Inheritance: Java does not support multiple inheritance with classes. Interfaces allow a class to inherit the type of multiple interfaces, providing a form of multiple inheritance.
Abstraction: Interfaces provide a way to achieve abstraction. They hide the implementation details and only expose the method signatures to the user.
Loose Coupling: Interfaces help in reducing the dependency of a class on other classes. This leads to better modularity and easier maintenance.
Standardization: Interfaces allow the definition of methods that multiple classes can implement, ensuring a standard way of interacting with these classes.

❖ Syntax for Java Interfaces

```
// Interface declaration
        public interface MyInterface {
// Constant declarations (implicitly public, static, and final)
         int CONSTANT = 100;
// Abstract method declaration (implicitly public and abstract)
        void myMethod();
 // Default method (with implementation)
          default void defaultMethod() {
           System.out.println("This is a default method");
          }
// Static method (with implementation)
   static void staticMethod() {
      System.out.println("This is a static method");
   }
}
```

❖ Features of Interfaces

1. Implicit Modifiers:
All fields in an interface are implicitly **`public`**, **`static`**, and **`final`**.
All methods in an interface are implicitly **`public`** and **`abstract`** (except for default and static methods).
2. Abstract Methods: An interface primarily contains abstract methods which must be implemented by the classes that use the interface.
3. Default Methods: Interfaces can have default methods with implementations. These methods can be overridden by implementing classes if needed.
4. Static Methods: Interfaces can have static methods with implementations, which can be called independently of any object.
5. Inheritance: An interface can extend multiple interfaces. This allows for a form of multiple inheritance.
6. No Constructor: Interfaces cannot have constructors as they cannot be instantiated directly.

Q.5 List any four built-in packages from Java

java.sql: Provides the classes for accessing and processing data stored in a database. Classes like Connection, DriverManager, PreparedStatement, ResultSet, Statement, etc. are part of this package.
java.lang: Contains classes and interfaces that are fundamental to the design of the Java programming language. Classes like String, StringBuffer, System, Math, Integer, etc. are part of this package.
java.util: Contains the collections framework, some internationalization support classes, properties, random number generation classes. Classes like ArrayList, LinkedList, HashMap, Calendar, Date, Time Zone, etc. are part of this package.
java.net: Provides classes for implementing networking applications. Classes like Authenticator, HTTP Cookie, Socket, URL, URLConnection, URLEncoder, URLDecoder, etc. are part of this package.

Q.6 Explain inheritance and polymorphism features of Java.

- **Inheritance**

Inheritance is a fundamental concept in object-oriented programming that allows a class to inherit properties and behaviors (fields and methods) from another class. The class that is inherited from is called the **superclass** or **parent class**, and the class that inherits is called the **subclass** or **child class**.

Key Points:
Reusability: Inheritance promotes code reusability. Instead of writing the same code multiple times, you can define it once in a superclass and inherit it in multiple subclasses.
Method Overriding: Subclasses can override methods of the superclass to provide specific implementations.
Single Inheritance: Java supports single inheritance, meaning a class can inherit from only one superclass. However, a class can implement multiple interfaces, providing a way to achieve multiple inheritance.

- **Polymorphism**

The word polymorphism means having many forms. In simple words, we can define Java Polymorphism as the ability of a message to be displayed in more than one form. In this article, we will learn what is polymorphism and it's type.
Real-life Illustration of Polymorphism in Java:A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, and an employee. So the same person possesses different behaviors in different situations. This is called polymorphism.

Compile-time Polymorphism (Method Overloading): Achieved by having multiple methods with the same name but different parameters (different type, number, or both).
Runtime Polymorphism (Method Overriding): Achieved through inheritance where a subclass provides a specific implementation of a method that is already defined in its superclass.

Q.7 Explain method overriding with suitable example

Method overriding is a feature in Java that allows a subclass to provide a specific implementation of a method that is already defined in its superclass. When a method in a subclass has the same name, return type, and parameters as a method in its superclass, the method in the subclass overrides the method in the superclass.

```
class Animal {
  public void displayInfo() {
    System.out.println("I am an animal.");
  }
```

Output: `I am a dog.`

```
}

class Dog extends Animal {
  @Override
  public void displayInfo() {
    System.out.println("I am a dog.");
  }
}

class Main {
  public static void main(String[] args) {
    Dog d1 = new Dog();
    d1.displayInfo();
  }
}
```

Q.8 Give the difference between method overloading & method overriding.

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.
In method overloading, the return type can or can not be the same, but we just have to change the parameter.	In method overriding, the return type must 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.

Q.9 What is importance of super and this keyword in inheritance? Illustrate with suitable example

In Java, the **super** and **this** keywords are essential for managing inheritance and object-oriented design. These keywords help in distinguishing between members of the current class and the superclass, and provide a way to call constructors and methods of the superclass.

'This' Keyword

The **this** keyword refers to the current instance of the class. It is used for:

1. Differentiating Instance Variables from Parameters: When instance variables and parameters have the same name, **this** is used to refer to the instance variable.
2. Calling Other Constructors: The **this** keyword can be used to call another constructor within the same class.

