IMPORTANT QUESTIONS OF OOPs JAVA

IMPORTANT QUESTIONS OF OOPs JAVA

Q. Explain the concept of OOPs. Why we need Java?

Object-oriented programming (OOP) is a programming paradigm that is based on the concept of objects, which can contain data and code that operates on that data. OOP emphasizes the use of classes and objects to represent real-world entities and their relationships, making it a useful approach for modeling complex systems.

There are four key principles of OOP:

1. Encapsulation: The concept of encapsulation involves bundling data and methods that operate on that data into a single entity, known as a class. This allows for better control over access to data and helps to maintain data integrity.
2. Inheritance: Inheritance allows classes to inherit properties and behavior from other classes, allowing for code reuse and promoting modularity.
3. Polymorphism: Polymorphism refers to the ability of objects to take on multiple forms, which allows for more flexible and extensible code.
4. Abstraction: Abstraction involves focusing on the essential features of an object or system and ignoring the details that are not relevant to its function. This allows for better understanding and management of complex systems.

Java is a popular language for implementing OOP principles because it offers a number of features that make it well-suited for OOP development, including:

1. Strong support for encapsulation through access modifiers such as private, public, and protected.
2. Inheritance and polymorphism are supported through the use of classes and interfaces.
3. Java's syntax and class libraries provide a high level of abstraction, making it easier to work with complex systems.
4. Java's automatic memory management (garbage collection) and exception handling provide a robust and reliable runtime environment for OOP applications.

Overall, Java is a popular choice for OOP development due to its support for key OOP principles, as well as its portability, reliability, and ease of use.

Q. Discuss about abstraction, encapsulation and inheritance in java.

Ans:- Abstraction, encapsulation, and inheritance are three key concepts in object-oriented programming (OOP) in Java.

1. Abstraction: Abstraction is the process of hiding complex implementation details and focusing on the essential features of an object or system. In Java, abstraction is achieved through the use of abstract classes and interfaces. An abstract class is a class that cannot be instantiated and is used as a base class for other classes. An interface is a collection of abstract methods that define a set of behaviors that can be implemented by other classes.

For example, if we have a car class, we may want to define a method for driving the car, but we do not need to know the details of how the car engine works. Abstraction allows us to focus on the essential features of the car, such as the ability to drive it, without worrying about the underlying implementation details.

2. Encapsulation: Encapsulation is the process of bundling data and methods that operate on that data into a single entity, known as a class. Encapsulation is important for ensuring data integrity and controlling access to data. In Java, encapsulation is achieved through the use of access modifiers, such as private, public, and protected.

For example, if we have a car class, we may want to prevent direct access to the car's engine. Encapsulation allows us to define the engine as a private variable within the car class, and then provide public methods for accessing and modifying the engine.

3. Inheritance: Inheritance allows classes to inherit properties and behavior from other classes, allowing for code reuse and promoting modularity. In Java, inheritance is achieved through the use of the "extends" keyword in the class definition.

For example, if we have a vehicle class, we may want to define a car class that inherits the properties and methods of the vehicle class. This allows us to reuse code that is common to both the vehicle and car classes, while still allowing us to define unique properties and methods for the car class.

Overall, abstraction, encapsulation, and inheritance are key concepts in Java and are important for creating well-structured and maintainable code that is easy to extend and modify.

Q. Differentiate between Java and C++.

COMPARISON PARAMETER	**C++**	**JAVA**
Developed / Founded by	**C++ was developed by Bjarne Stroustrup at Bell Labs in 1979. It was developed as an extension of the C language.**	**Java was developed by James Gosling at Sun Microsystems. Now, it is owned by Oracle.**
Programming model	**It has support for both procedural programming and object oriented programming.**	**Java has support only for object oriented programming models.**
Platform dependence	**C++ is platform dependent. It is based on the concept of Write Once Compile Anywhere.**	**Java is platform independent. It is based on the concept of Write Once Run Anywhere.**
Features supported	**C++ supports features like operator overloading, Goto statements, structures, pointers, unions, etc.**	**Java does not support features like operator overloading, Goto statements, structures, pointers, unions, etc.**
Compilation and Interpretation	**C ++ is only compiled and cannot be interpreted.**	**Java can be both compiled and interpreted.**
Library and Code reusability support	**C++ has very limited libraries with low level functionalities. C++ allows direct calls to native system libraries.**	**Java, on the other hand, has more diverse libraries with a lot of support for code reusability. In Java, only calls through the Java**

		Native Interface and recently Java Native Access are allowed.
Memory Management	In C++, memory management is manual.	In Java, memory management is System controlled.
Type semantics	C++ is pretty consistent between primitive and object types.	In Java, semantics differs for primitive and object types.
Global Scope	In C++, both global and namespace scopes are supported.	Java has no support for global scope.
Access control and object protection	In C++, a flexible model with constant protection is available.	In Java, the model is cumbersome and encourages weak encapsulation.

Q. Write the difference between JDK and JRE in Java.

Ans:-

JDK	JRE
JDK(Java Development Kit) is used to develop Java applications. JDK also contains numerous development tools like compilers, debuggers, etc.	JRE(Java Runtime Environment) is the implementation of JVM(Java Virtual Machine) and it is specially designed to execute Java programs.
It is mainly used for the execution of code and its main functionality is development.	It is mainly used for creating an environment for code execution.
It is platform-dependent.	It is also platform-dependent like JDK.
JDK is responsible for the development purpose, therefore it contains tools which are required for development and debugging purpose.	JRE is not responsible for development purposes so it doesn't contain such tools as the compiler, debugger, etc. Instead, it contains class libraries and supporting files required for the purpose of execution of the program.
JDK = JRE + other development tools.	JRE = JVM + other class libraries.

Q. What do you mean by life cycle of applet? Explain it with suitable diagram.

Ans:- In Java, an applet is a program that runs inside a web browser and is written in the Java programming language. The life cycle of an applet refers to the various stages that an applet goes through, from its initialization to its termination.