EX:-

```
class Student {
   private String name;
   private int age;

   // Constructor using this to differentiate instance variables from parameters
   public Student(String name, int age) {
      this.name = name;
      this.age = age;
   }

   // Method to display student details
   public void display() {
      System.out.println("Name: " + this.name);
      System.out.println("Age: " + this.age);
   }
}

public class ThisKeywordExample {
   public static void main(String[] args) {
      Student student = new Student("Alice", 20);
      student.display();
   }
}
```

'Super' Keyword

The **super** keyword refers to the immediate superclass of the current class. It is used for:

1. Accessing Superclass Methods and Fields: **super** can be used to call methods and access fields of the superclass.
2. Calling Superclass Constructors: The **super** keyword can be used to call the constructor of the superclass.

```
// Superclass
class Animal {
   protected String name;
   public Animal(String name) {
      this.name = name;
   }
```

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

// Subclass
class Dog extends Animal {
    public Dog(String name) {
 super(name); // Calling superclass constructor
    }
    @Override
    public void sound() {
        super.sound(); // Calling superclass method
        System.out.println("Dog barks");
    }
    public void display() {
        System.out.println("Name: " + super.name); // Accessing superclass field
    }
}
public class SuperKeywordExample {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
        dog.sound();  // Calls the overridden method
        dog.display();
    }
}
```

Q.10 Write java program to overload the addition method.

```
public class AdditionOverload {

    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Method to add three integers
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method to add two double values
    public double add(double a, double b) {
        return a + b;
    }

    // Method to add an array of integers
    public int add(int[] numbers) {
        int sum = 0;
        for (int num : numbers) {
            sum += num;
        }
        return sum;
```

```
    }

    public static void main(String[] args) {
        AdditionOverload addition = new AdditionOverload();

        // Adding two integers
        System.out.println("Sum of 10 and 20: " + addition.add(10, 20));

        // Adding three integers
        System.out.println("Sum of 10, 20, and 30: " + addition.add(10, 20, 30));

        // Adding two double values
        System.out.println("Sum of 10.5 and 20.5: " + addition.add(10.5, 20.5));

        // Adding an array of integers
        int[] numbers = {1, 2, 3, 4, 5};
        System.out.println("Sum of array {1, 2, 3, 4, 5}: " + addition.add(numbers));
    }
}
```

Explanation

- **Method Overloading**:

The **`add`** method is overloaded to handle different numbers and types of parameters.
There are four versions of the **`add`** method:
`add(int a, int b)`: Adds two integers.
`add(int a, int b, int c)`: Adds three integers.
`add(double a, double b)`: Adds two double values.
`add(int[] numbers)`: Adds all integers in an array.

- **Main Method**:

The **`main`** method creates an instance of the **`AdditionOverload`** class and calls the overloaded **`add`** methods with different parameters.
It prints the results of each addition operation to the console.

OUTPUT

```
Sum of 10 and 20: 30
Sum of 10, 20, and 30: 60
Sum of 10.5 and 20.5: 31.0
Sum of array {1, 2, 3, 4, 5}: 15
```

UNIT-3

Q.1 List the various syntax of creating array in java

Q.2 Define the array in java

In Java, an array is an object that stores a fixed-size sequential collection of elements of the same type. It is a data structure that allows you to store multiple values of the same data type under a single name. Arrays are widely used because they provide a convenient way to manage collections of data.

Q.3 Write java program to display array elements using for loop.

```
public class DisplayArrayElements {
   public static void main(String[] args) {
      // Array declaration and initialization
      int[] numbers = {1, 2, 3, 4, 5};

      // Displaying array elements using a for loop
      System.out.println("Array elements:");
      for (int i = 0; i < numbers.length; i++) {
         System.out.println(numbers[i]);
      }
   }
}
```

OUTPUT

```
Array elements:
1
2
3
4
5
```

Q.4 What is Constructor in Java

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 the constructor, memory for the object is allocated in the memory. It is a special type of method that is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called.

Q.5 Describe the structure of Employee class in java

The structure of an Employee class in Java typically includes instance variables (also known as fields), constructors, getter and setter methods, and other utility methods. Below is an example of how an Employee class might be structured:

```
public class Employee {
   // Instance variables
   private String name;
   private int age;
   private double salary;

   // Constructors
   public Employee() {
      // Default constructor
   }

   public Employee(String name, int age, double salary) {
      this.name = name;
      this.age = age;
      this.salary = salary;
   }

   // Getter and setter methods
   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;
  }

  public double getSalary() {
    return salary;
  }

  public void setSalary(double salary) {
    this.salary = salary;
  }

  // Utility methods
  public void displayDetails() {
    System.out.println("Name: " + name);
    System.out.println("Age: " + age);
    System.out.println("Salary: $" + salary);
  }
}
```

Q.6 Write java program to perform addition of two matrix.

```
public class MatrixAddition {
  public static void main(String[] args) {
    int[][] matrix1 = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int[][] matrix2 = {{9, 8, 7}, {6, 5, 4}, {3, 2, 1}};

    int rows = matrix1.length;
    int columns = matrix1[0].length;

    int[][] sumMatrix = new int[rows][columns];

    // Performing matrix addition
    for (int i = 0; i < rows; i++) {
      for (int j = 0; j < columns; j++) {
        sumMatrix[i][j] = matrix1[i][j] + matrix2[i][j];
      }
    }
    // Displaying the result matrix
    System.out.println("Result of Matrix Addition:");
    for (int i = 0; i < rows; i++) {
      for (int j = 0; j < columns; j++) {
        System.out.print(sumMatrix[i][j] + " ");
      }
      System.out.println();}
} }
```

OUTPUT

```
Result of Matrix Addition:
10 10 10
10 10 10
10 10 10
```

Q.7 Explain method overloading in java with example.

Method overloading in Java allows a class to have multiple methods with the same name but with different parameters. It enables the programmer to define multiple methods with the same name in a class, differentiating them by the number or type of parameters. When a method is called, the Java compiler determines which method to execute based on the number and type of arguments passed to it.