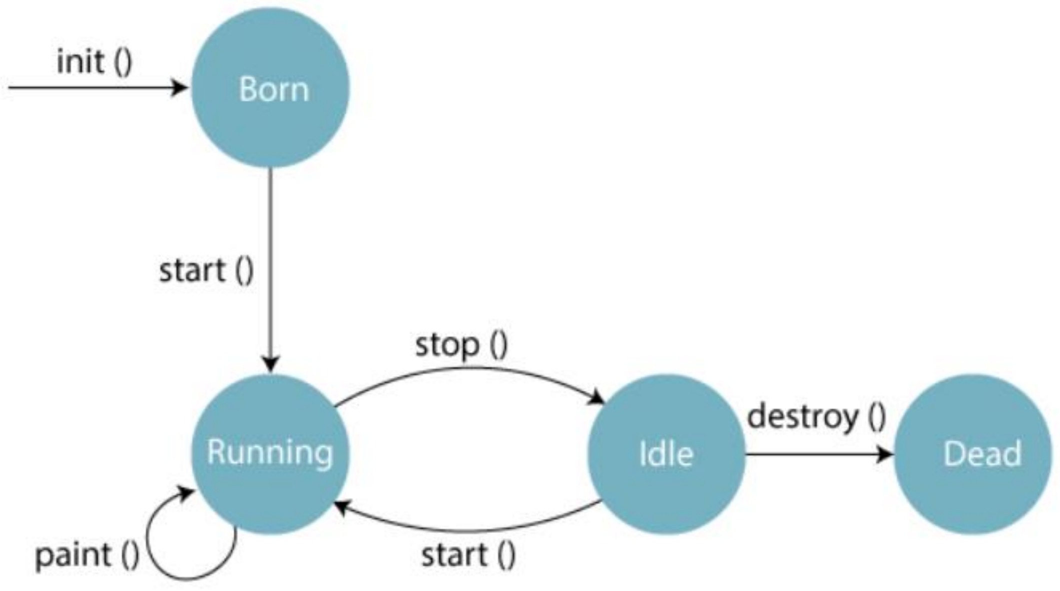
The life cycle of an applet is divided into four stages:

1. **Initialization: In this stage, the applet is loaded by the web browser and the init() method is called. This method is used to initialize the applet and is typically used to set up any resources that the applet needs.**

2. **Execution: In this stage, the applet is in its active state and the start() method is called. This method is used to start the execution of the applet and is typically used to begin any animation or other dynamic content.**

3. **Termination: In this stage, the applet is stopped and the stop() method is called. This method is used to stop the execution of the applet and is typically used to clean up any resources that were used by the applet.**
4. **Deletion: In this stage, the applet is deleted and the destroy() method is called. This method is used to release any resources that were used by the applet and is typically used to free up memory.**

Here is a diagram that illustrates the life cycle of an applet:



Q. Write short notes on:

(a) Byte Code

Ans:- Bytecode is an important concept in Java programming, as it is the intermediate code that is generated by the Java compiler and executed by the Java Virtual Machine (JVM). Bytecode is a binary representation of Java code that is platform-independent, which means that it can be executed on any device that has a JVM installed.

(b) Java applet

Ans:- A Java applet is a type of Java program that is designed to run within a web browser. It is a small application that is written in the Java programming language, and is typically used to provide interactive content on a website.

Java applets are designed to be platform-independent, which means that they can run on any device that has a Java Virtual Machine (JVM) installed. This makes them ideal for creating interactive content that can be accessed from a variety of devices and platforms.

(c) JVM

Ans:- JVM stands for Java Virtual Machine, which is a virtual machine that is used to run Java programs. The JVM is an important part of the Java platform, as it provides a platform-independent execution environment for Java programs.

The JVM is responsible for interpreting Java bytecode, which is the binary format of Java programs that is generated by the Java compiler. When a Java program is executed, the JVM loads the bytecode and executes it in a way that is optimized for the underlying platform.

Q. What do you mean by primitive and non-primitive data types? Define it with examples.

Ans:- In Java, there are two types of data types: primitive data types and non-primitive data types.

Primitive data types are the most basic data types in Java, and they represent simple values. They are built into the Java language, and are not objects. There are eight primitive data types in Java:

1. boolean: represents a true or false value.
2. byte: represents an 8-bit integer value.
3. short: represents a 16-bit integer value.
4. int: represents a 32-bit integer value.
5. long: represents a 64-bit integer value.
6. float: represents a single-precision floating-point value.
7. double: represents a double-precision floating-point value.
8. char: represents a single character value.

Examples of primitive data types in Java:

```
boolean flag = true;

byte b = 127;

short s = 32767;

int i = 2147483647;

long l = 9223372036854775807L;

float f = 3.14f;

double d = 3.14159;

char c = 'a';
```

Non-primitive data types, also known as reference types, are data types that are derived from the primitive data types. They are objects that are created from classes, and they store a reference to the memory location where the actual data is stored. Non-primitive data types include classes, interfaces, arrays, and enums.

Examples of non-primitive data types in Java:

```
String str = "Hello World";

Integer num = 10;

int[] arr = {1, 2, 3, 4, 5};

ArrayList<String> list = new ArrayList<String>();
```

Q. Discuss in details about type casting and type conversion in Java.

Ans:- Type casting and type conversion are two important concepts in Java that are used to convert one data type to another. In this context, data type refers to the type of data that a variable can store, such as int, double, or boolean.

Type Casting: Type casting is the process of converting one data type to another. In Java, there are two types of casting: implicit casting and explicit casting.