```
public class Calculator {
  // Method to add two integers
  public int add(int a, int b) {
    return a + b;
  }

  // Overloaded method to add three integers
  public int add(int a, int b, int c) {
    return a + b + c;
  }

  // Overloaded method to add two double values
  public double add(double a, double b) {
    return a + b;
  }

  // Overloaded method to concatenate two strings
  public String add(String str1, String str2) {
    return str1 + str2;
  }

  public static void main(String[] args) {
    Calculator calculator = new Calculator();

    // Calling overloaded methods
    System.out.println("Sum of 5 and 3: " + calculator.add(5, 3));
    System.out.println("Sum of 5, 3, and 2: " + calculator.add(5, 3, 2));
    System.out.println("Sum of 5.5 and 3.5: " + calculator.add(5.5, 3.5));
    System.out.println("Concatenation of 'Hello' and 'World': " + calculator.add("Hello", "World"));
  }
}
```

OUTPUT

```
Sum of 5 and 3: 8
Sum of 5, 3, and 2: 10
Sum of 5.5 and 3.5: 9.0
Concatenation of 'Hello' and 'World':
HelloWorld
```

In this example:

- The `Calculator` class contains multiple overloaded `add` methods, each with a different parameter list.
- The methods perform addition of integers, addition of doubles, and concatenation of strings.
- The `main` method demonstrates calling these overloaded methods with different types and numbers of arguments.
- Depending on the arguments passed, the compiler resolves the method calls to the appropriate overloaded method at compile-time.

Q.8 How to create object of Student class using new keyword and call default constructor.

To create an object of the `Student` class using the `new` keyword and call the default constructor, you simply need to use the following syntax:

```
Student student = new Student();
```

Assuming you have a `Student` class with a default constructor defined like this:

```
public class Student {
   // Default constructor
   public Student() {
      // Constructor logic goes here
   }
}
```

Here's how you would create an object of the `Student` class using the `new` keyword and call the default constructor:

```
public class Main {
   public static void main(String[] args) {
      // Creating an object of Student class using new keyword
      // and calling the default constructor
      Student student = new Student();

      // Now you can use the student object to access its methods or fields
      // or perform any other operations you need
   }
}
```

This code creates a new instance of the `Student` class named `student` using the `new` keyword and calls its default constructor. After creating the object, you can perform any desired operations on the `student` object, such as accessing its methods or fields.

Q.9 Write java program to perform constructor overloading in java.

```
public class Student {
   // Instance variables
   private String name;
   private int age;
   private String major;

   // Default constructor
   public Student() {
      this.name = "Unknown";
      this.age = 0;
      this.major = "Undeclared";
   }

   // Constructor with name parameter
   public Student(String name) {
      this.name = name;
      this.age = 0;
      this.major = "Undeclared";
   }

   // Constructor with name and age parameters
   public Student(String name, int age) {
```

```
        this.name = name;
        this.age = age;
        this.major = "Undeclared";
    }

    // Constructor with all parameters
    public Student(String name, int age, String major) {
        this.name = name;
        this.age = age;
        this.major = major;
    }

    // Method to display student details
    public void displayDetails() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Major: " + major);
    }

    public static void main(String[] args) {
        // Creating objects using different constructors
        Student student1 = new Student();
        Student student2 = new Student("ADARSH",20,"IBM");
        Student student3 = new Student("HARSH", 20,"MICROSOFT");
        Student student4 = new Student("KALPESH", 20, "TCS");
        Student student5 = new Student("SIDDHARTH",20,"TWITTER");
        Student student6 = new Student("DILIP",20,"GOOGLE");

        // Displaying details of each student
        System.out.println("Student 1:");
        student1.displayDetails();

        System.out.println("\nStudent 2:");
        student2.displayDetails();

        System.out.println("\nStudent 3:");
        student3.displayDetails();

        System.out.println("\nStudent 4:");
        student4.displayDetails();

        System.out.println("\nStudent 5:");
        student5.displayDetails();

        System.out.println("\nStudent 6:");
        student6.displayDetails();