Implicit casting, also known as widening or automatic casting, occurs when a smaller data type is converted to a larger data type. This happens automatically, without the need for any explicit syntax. For example, an int can be implicitly cast to a double:

int i = 10;

double d = i; // implicit casting from int to double

Explicit casting, also known as narrowing or manual casting, occurs when a larger data type is converted to a smaller data type. This requires the use of explicit syntax, in the form of a cast operator. For example, a double can be explicitly cast to an int:

double d = 3.14159;

int i = (int) d; // explicit casting from double to int

Type Conversion:

Type conversion is the process of converting one data type to another. In Java, type conversion can occur in several ways:

1. Automatic conversion: This occurs when a value of one data type is assigned to a variable of another data type, and the conversion is allowed by the Java compiler. For example, an int can be automatically converted to a double:

int i = 10;

double d = i; // automatic conversion from int to double

2. Manual conversion: This occurs when a value of one data type is explicitly converted to another data type using a conversion method provided by the Java programming language. For example, a String can be manually converted to an int using the parseInt() method:

String str = "123";

int i = Integer.parseInt(str); // manual conversion from String to int

3. Object conversion: This occurs when an object of one class is converted to an object of another class using casting. For example, a Circle object can be converted to a Shape object using casting:

Circle c = new Circle();

Shape s = (Shape) c; // object conversion from Circle to Shape using casting

Q. What is an array? Discuss different types of array in Java.

Ans:- An array is a data structure in Java that allows you to store multiple values of the same data type in a single variable. Arrays in Java are static, which means that their size cannot be changed after they are created. However, you can change the values of individual elements within an array.

There are different types of arrays in Java:

1. One-Dimensional Array: A one-dimensional array is a list of elements that are all of the same data type. Each element in the array is assigned an index number that represents its position in the array. The index numbers start at 0 and go up to the number of elements in the array minus one.

int[] numbers = {1, 2, 3, 4, 5};

2. Multi-Dimensional Array: A multi-dimensional array is an array that contains one or more arrays as its elements. The most common type of multi-dimensional array is a two-dimensional array, which is a table of elements that are all of the same data type. Each element in the table is assigned two index numbers, one for the row and one for the column.

int[][] table = {{1, 2}, {3, 4}, {5, 6}};

Q. What do you mean by identifiers in Java? Explain it.

Ans:- Identifiers in Java are names given to entities such as classes, variables, methods, and interfaces. An identifier can be composed of letters, digits, the underscore character (_) and the dollar sign ($). However, the first character of an identifier must always be a letter, underscore or a dollar sign.

In Java, identifiers are case sensitive, which means that two identifiers with the same spelling but different capitalization are considered to be different. For example, the identifiers "name", "Name", and "NAME" are all different in Java.

Examples of valid identifiers in Java are:

int age;

String firstName;

double _salary;

float $amount;

On the other hand, the following are examples of invalid identifiers in Java:

int 2numbers; // identifier cannot start with a digit

float my salary; // identifier cannot contain spaces

String first-name; // identifier cannot contain hyphens

Q. What do you mean by bitwise logical operator and Boolean logical operator? Define it.

Ans:- In Java, there are two types of logical operators: bitwise logical operators and Boolean logical operators.

Bitwise logical operators are used to perform operations on individual bits of an integer type. These operators include:

1. AND operator (&): This operator compares each bit of two operands and returns a 1 in the result if both bits are 1, otherwise it returns a 0.
2. OR operator (|): This operator compares each bit of two operands and returns a 1 in the result if either of the bits is 1, otherwise it returns a 0.
3. XOR operator (^): This operator compares each bit of two operands and returns a 1 in the result if the bits are different, otherwise it returns a 0.
4. NOT operator (~): This operator is a unary operator that flips the bits of the operand.

Here's an example of how bitwise operators work:

int a = 5; // 101 in binary

int b = 3; // 011 in binary

int result = a & b; // 001 in binary

In the above example, the AND operator is used to compare the bits of 5 (101 in binary) and 3 (011 in binary). The result is 001 in binary, which is equal to 1 in decimal.

Boolean logical operators are used to perform operations on Boolean values. These operators include:

1. AND operator (&&): This operator returns true if both operands are true, otherwise it returns false.
2. OR operator (||): This operator returns true if either operand is true, otherwise it returns false.
3. NOT operator (!): This operator is a unary operator that flips the value of the operand.

Here's an example of how Boolean operators work:

```
boolean a = true;
boolean b = false;

boolean result = (a && b) || (!a); // true
```

In the above example, the AND operator is used to compare the values of a and b. Since b is false, the result of (a && b) is false. However, the OR operator is used to compare the result of (a && b) and !a. Since !a is true, the result of the entire expression is true.

Q. Explain in brief break and continue statements in java.

Ans:- In Java, the break and continue statements are used to alter the flow of a loop.

The break statement is used to exit a loop prematurely. When a break statement is encountered in a loop, the loop is terminated immediately, and the program execution continues with the next statement after the loop. This is useful when you want to exit a loop based on some condition. Here's an example:

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break;
    }
    System.out.println(i);
}
```

In this example, the loop will iterate from 0 to 9. However, when i equals 5, the break statement is executed, and the loop is terminated. Therefore, only the values 0 to 4 will be printed to the console.

The continue statement is used to skip the current iteration of a loop and continue with the next iteration. When a continue statement is encountered in a loop, the loop immediately goes to the next iteration, skipping any code that follows the continue statement in the current iteration. This is useful when you want to skip a particular iteration of a loop based on some condition. Here's an example:

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        continue;
    }
    System.out.println(i);
}
```

In this example, the loop will iterate from 0 to 9. However, when i equals 5, the continue statement is executed, and the loop immediately goes to the next iteration without executing the println statement. Therefore, the value 5 will be skipped in the output.

Q. Write any program to check leap year in java.

Ans:-

```
import java.util.Scanner;

public class LeapYearChecker {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a year: ");
        int year = scanner.nextInt();
        boolean isLeapYear = false;

        if (year % 4 == 0) {
            if (year % 100 == 0) {
                if (year % 400 == 0) {
                    isLeapYear = true;
                }
            } else {
                isLeapYear = true;
            }
        }

        if (isLeapYear) {
            System.out.println(year + " is a leap year.");
        } else {
            System.out.println(year + " is not a leap year.");
        }
    }
}
```

Q. Write a program in java to find the root of quadratic equation.

Ans:-

```
import java.util.Scanner;

public class QuadraticEquationRoots {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter the values of a, b, and c:");
        double a = scanner.nextDouble();
        double b = scanner.nextDouble();
        double c = scanner.nextDouble();
        double discriminant = b * b - 4 * a * c;
        double root1, root2;

        if (discriminant > 0) {
            root1 = (-b + Math.sqrt(discriminant)) / (2 * a);
            root2 = (-b - Math.sqrt(discriminant)) / (2 * a);
            System.out.println("The roots are real and distinct.");
            System.out.println("Root 1 = " + root1);
            System.out.println("Root 2 = " + root2);
        } else if (discriminant == 0) {
            root1 = -b / (2 * a);
            System.out.println("The roots are real and equal.");
            System.out.println("Root 1 = Root 2 = " + root1);
        } else {
            double realPart = -b / (2 * a);
            double imaginaryPart = Math.sqrt(-discriminant) / (2 * a);
            System.out.println("The roots are complex and conjugate.");
            System.out.println("Root 1 = " + realPart + "+" + imaginaryPart + "i");
            System.out.println("Root 2 = " + realPart + "-" + imaginaryPart + "i");
        }
    }
}
```

Q. What is class in java? How to declare class in java?

Ans:- In Java, a class is a blueprint or a template that defines the properties and behaviors of objects. It describes the state and behavior that objects of the class will possess. A class can be thought of as a user-defined data type, which can be used to create instances or objects of that type.

To declare a class in Java, you need to use the class keyword followed by the name of the class. Here is the basic syntax for declaring a class in Java:

```
public class MyClass {
    // instance variables (properties)
    // constructor
    // methods
}
```

Let's look at each part of this syntax in more detail:

- **public**: This is an access modifier that specifies the visibility of the class. In this case, the class is declared as public, which means that it can be accessed from any other class in the same package or from other packages as well.
- **class**: This is a keyword that tells the Java compiler that you are declaring a class.
- **MyClass**: This is the name of the class. Class names in Java should always start with a capital letter and follow CamelCase convention.
- Instance variables or properties: These are the variables that define the state or attributes of the objects of the class. They are declared inside the class, but outside of any methods.
- Constructor: This is a special method that is used to initialize the objects of the class. It has the same name as the class and no return type.
- Methods: These are the functions that define the behavior of the objects of the class. They are declared inside the class, after the instance variables and constructor.

Here's an example of a simple Java class:

```
public class Rectangle {
    private int length;
    private int width;

    public Rectangle(int length, int width) {
        this.length = length;
        this.width = width;
    }

    public int getArea() {
        return length * width;
    }

    public int getPerimeter() {
        return 2 * (length + width);
    }
}
```

In this example, we have defined a class named **Rectangle** that has two instance variables (**length** and **width**) and two methods (**getArea** and **getPerimeter**). We have also defined a constructor that initializes the **length** and **width** instance variables when a new **Rectangle** object is created.

Q. What is constructor? Explain different types of constructor in java.

Ans:- In Java, a constructor is a special type of method that is used to initialize the state of an object when it is created. It has the same name as the class and no return type. Constructors are called automatically when an object is created using the new keyword.

There are three types of constructors in Java:

1. Default constructor: If a class does not have any constructor, then the Java compiler provides a default constructor that takes no arguments and has an empty body. The default constructor initializes the instance variables with their default values (0 for numeric types, false for boolean, null for reference types).

Here's an example of a default constructor:

```
public class MyClass {
    private int x;
    private String name;
    // default constructor
    public MyClass() {
    }
}
```

2. Parameterized constructor: A parameterized constructor is a constructor that takes one or more parameters. It is used to set the initial values of the instance variables when an object is created.

Here's an example of a parameterized constructor:

```
public class Rectangle {
    private int length;
    private int width;

    // parameterized constructor
    public Rectangle(int length, int width) {
        this.length = length;
        this.width = width;
    }
}
```

3. Copy constructor: A copy constructor is a constructor that takes an object of the same class as a parameter and creates a new object with the same state as the parameter object. It is used to create a new object that is a copy of an existing object.

Here's an example of a copy constructor:

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

      // copy constructor
      public Student(Student other) {
            this.name = other.name;
            this.age = other.age;
```