    }
}
```

OUTPUT

```
Student 1:
Name: Unknown
Age: 0
Major: Undeclared

Student 2:
Name: ADARSH
Age: 20
Major: IBM

Student 3:
Name: HARSH
Age: 20
Major: MICROSOFT

Student 4:
Name: KALPESH
Age: 20
Major: TCS

Student 5:
Name: SIDDHARTH
Age: 20
Major: TWITTER

Student 6:
Name: DILIP
Age: 20
Major: GOOGLE
```

Q.10 Give the importance of 'this' keyword for changing the scope of the variables.

In Java, the **`this`** keyword is a reference to the current object within an instance method or a constructor. It is primarily used to eliminate the ambiguity between instance variables (fields) and parameters or local variables when they have the same names. By using **`this`**, you can explicitly specify that you are referring to the instance variables of the current object.

Importance of '`this`' Keyword

1.Differentiating Instance Variables from Parameters: When constructor or method parameters have the same names as instance variables, the **`this`** keyword helps distinguish between them.

2. Accessing Instance Variables and Methods: It is used to access the instance variables and methods of the current object.

3. Chaining Constructors: It is used to call one constructor from another within the same class (constructor chaining).

4. Returning the Current Object: It can be used to return the current object from a method.

5. Passing the Current Object as an Argument: It can be used to pass the current object as an argument to another method.

UNIT-2

Q.1 List the data types in java.

Data Type	Size (bits)	Min Value	Max Value	Default Value
byte	8	-128	127	0
short	16	-32,768	32,767	0
int	32	-2^31	2^31 - 1	0
long	64	-2^63	2^63 - 1	0L
float	32	±1.4E-45	±3.4E+38	0.0f
double	64	±4.9E-324	±1.7E+308	0.0d
boolean	1	false	true	false
char	16	'\u0000' (or 0)	'\uffff' (or 65,535 inclusive)	'\u0000' (or 0)

Q.2 Explain Relational & Logical operator in java.

- **Relational Operators in Java**

Java relational operators are assigned to check the relationships between two particular operators. There are various relational operators in Java, such as

Operators	Description	Example
==	Is equal to	3 == 5 returns false
!=	Not equal to	3 != 5 returns true
>	Greater than	3 > 5 returns false
<	Less than	3 < 5 returns true
>=	Greater than or equal to	3 >= 5 returns false
<=	Less than or equal to	3 <= 5 returns true

- **Logical Operators in Java**

Logical Operators in Java check whether the expression is true or false. It is generally used for making any decisions in Java programming. Not only that but Jump statements in Java are also used for checking whether the expression is true or false. It is generally used for making any decisions in Java programming.

Operators	Example	Meaning
&& [logical AND]	expression1 && expression2	(true) only if both of the expressions are true
|| [logical OR]	expression1 || expression2	(true) if one of the expressions in true
! [logical NOT]	!expression	(true) if the expression is false and vice-versa

Q.3 What are the various decision making constructs in java

Sr.No.	Statement & Description
1	**if statement** An if statement consists of a boolean expression followed by one or more statements.
2	**if...else statement** An if statement can be followed by an optional else statement, which executes when the boolean expression is false.
3	**nested if statement**

	You can use one if or else if statement inside another if or else if statement(s).
4	switch statement A switch statement allows a variable to be tested for equality against a list of values.

Q.4 Explain structure of for loop with suitable example.

A **`for`** loop in Java is used to execute a block of code a specific number of times. It is particularly useful when the number of iterations is known beforehand. The structure of a **`for`** loop consists of three main components: initialization, condition, and update.

Structure of a `for` Loop