```
        }
 }
```

In this example, we have defined a copy constructor for the Student class that takes another Student object as a parameter and creates a new Student object with the same name and age as the parameter object.

Note that a class can have multiple constructors with different signatures (i.e., different number or types of parameters). This is known as constructor overloading.

Q. What do you mean by constructor and destructor?

Ans:- Constructors and destructors are both special methods in object-oriented programming that are used to initialize and clean up objects, respectively.

A constructor is a special method that is called automatically when an object is created. It has the same name as the class and no return type. The purpose of the constructor is to initialize the state of the object, i.e., to set the initial values of the instance variables. Constructors can take zero or more arguments, and they can be overloaded, meaning a class can have multiple constructors with different signatures.

Here's an example of a constructor:

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

In this example, we have defined a constructor for the Person class that takes two arguments (name and age) and sets the corresponding instance variables.

A destructor, on the other hand, is a special method that is called automatically when an object is destroyed or goes out of scope. In Java, there is no explicit destructor like in C++. Instead, Java has a garbage collector that automatically frees up memory that is no longer being used by objects. The garbage collector runs periodically and determines which objects are no longer needed, and then deallocates the memory used by those objects.

So, in summary, constructors are used to initialize objects, while destructors are used to clean up objects. However, in Java, you don't need to worry about destructors because the garbage collector takes care of memory management for you.

Q. Differentiate between final and finalize in java.

Ans:-

Final	Finalize
final is the keyword which is used to apply restrictions on a class, method or variable.	finalize is the method in Java which is used to perform clean up processing just before object is garbage collected.
Final keyword is used with the classes, methods and variables.	finalize() method is used with the objects.

Once declared, final variable then it cannot be modified, overridden & inherited.	Finalize method performs the cleaning activities with respect to the object before its destruction.
Final method is executed only when we call it.	finalize method is executed just before the object is destroyed.

Q. What do you mean by method in java? Why we use method in java program?

Ans:- In Java, a method is a set of instructions that perform a specific task. It is a named block of code that can be called from other parts of the program, allowing you to reuse code and avoid duplicating the same code in multiple places.

Methods can take zero or more input parameters, and they may or may not return a value. A method with a return type returns a value after executing its code block, whereas a method without a return type simply performs some action without returning anything.

Here's an example of a method that takes two integers as input and returns their sum:

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

In this example, the method is named add and takes two integers as input (a and b). It returns the sum of the two integers by adding them together using the + operator.

We use methods in Java for several reasons, including:

1. Reusability: By defining a method once, we can reuse it multiple times in our program, reducing code duplication and making our code more concise.
2. Modularity: Methods allow us to break down a complex program into smaller, more manageable pieces, making it easier to understand and maintain.
3. Abstraction: Methods allow us to abstract away the details of how a particular task is accomplished, making it easier to reason about the program at a higher level.
4. Encapsulation: Methods can be used to encapsulate data and behavior within an object, ensuring that only the appropriate methods can access and modify the object's state.

Overall, methods are an essential part of Java programming, providing a powerful mechanism for organizing and reusing code.

Q. Discuss about method overloading and constructor overloading in java.

Ans:-

Method Overloading

Method overloading is the process of defining multiple methods with the same name in the same class, but with different parameters. The signature of the method must be different for each method, based on the type and/or number of parameters. This allows you to reuse the same method name for multiple operations that perform similar tasks but with different input values.

Here's an example of method overloading:

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

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

In this example, we have defined two methods with the same name **add**, but with different parameters. The first method takes two integers as input, and the second method takes two doubles as input. The methods perform similar tasks (adding two numbers) but with different input values.

When we call the **add** method with integer arguments, the first method will be invoked, and when we call it with double arguments, the second method will be invoked.

Constructor Overloading

Constructor overloading is similar to method overloading, but it involves defining multiple constructors with the same name in the same class, but with different parameters. This allows you to create objects with different initial states based on the type and/or number of parameters passed to the constructor.

Here's an example of constructor overloading:

```
public class MyClass {
    private int x;
    private int y;

    public MyClass() {
        this.x = 0;
        this.y = 0;
    }

    public MyClass(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

In this example, we have defined two constructors with the same name **MyClass**, but with different parameters. The first constructor takes no arguments and initializes the **x** and **y** fields to 0, while the second constructor takes two integer arguments and initializes the **x** and **y** fields to those values.

When we create a new **MyClass** object using the first constructor (**MyClass obj1 = new MyClass()**), the **x** and **y** fields will be initialized to 0. When we create a new **MyClass** object using the second constructor (**MyClass obj2 = new MyClass(10, 20)**), the **x** and **y** fields will be initialized to 10 and 20, respectively.

Overall, method overloading and constructor overloading are powerful features of Java that allow you to write more concise and reusable code. By defining multiple methods or constructors with the same name but different parameters, you can create more flexible and versatile programs.

Q. What do you mean by recursive method? Explain it with example.

Ans:- In Java, a recursive method is a method that calls itself to solve a problem. Recursive methods are often used to solve problems that can be broken down into smaller, simpler subproblems that are similar in structure to the original problem.

Here's an example of a recursive method that calculates the factorial of a given number:

```
public class MyClass {
    public static int factorial(int n) {
        if (n == 0) {
            return 1;
        } else {
            return n * factorial(n - 1);
        }
    }

    public static void main(String[] args) {
        int result = factorial(5);
        System.out.println(result); // Output: 120
    }
}
```

In this example, the **factorial** method takes an integer **n** as input and returns the factorial of **n**. If **n** is 0, the method returns 1 (since the factorial of 0 is 1). Otherwise, it calculates the factorial of **n-1** by calling itself recursively, and multiplies the result by **n** to get the factorial of **n**.

When we call the **factorial** method with **n=5**, the method first calculates the factorial of 4 by calling itself with **n=4**, then calculates the factorial of 3 by calling itself with **n=3**, and so on, until it reaches **n=0** and returns 1. The final result is the factorial of 5, which is 120.

Recursive methods can be very powerful, but they also require careful design to ensure that they terminate and do not cause infinite loops or stack overflows. It's important to have a base case that stops the recursion and to ensure that the input values converge towards the base case.

Q. Write short notes on:

(a) This keyword
Ans:- In Java, the this keyword is used to refer to the current object within a class. It can be used in several ways:

1. To refer to instance variables or methods of the current object:

```
public class MyClass {
    private int value;

    public MyClass(int value) {
        this.value = value; // use "this" to refer to the instance variable
    }

    public void printValue() {
        System.out.println(this.value); // use "this" to refer to the instance variable
    }
}
```

2. **To invoke a constructor from another constructor in the same class:**

```
public class MyClass {
   private int value;

   public MyClass() {
      this(0); // invoke the other constructor with a default value of 0
   }

   public MyClass(int value) {
      this.value = value;
   }
}
```

3. **To return the current object from a method:**

```
public class MyClass {
   private int value;

   public MyClass(int value) {
      this.value = value;
   }

   public MyClass increment() {
      this.value++; // increment the instance variable
      return this; // return the current object
   }
}
```

In all cases, the this keyword is used to refer to the current object within the class. It can be especially useful when working with instance variables or when invoking constructors with different parameters.

(b) Static keyword

Ans:- In Java, the static keyword is used to define a class-level variable or method that belongs to the class itself rather than to any particular instance of the class.

When applied to a variable, the static keyword means that the variable is shared among all instances of the class. For example:

```
public class MyClass {
   public static int count = 0; // declare a static variable

   public MyClass() {
      count++; // increment the static variable in the constructor
   }
}
```

In this example, the count variable is shared among all instances of the MyClass class. When a new instance is created, the count variable is incremented, and this change is reflected in all other instances.