```
for (initialization; condition; update) {
   // Code to be executed
}
```

Initialization: This step is executed once before the loop starts. It is used to initialize the loop control variable(s).

Condition: This is a boolean expression that is checked before each iteration. If the condition is true, the loop body is executed. If it is false, the loop terminates.

Update: This step is executed after each iteration of the loop body. It is typically used to update the loop control variable(s).

Q.5 Define the implicit & explicit type casting java

There are two main types of type casting: implicit (or automatic) and explicit (or manual).

Implicit Type Casting (Automatic Type Conversion)

Implicit type casting, also known as widening conversion, occurs when a smaller data type is automatically converted to a larger data type. This type of casting is done by the Java compiler and does not require explicit code from the programmer. Implicit type casting is safe and does not result in data loss.

Explicit Type Casting (Manual Type Conversion)

Explicit type casting, also known as narrowing conversion, occurs when a larger data type is explicitly converted to a smaller data type. This type of casting must be performed manually by the programmer and can result in data loss or precision loss if the target type cannot fully accommodate the value being cast.

Key Points to Remember

Implicit Type Casting:

1. Automatically performed by the Java compiler.
2. No data loss occurs.
3. Example: **`int`** to **`double`**.

Explicit Type Casting:

1. Must be explicitly performed by the programmer.
2. Can result in data loss or precision loss.
3. Example: **`double`** to **`int`**

Q.6 List the various keywords in java.

Keywords in Java

1. **Abstract**: Indicates that a class or method is abstract.

2. **Assert**: Used for debugging purposes.
3. **Boolean**: A data type that can hold `true` or `false`.
4. **Break**: Exits from a loop or a switch statement.
5. **Byte**: A data type that holds an 8-bit integer.
6. **Case**: Defines a group of statements to execute in a switch statement.
7. **Catch**: Catches exceptions generated by the try statements.
8. **Char**: A data type that holds a 16-bit Unicode character.
9. **Class**: Declares a class.
10. **Const**: Not used (reserved for future use).
11. **Continue**: Skips the current iteration of a loop.
12. **Default**: Specifies the default block of code in a switch statement.
13. **Do**: Used in control statements to declare a loop that will iterate a block of statements.
14. **Double**: A data type that holds a 64-bit floating-point number.
15. **Else**: Used in conditional statements.
16. **Enum**: Declares an enumerated type.
17. **Extends**: Indicates that a class is inherited from another class.
18. **Final**: Defines an entity once that cannot be changed or inherited later.
19. **Finally**: Used to execute code after try and catch blocks.
20. **Float**: A data type that holds a 32-bit floating-point number.
21. **For**: Declares a loop that iterates a block of statements.
22. **Goto**: Not used (reserved for future use).
23. **If**: Used to specify a condition.
24. **Implements**: Indicates that a class implements an interface.
25. **Import**: Brings other classes or entire packages into visibility.
26. **Instanceof**: Tests whether an object is an instance of a class.
27. **Int**: A data type that holds a 32-bit integer.
28. **Interface**: Declares an interface.
29. **Long**: A data type that holds a 64-bit integer.
30. **Native**: Specifies that a method is implemented in native code using JNI (Java Native Interface).
31. **New**: Creates new objects.
32. **Null**: Represents a null reference.
33. **Package**: Declares a package.
34. **Private**: An access modifier indicating private access.
35. **Protected**: An access modifier indicating protected access.
36. **Public**: An access modifier indicating public access.
37. **Return**: Exits from the current method and optionally returns a value.
38. **Short**: A data type that holds a 16-bit integer.
39. **Static**: Indicates that a field or method belongs to the class, rather than instances of the class.
40. **Strictfp**: Restricts floating-point calculations to ensure portability.
41. **Super**: Refers to the superclass of the current object.
42. **Switch**: A control statement that allows multiple possible execution paths.
43. **Synchronized**: Indicates that a method or block is synchronized.
44. **This**: Refers to the current instance of the class.
45. **Throw**: Throws an exception.
46. **Throws**: Indicates what exceptions may be thrown by a method.
47. **Transient**: Prevents serialization of fields.
48. **Try**: Starts a block of code that will be tested for exceptions.
49. **Void**: Specifies that a method does not return a value.
50. **Volatile**: Indicates that a field may be changed asynchronously.
51. **While**: Starts a while loop.

Q.7 Give the relationship between JDK, JRE and JVM

In the Java ecosystem, the Java Development Kit (JDK), Java Runtime Environment (JRE), and Java Virtual Machine (JVM) are three core components that play distinct roles. Understanding the relationship between these components is essential for Java development and execution.

Java Development Kit (JDK)

- **Purpose**: The JDK is a complete software development kit that provides all the tools necessary to develop, compile, debug, and run Java applications.
- **Components**:

1. **JRE**: The JDK includes the JRE, which means it has all the components needed to run Java programs.
2. **Development Tools**: Tools like `javac` (Java compiler), `javadoc` (documentation generator), `jdb` (debugger), and other tools required for Java development.
3. **Libraries**: Additional libraries required for development.

- **Usage**: It is used by developers to write and compile Java programs.

Java Runtime Environment (JRE)

- **Purpose**: The JRE provides the libraries, Java Virtual Machine (JVM), and other components to run applications written in Java.
- **Components**:

1. **JVM**: The core component that executes Java bytecode.
2. **Core Libraries**: Libraries and other files required for the JVM to run Java applications.

- **Usage**: It is used by end-users to run Java applications. It does not include development tools like compilers.

Java Virtual Machine (JVM)

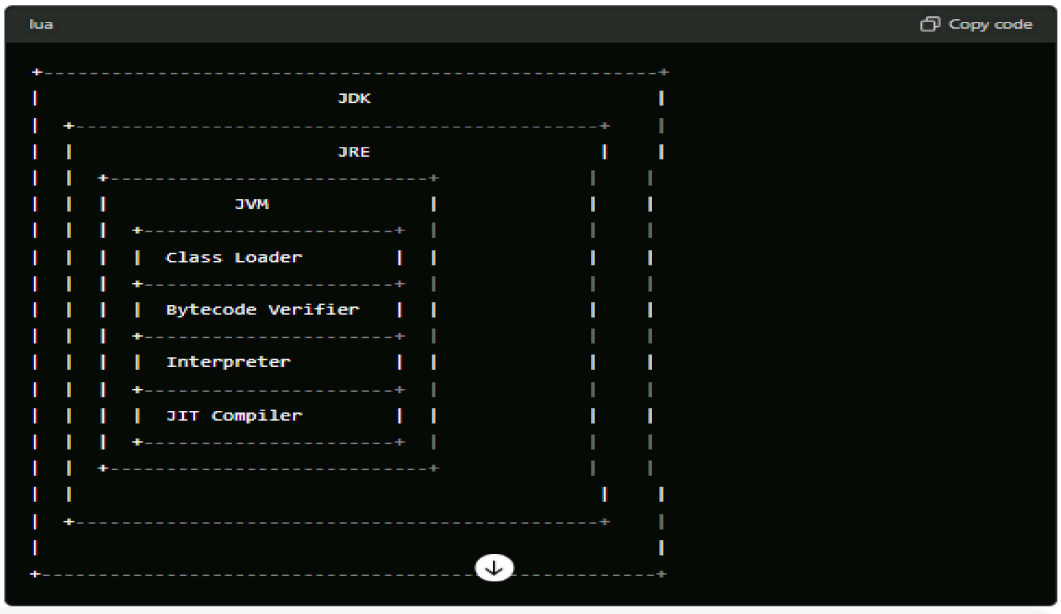
- **Purpose**: The JVM is an abstract computing machine that enables a computer to run Java programs. It is responsible for converting bytecode into machine-specific code and executing it.
- **Components**:

1. **Class Loader**: Loads class files into memory.
2. **Bytecode Verifier**: Ensures the bytecode adheres to the Java language specification and is safe to execute.
3. **Interpreter**: Executes the bytecode instructions.
4. **Just-In-Time (JIT) Compiler**: Converts bytecode into native machine code for performance optimization.

- **Usage**: It provides a runtime environment in which Java bytecode can be executed, making Java platform-independent.

Diagram of Relationship

lua Copy code



Q.8 Explain the various features of java programming language.

Java programming language is widely used for developing a variety of applications, from web and mobile applications to enterprise systems and embedded devices. It is known for its simplicity, robustness, platform independence, and security. Here are some key features of the Java programming language:

1. Platform Independence

Java programs are compiled into bytecode, which can be executed on any platform with a Java Virtual Machine (JVM).
Write once, run anywhere (WORA) principle allows Java applications to run on diverse platforms without modification.

2. Object-Oriented

Java is a pure object-oriented programming language.
Encapsulation, inheritance, polymorphism, and abstraction are core principles of Java's object-oriented design.

3. Simple and Easy to Learn

Java syntax is based on C and C++, making it familiar to many programmers.
Elimination of complex features such as pointers and operator overloading simplifies the language.

4. Robustness

Java's strong memory management, garbage collection, and exception handling make it robust.
Compile-time error checking and runtime checking of data types enhance reliability.

5. Security

Java's security features, such as bytecode verification, classloader, and security manager, protect against unauthorized access and malicious attacks.
Sandboxing allows execution of untrusted code in a restricted environment.

6. Multi-threading

Java provides built-in support for multi-threading, allowing concurrent execution of multiple threads within a program.
Thread management is simplified with the `java.lang.Thread` class and `java.util.concurrent` package.

7. High Performance

Java's Just-In-Time (JIT) compiler optimizes bytecode into native machine code at runtime, improving performance.
Java applications can achieve high performance with advanced profiling and tuning techniques.

8. Distributed Computing

Java's Remote Method Invocation (RMI) and Java Message Service (JMS) facilitate distributed computing.
Java EE (Enterprise Edition) provides APIs and frameworks for building scalable, distributed enterprise applications.

9. Rich Standard Library

Java Standard Edition (SE) comes with a comprehensive standard library (Java API) covering data structures, networking, I/O, utilities, and more.
Java EE extends the standard library with enterprise-specific APIs for web development, persistence, messaging, and transactions.

10. Dynamic and Extensible

Java's reflection API allows runtime inspection and manipulation of classes, interfaces, methods, and fields.
Java's modular system (introduced in Java 9) enables better organization, maintenance, and scalability of large-scale applications.

11. Community and Ecosystem

Java has a large and active community of developers, contributing to libraries, frameworks, and tools.
Java's ecosystem includes a wide range of third-party libraries, frameworks (e.g., Spring, Hibernate), and integrated development environments (IDEs) such as Eclipse, IntelliJ IDEA, and NetBeans.

Q.9 Write java program to perform factorial of given number [no taken from user by using Scanner class]