When applied to a method, the static keyword means that the method is associated with the class itself, rather than with any particular instance of the class. For example:

```
public class MyClass {
   public static void printHello() {
      System.out.println("Hello, world!"); // a static method that prints a message
   }
}
```

In this example, the printHello() method is associated with the MyClass class itself, rather than with any particular instance of the class. This means that it can be called directly on the class, without the need to create an instance:

```
MyClass.printHello(); // call the static method directly on the class
```

The **static** keyword can also be used to define a static block of code that is executed when the class is loaded into memory, before any instances of the class are created. This can be useful for initializing static variables or performing other setup tasks:

```
public class MyClass {
   public static int count;

   static {
      count = 0; // initialize the static variable
   }
}
```

In this example, the static block initializes the **count** variable to 0 before any instances of the **MyClass** class are created.

Overall, the **static** keyword allows for the creation of class-level variables and methods that can be shared among all instances of the class, as well as for the execution of static code that is run when the class is loaded into memory.

(c) Garbage collection

Ans:- Garbage collection is an automated memory management process in Java that frees up memory that is no longer being used by the program. It automatically identifies and removes objects that are no longer being referenced by the program, which helps to prevent memory leaks and improves program performance.

The garbage collector is responsible for managing the memory of objects created by the Java Virtual Machine (JVM). When an object is created in Java, it is allocated memory on the heap. When the object is no longer needed, the memory it occupies is marked as "garbage" and becomes eligible for garbage collection. The garbage collector periodically scans the heap for such garbage and frees up the memory used by those objects.

Q. What is inheritance in java? Explain its different types.

Ans:- Inheritance is a mechanism in Java that allows a class to inherit the properties (methods and variables) of another class. The class that is being inherited is called the superclass, while the class that inherits the superclass is called the subclass. In Java, inheritance is implemented using the "extends" keyword.

The main benefit of inheritance is that it promotes code reusability, since the subclass can reuse the methods and variables of the superclass without having to redefine them.

There are several types of inheritance in Java, including:

1. Single Inheritance: In this type of inheritance, a subclass can only inherit from a single superclass.
2. Multilevel Inheritance: In this type of inheritance, a subclass inherits from a superclass, which in turn inherits from another superclass, and so on.
3. Hierarchical Inheritance: In this type of inheritance, multiple subclasses inherit from a single superclass.
4. Multiple Inheritance (not supported in Java): In this type of inheritance, a subclass can inherit from multiple superclasses. Java does not support multiple inheritance directly, but it can be achieved through the use of interfaces.
5. Hybrid Inheritance (not supported in Java): This is a combination of two or more types of inheritance.

Inheritance in Java is also subject to certain rules and restrictions, such as the fact that a subclass cannot inherit private members of a superclass, and that the subclass must invoke the constructor of the superclass in order to initialize its inherited members.

Overall, inheritance is a powerful feature of Java that allows for code reuse and promotes modularity and flexibility in object-oriented programming.

Q. What do you mean by multi-level inheritance? Explain it with example.

Ans:- The multi-level inheritance includes the involvement of at least two or more than two classes. One class inherits the features from a parent class and the newly created sub-class becomes the base class for another new class.

Example:-

```
class Car{
  public Car()
  {
        System.out.println("Class Car");
  }
  public void vehicleType()
  {
        System.out.println("Vehicle Type: Car");
  }
}
class Maruti extends Car{
  public Maruti()
  {
        System.out.println("Class Maruti");
  }
  public void brand()
  {
        System.out.println("Brand: Maruti");
  }
  public void speed()
  {
        System.out.println("Max: 90Kmph");
  }
}
```

```
public class Maruti800 extends Maruti{

   public Maruti800()
   {
         System.out.println("Maruti Model: 800");
   }
   public void speed()
   {
         System.out.println("Max: 80Kmph");
   }
   public static void main(String args[])
   {
          Maruti800 obj=new Maruti800();
          obj.vehicleType();
          obj.brand();
          obj.speed();
   }
}
```

Q. Differentiate between method overloading and method overriding.

Method Overloading	Method Overriding
Overloading is a compile-time polymorphism.	Overriding is a run-time polymorphism.
It helps to increase the readability of the program.	It is used to specific implementation of the method which is already provided by its parent class.
It occurs within the class.	It is performed in two classes.
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.
Poor Performance due to compile time polymorphism.	It gives better performance.

Q. What do you mean by interface? Define it with example.

Ans:- In object-oriented programming, an interface is a set of methods or functions that a class must implement in order to be considered compatible with that interface. An interface defines a contract between a class and the code that uses it, specifying what methods the class must provide and how they must behave.

An interface is similar to an abstract class in that it defines a set of methods that a class must implement, but unlike an abstract class, an interface cannot provide any implementation of its own methods.

Here is an example of an interface in Java:

```
public interface Shape {
    double area();
    double perimeter();
}
```

In this example, we have defined an interface called Shape. The Shape interface has two methods: area() and perimeter(). Any class that implements the Shape interface must provide an implementation for these two methods.

For example, we could define a Circle class that implements the Shape interface:

```
public class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double area() {
        return Math.PI * radius * radius;
    }

    public double perimeter() {
        return 2 * Math.PI * radius;
    }
}
```

In this example, the Circle class implements the Shape interface by providing implementations for the area() and perimeter() methods. The Circle class also has a constructor that takes a radius parameter and stores it as an instance variable.

Because the Circle class implements the Shape interface, it can be used anywhere that expects an object that implements the Shape interface. For example, we could create an array of Shape objects and add a Circle object to it:

```
Shape[] shapes = new Shape[1];
shapes[0] = new Circle(5.0);
```

In this example, we have created an array of Shape objects and added a Circle object to it. Because the Circle class implements the Shape interface, it is compatible with the Shape array. We can then call the area() and perimeter() methods on the Circle object as if it were a Shape object:

```
double area = shapes[0].area();
double perimeter = shapes[0].perimeter();
```

Q. What do you mean by class modifier? Define it with example.

Ans:- In Java, a class modifier is a keyword that can be used to modify the accessibility or behavior of a class. There are several class modifiers in Java, including public, private, protected, abstract, final, and static.

Here are some examples of class modifiers in Java and their effects:

- **public:** A public class is accessible from anywhere in the program. Other classes can create instances of the public class and call its methods. Here's an example of a public class in Java:

```
public class MyClass {
    // class definition goes here
}
```

private: A private class is only accessible within the same file or class that defines it. Other classes cannot create instances of the private class or call its methods. Here's an example of a private class in Java:

```
private class MyClass {
    // class definition goes here
}
```

protected: A protected class is accessible within the same package or subclass that defines it. Other classes outside the package cannot create instances of the protected class or call its methods. Here's an example of a protected class in Java:

```
protected class MyClass {
    // class definition goes here
}
```

abstract: An abstract class is a class that cannot be instantiated directly, but can be used as a base class for other classes. Abstract classes often define some methods that must be implemented by any class that inherits from them. Here's an example of an abstract class in Java:

```
public abstract class MyBaseClass {
    public abstract void myMethod(); // abstract method
}

public class MySubclass extends MyBaseClass {
    public void myMethod() {
        // implementation of the abstract method goes here
    }
}
```

In this example, the **MyBaseClass** class defines an abstract method called **myMethod()**, which means that any class that inherits from **MyBaseClass** must implement this method. The **MySubclass** class inherits from **MyBaseClass** and provides an implementation of the **myMethod()** method.

- **final**: A final class is a class that cannot be subclassed by other classes. Here's an example of a final class in Java:

```
public final class MyClass {
    // class definition goes here
}
```

In this example, the **MyClass** class cannot be subclassed by other classes.

- **static**: A static class is a class that can be accessed without creating an instance of it. Static classes often contain utility methods that can be used throughout the program. Here's an example of a static class in Java:

```
public class MyUtilityClass {
    public static void myMethod() {
        // implementation of the static method goes here
    }
}
```

In this example, the **MyUtilityClass** class contains a static method called **myMethod()**, which can be called without creating an instance of **MyUtilityClass**.

Q. What is Exception in Java? Explain its type in brief.

Ans:- In Java, an exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. When an exception occurs, the program stops executing its normal instructions and starts looking for an exception handler that can deal with the situation.

In Java, exceptions are objects that are created when an error or other exceptional condition occurs. These objects are thrown by the code that encounters the problem, and can be caught and handled by other code that knows how to deal with them.

There are two main types of exceptions in Java: checked exceptions and unchecked exceptions.

- Checked Exceptions: Checked exceptions are exceptions that the programmer is required to handle explicitly in the code. These exceptions are checked by the compiler at compile-time, and the programmer must include code to handle them. If the programmer does not handle the checked exception, the code will not compile. Some examples of checked exceptions in Java include IOException, SQLException, and ClassNotFoundException.

- Unchecked Exceptions: Unchecked exceptions are exceptions that the programmer is not required to handle explicitly in the code. These exceptions are not checked by the compiler at compile-time, and the programmer is not required to include code to handle them. If an unchecked exception occurs and is not caught by the code, the program will terminate with an error message. Some examples of unchecked exceptions in Java include NullPointerException, ArrayIndexOutOfBoundsException, and ArithmeticException.

In addition to these two main types of exceptions, Java also includes other types of exceptions, including runtime exceptions and errors. Runtime exceptions are unchecked exceptions that occur at runtime, such as NullPointerException and ArrayIndexOutOfBoundsException. Errors are exceptional conditions that are not recoverable, such as OutOfMemoryError and StackOverflowError, and typically cause the program to terminate.

Q. What do you mean by un-checked exceptions? Explain it.

Ans:- In Java, an unchecked exception is an exception that occurs during the execution of a program that is not required to be caught or handled explicitly by the program's code. Unchecked exceptions are not checked by the compiler at compile-time, and the programmer is not required to include code to handle them.

Unchecked exceptions are typically caused by programming errors, such as null pointer dereferences, array index out of bounds errors, or arithmetic errors. These types of errors are usually indicative of bugs in the program's logic, and are not expected to occur during normal program execution.

When an unchecked exception occurs, the program will terminate and an error message will be displayed. This can be useful for debugging and identifying problems in the program's logic.

Some examples of unchecked exceptions in Java include:

- NullPointerException: Occurs when a null reference is used in a method or expression that requires a non-null reference.

- ArrayIndexOutOfBoundsException: Occurs when an array index is out of bounds.
- ArithmeticException: Occurs when an arithmetic operation results in an overflow or divide-by-zero error.
- ClassCastException: Occurs when an object is cast to an incompatible type.
- IllegalArgumentException: Occurs when an invalid argument is passed to a method.

It is generally a good practice to handle unchecked exceptions in the code, even though it is not required by the Java language. This can help make the program more robust and avoid unexpected crashes or errors. However, it is important to note that some unchecked exceptions may be impossible to handle, such as OutOfMemoryError, which occurs when the Java Virtual Machine runs out of memory.

Q. Explain various available Exception keywords in java.

Ans:- In Java, there are several keywords related to exceptions that are used to handle exceptions in different ways. Here is a brief overview of the most commonly used exception keywords:

1. try: The try block is used to enclose a block of code that may throw an exception. If an exception is thrown within the try block, the program will immediately jump to the corresponding catch block to handle the exception.
2. catch: The catch block is used to handle exceptions thrown within the corresponding try block. Each catch block specifies the type of exception it can handle, and the code within the block is executed only if that type of exception is thrown. Multiple catch blocks can be chained together to handle different types of exceptions.
3. finally: The finally block is used to specify code that will be executed regardless of whether an exception is thrown or not. This block is typically used to clean up resources that were allocated within the try block.
4. throw: The throw keyword is used to explicitly throw an exception from a method or block of code.
5. throws: The throws keyword is used to specify that a method may throw a particular type of exception. When a method is declared with a throws clause, any code that calls that method must either handle the exception or declare that it may throw the exception as well.