```
import java.util;
public class Main
{
   public static void main(String []args)
   {
     int i=1,fact=1;
     while(i<=5)
     {
        fact=fact*i;
        i++;
     }
     System.out.println("Factorial of the number: "+fact);
   }
}
```

OUTPUT

Factorial of the number: 120

Q.10 Explain the structure of java program.

1. Package Declaration (Optional)

- If your Java program is organized into packages, you can start with a `package` declaration.
- This statement specifies the package to which the current Java file belongs.

2. Import Statements (Optional)

- Import statements are used to include classes or entire packages from other packages.
- They help in avoiding fully qualified class names throughout the program.

3. Class Declaration

- Every Java program must have at least one class.
- The class declaration is the blueprint for creating objects of that class.
- The main method (`public static void main(String[] args)`) is typically included in the class declaration.

4. Main Method

- The `main` method is the entry point of a Java program.
- It is called by the Java Virtual Machine (JVM) to execute the program.
- The `main` method has a specific signature: `public static void main(String[] args)`.
- It can accept command-line arguments as an array of strings (`args`).

5. Program Logic

- This section contains the actual code that performs the desired functionality of the program.
- It can include variable declarations, method definitions, control flow statements (if-else, loops), etc.

6. Comments

- Comments are used to document the code for better understanding.
- They can be single-line (`//`) or multi-line (`/* */`).
- Comments are ignored by the compiler and are not executed.

UNIT-1

Q.1 Give the difference between C++ and JAVA

Parameters	Java	C++
Official Website	oracle.com/java	isocpp.org
Influenced By	Java was Influenced by Ada 83, Pascal, C++, C#, etc. languages.	C++ was Influenced by Influenced by Ada, ALGOL 68, C, ML, Simula, Smalltalk, etc. languages.
Influenced to	Java was influenced to develop BeanShell, C#, Clojure, Groovy, Hack, J#, Kotlin, PHP, Python, Scala, etc. languages.	C++ was influenced to develop C99, Java, JS++, Lua, Perl, PHP, Python, Rust, Seed7, etc. languages.
Platform Dependency	Platform-independent, Java bytecode works on any operating system.	Platform dependent should be compiled for different platforms.
Portability	It can run on any OS hence it is portable.	C++ is platform-dependent. Hence it is not portable.
Compilation	Java is both a Compiled and Interpreted Language.	C++ is a Compiled Language.
Memory Management	Memory Management is System Controlled.	Memory Management in C++ is Manual.
Virtual Keyword	It doesn't have Virtual keywords.	It has Virtual keywords.
Overloading	It supports only method overloading and doesn't allow operator overloading.	It supports both method and operator overloading.
Pointers	It has limited support for pointers.	It strongly supports pointers.
Libraries	Libraries have a wide range of classes for various high-level services.	C++ libraries have comparatively low-level functionalities.
Documentation Comment	It supports documentation comments (e.g., /**.. */) for source code.	It doesn't support documentation comments for source code.
Thread Support	Java provides built-in support for multithreading.	C++ doesn't have built-in support for threads, depends on third-party threading libraries.
Type	Java is only an object-oriented programming language.	C++ is both a procedural and an object-oriented programming language.
Input-Output mechanism	Java uses the (System class): **System.in** for input and **System.out** for output.	C++ uses **cin** for input and **cout** for output operation.
goto Keyword	Java doesn't support the goto Keyword	C++ supports the goto keyword.
Structures and Unions	Java doesn't support Structures and Unions.	C++ supports Structures and Unions.
Parameter Passing	Java supports only the Pass by Value technique.	C++ supports both Pass by Value and pass-by-reference.
Global Scope	It supports no global scope.	It supports both global scope and namespace scope.
Call by Value and Call by Reference	Java supports only calls by value.	C++ both supports call by value and call by reference.
Hardware	Java is not so interactive with hardware.	C++ is nearer to hardware

Q.2 What is object? Give one example of object with its properties & methods in java.

In object-oriented programming (OOP), an object is a real-world entity that represents a specific instance of a class. It combines data (properties) and behavior (methods) into a single unit. Objects are instances of classes, and each object has its own state (values of properties) and behavior (methods to perform actions)

```
public class Car {

    // Properties
    private String brand;
    private String model;
    private int year;
    private double price;

    // Methods
    public void displayInfo() {
        System.out.println("Car Information:");
        System.out.println("Brand: " + brand);
        System.out.println("Model: " + model);
        System.out.println("Year: " + year);
        System.out.println("Price: $" + price);
    }

    public void start() {
        System.out.println("Starting the car...");
    }

    public void accelerate() {
        System.out.println("Accelerating the car...");
    }

    public void brake() {
        System.out.println("Applying brakes...");
    }
}


public class Main {
    public static void main(String[] args) {
        // Creating a car object
        Car myCar = new Car("Toyota", "Camry", 2020, 25000.00);

        // Accessing properties and calling methods
        myCar.displayInfo();  // Display car information
        myCar.start();        // Start the car
        myCar.accelerate();   // Accelerate the car
        myCar.brake();        // Apply brakes
    }
}
```

OUTPUT

```
Car Information:
Brand: Toyota
Model: Camry
Year: 2020
Price: $25000.0
Starting the car...
Accelerating the car...
Applying brakes...
```

Q.3 List the various access modifiers in java.

1. Default – No keyword required
2. Private
3. Protected
4. Public

Q.4 What is data abstraction Java.

Data Abstraction in Java is the process of reducing the object to its essence in order to only display the necessary characteristics to users. Abstraction defines an object in terms of attributes, methods and interfaces. In this article, we will be discussing the top interview questions based on data abstraction.

We can achieve data abstraction through Abstract classes and interfaces. Through abstract classes, you can achieve partial or complete abstraction as abstract classes contain methods that have an implementation which can result in partial abstraction. On the other hand, interfaces allow complete (100%) abstraction since these allow users to abstract the implementation completely. In Java, data abstraction is implemented through abstract keywords while declaring a class or a method.

Q.5 Write java program to display multiplication table of 4.

```
public class MultiplicationTable {
   public static void main(String[] args) {
      int number = 4; // The number for which we want to display the multiplication table
      int range = 10; // The range up to which we want the multiplication table

      System.out.println("Multiplication Table of " + number + ":");
      for (int i = 1; i <= range; i++) {
         System.out.println(number + " x " + i + " = " + (number * i));
      }
   }
}
```

OUTPUT

```
Multiplication Table of 4:
4 x 1 = 4
4 x 2 = 8
4 x 3 = 12
4 x 4 = 16
4 x 5 = 20
4 x 6 = 24
4 x 7 = 28
4 x 8 = 32
4 x 9 = 36
4 x 10 = 40
```

Q.6 Write java program to display given number is positive or negative.

```
import java.util.Scanner;

public class PositiveOrNegative {
   public static void main(String[] args) {
      // Create a Scanner object to read input from the user
      Scanner scanner = new Scanner(System.in);

      // Prompt the user to enter a number
      System.out.print("Enter a number: ");
      int number = scanner.nextInt();
      // Check if the number is positive, negative, or zero
      if (number > 0) {
         System.out.println("The number is positive.");
      } else if (number < 0) {
```

OUTPUT

If the user enters a positive number:
Enter a number: 5
The number is positive.
If the user enters a positive number:
Enter a number: -3
The number is negative.
If the user enters a positive number:
Enter a number: 0
The number is zero.

```
            System.out.println("The number is negative.");
        } else {
            System.out.println("The number is zero.");
        }

        // Close the scanner
        scanner.close();
    }
}
```

Q.7 How to create class in java? Explain various components of class

Creating a class in Java involves defining a new type of object that encapsulates data and methods to operate on that data. Here's how to create a class and an explanation of the various components of a class.

```
public class ClassName {
    // Fields (or attributes)
    // Constructors
    // Methods
}
```

Explanation of Components in Detail

1. **Class Declaration**:
- **Visibility Modifier** (`public`): Determines the visibility of the class. Other options include `private` (not allowed for top-level classes) and package-private (no modifier), meaning the class is accessible only within its own package.
- **Class Name** (`Car`): Should be a noun representing the entity you are modeling.
2. **Fields**:
- **Access Modifiers** (`private`): Control the visibility of the field. Other options include `public`, `protected`, and package-private (no modifier).
- **Data Types** (`String`, `int`): Specify the type of data the field will hold.
- **Field Names** (`brand`, `model`, `year`): Should be descriptive and follow camelCase naming conventions.
3. **Constructors**:
- **Constructor Name**: Must match the class name.
- **Parameters** (`String brand`, `String model`, `int year`): Allow passing values to initialize the object's fields.
- **Body**: Contains the code to initialize the fields.
4. **Methods**:
- **Access Modifiers**: Determine the visibility of the method.
- **Return Type** (`void`, or any other data type): Specifies what the method returns. `void` means the method does not return any value.
- **Method Name**: Should be a verb or verb phrase describing the action the method performs.
- **Parameters**: Define what data, if any, you can pass to the method.
- **Method Body**: Contains the code defining the method's behavior.

Q.8 Explain the various features about object oriented programming language.

1. **Encapsulation**

Encapsulation is the mechanism of wrapping data (variables) and code (methods) together as a single unit. In encapsulation, the variables of a class are hidden from other classes and can only be accessed through the methods of their current class. This is also known as data hiding.

2. **Inheritance**

Inheritance is a mechanism wherein a new class is derived from an existing class. The new class (child or subclass) inherits the attributes and methods of the existing class (parent or superclass), which helps in code reusability and hierarchical classification.

3. **Polymorphism**
Polymorphism means "many forms," and it allows methods to do different things based on the object it is acting upon. There are two types of polymorphism in Java: compile-time polymorphism (method overloading) and runtime polymorphism (method overriding).

4. **Abstraction**
Abstraction is the concept of hiding the complex implementation details and showing only the necessary features of an object. This can be achieved using abstract classes and interfaces.

5. **Classes and Objects**
A class is a blueprint for creating objects. An object is an instance of a class. It contains states (attributes) and behaviours (methods).

6. **Constructors**
Constructors are special methods that are called when an object is instantiated. They are used to initialize the object's state.

Q.9 Why java is not a fully object oriented programming language? Elaborate your answered.

Java is often described as a primarily object-oriented programming language, but it is not considered a fully object-oriented programming language. The primary reason for this is that Java includes some primitive data types that are not objects. Here's a detailed explanation of why Java is not fully object-oriented:

Primitive Data Types
Java includes eight primitive data types: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`. These data types are not objects and do not belong to any class. They are included in the language for efficiency purposes, as using objects for every piece of data can be costly in terms of memory and performance.
In contrast, a fully object-oriented language would treat everything as an object, without the need for primitive types.

Autoboxing and Unboxing
To bridge the gap between primitive types and objects, Java provides wrapper classes for each primitive type (e.g., `Integer` for `int`, `Double` for `double`). Autoboxing and unboxing allow automatic conversion between primitives and their corresponding wrapper objects.
Even with autoboxing, the existence of primitives means that Java cannot be considered fully object-oriented.

Static Methods and Variables
Java allows the use of static methods and static variables. These belong to the class itself rather than to any instance of the class, meaning they can be accessed without creating an object of the class. This concept deviates from the pure object-oriented paradigm where every method and variable should be part of an object.

Q.10 Explain the working of 'continue' and 'break' keywords in java.

The `continue` and `break` keywords in Java are control flow statements that are used to alter the flow of loop execution. They provide more control over how loops execute by allowing the program to skip iterations or terminate loops prematurely.

- **`continue` Keyword**

The `continue` statement skips the current iteration of a loop and proceeds to the next iteration. It can be used in `for`, `while`, and `do-while` loops. When the `continue` statement is encountered, the rest of the code inside the loop for the current iteration is skipped, and the loop proceeds with the next iteration.

```
public class ContinueExample {
   public static void main(String[] args) {
      for (int i = 1; i <= 10; i++) {
         if (i % 2 == 0) {
            continue; // Skip the rest of the loop body if i is even
         }
         System.out.println(i); // Print odd numbers
      }
   }
}
```

OUTPUT

```
1
3
5
7
9
```

- **`break` Keyword**

The `break` statement terminates the loop immediately and transfers control to the statement following the loop. It can be used in `for`, `while`, and `do-while` loops, as well as in `switch` statements. When the `break` statement is encountered, the loop stops executing and the program continues with the next statement outside the loop.

```
public class BreakExample {
   public static void main(String[] args) {
      for (int i = 1; i <= 10; i++) {
         if (i == 5) {
            break; // Terminate the loop if i is 5
         }
         System.out.println(i); // Print numbers until 4
      }
      System.out.println("Loop terminated.");
   }
}
```

OUTPUT

```
1
2
3
4
Loop terminated.
```

- `continue` skips the current iteration of a loop and proceeds to the next iteration.
- `break` terminates the loop immediately and continues with the next statement after the loop.