Here's an example that demonstrates the use of these keywords:

```
public class Example {
  public static void main(String[] args) {
    try {
      int a = Integer.parseInt(args[0]);
      int b = Integer.parseInt(args[1]);
      int result = divide(a, b);
      System.out.println("Result: " + result);
    } catch (NumberFormatException e) {
      System.out.println("Invalid input: " + e.getMessage());
    } catch (ArithmeticException e) {
      System.out.println("Division by zero: " + e.getMessage());
    } finally {
      System.out.println("Program complete.");
    }
  }

  public static int divide(int a, int b) throws ArithmeticException {
    if (b == 0) {
      throw new ArithmeticException("Division by zero");
    }
    return a / b;
  }
}
```

Q. Write short notes on:

(a) Abstract class

Ans:- In Java, an abstract class is a class that cannot be instantiated directly, and is typically used as a base class for other classes to inherit from. An abstract class may contain abstract methods, which are methods that have a signature but no implementation, and concrete methods, which have an implementation.

Here are some key features of abstract classes in Java:

- An abstract class is declared using the abstract keyword.
- An abstract class may contain both abstract and concrete methods.
- An abstract method is declared using the abstract keyword and has no implementation. Any class that extends an abstract class must implement all of its abstract methods.
- A concrete method is a method that has an implementation. Concrete methods can be called directly by instances of the class or by any subclass.
- An abstract class may not be instantiated directly. Instead, it must be subclassed and the subclass must provide implementations for all of its abstract methods.
- An abstract class can have instance variables, constructors, and non-abstract methods just like any other class.

Here's an example of an abstract class in Java:

```
public abstract class Animal {
  protected int age;

  public Animal(int age) {
    this.age = age;
  }

  public abstract void makeSound();

  public void eat() {
    System.out.println("Nom nom nom");
  }
}

public class Dog extends Animal {
  public Dog(int age) {
    super(age);
  }

  public void makeSound() {
    System.out.println("Woof woof");
  }
}

public class Cat extends Animal {
  public Cat(int age) {
    super(age);
  }

  public void makeSound() {
    System.out.println("Meow meow");
  }
}

public class Main {
  public static void main(String[] args) {
    Animal dog = new Dog(2);
    Animal cat = new Cat(3);
    dog.makeSound();
    cat.makeSound();
  }
}
```

(b) Object class

Ans:- In Java, the Object class is the root class of the class hierarchy. Every class in Java is a subclass of the Object class, either directly or indirectly. This means that the Object class provides a basic set of methods and properties that are inherited by all other classes in Java.

Here are some key features of the Object class in Java:

- The Object class provides a default implementation for several methods, such as toString(), equals(), and hashCode(). These methods are used by many other classes in Java and can be overridden by subclasses to provide custom implementations.
- The Object class defines the getClass() method, which returns the runtime class of an object. This method can be used to determine the class of an object at runtime.
- The Object class provides the wait(), notify(), and notifyAll() methods, which are used for inter-thread communication in Java.
- The Object class provides the finalize() method, which is called by the garbage collector when an object is no longer referenced. This method can be overridden by subclasses to perform cleanup tasks before the object is garbage collected.

Here's an example of how the Object class is used in Java:

```
public class MyClass {
  private int value;
  public MyClass(int value) {
    this.value = value;
  }

  public int getValue() {
    return value;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    MyClass myClass = (MyClass) o;
    return value == myClass.value;
  }

  @Override
  public int hashCode() {
    return Objects.hash(value);
  }

  @Override
  public String toString() {
    return "MyClass{" +
            "value=" + value +
            '}';
  }
}

public class Main {
  public static void main(String[] args) {
    MyClass obj1 = new MyClass(42);
    MyClass obj2 = new MyClass(42);

    System.out.println(obj1.equals(obj2)); // true
    System.out.println(obj1.hashCode()); // 42
    System.out.println(obj1.toString()); // MyClass{value=42}
  }
}
```

(c) Try & catch keyword

Ans:- In Java, the try and catch keywords are used for exception handling. The basic idea behind exception handling is to allow the program to gracefully handle unexpected errors or situations that can cause the program to terminate abruptly.

The try block contains the code that may throw an exception. If an exception is thrown, the program will jump to the corresponding catch block to handle the exception. Here's the basic syntax for using try and catch in Java:

```
try {
  // Code that may throw an exception
} catch (ExceptionType exceptionName) {
  // Code to handle the exception
}
```

Here's an example that demonstrates the use of try and catch:

```
public class Main {
  public static void main(String[] args) {
    int a = 10;
    int b = 0;

    try {
      int result = a / b;
      System.out.println("Result: " + result);
    } catch (ArithmeticException e) {
      System.out.println("Error: " + e.getMessage());
    }

    System.out.println("Program continues...");
  }
}
```

Q. What is thread? Explain life cycle of thread in Java.

Ans:- In Java, a thread is a separate flow of execution within a program. A thread allows a program to perform multiple tasks concurrently. Each thread has its own call stack, but shares the same heap memory with other threads in the same process.

The life cycle of a thread in Java consists of five states: New, Runnable, Blocked, Waiting, and Terminated.

1. New: A thread is in the New state when it is created, but has not yet started running. In this state, the thread has not yet been started and is not yet eligible for scheduling by the operating system.
2. Runnable: A thread is in the Runnable state when it is ready to run, but has not yet been selected by the thread scheduler to run. In this state, the thread is waiting for a CPU time slice to execute.
3. Blocked: A thread is in the Blocked state when it is waiting for a lock or some other condition to be released. In this state, the thread is not eligible for CPU time and is waiting for some event to occur before it can continue running.
4. Waiting: A thread is in the Waiting state when it is waiting for some other thread to perform a specific action before it can continue running. In this state, the thread is not eligible for CPU time and is waiting for some other thread to signal that it is ready to continue running.
5. Terminated: A thread is in the Terminated state when it has finished running or has been terminated by an exception. Once a thread is in the Terminated state, it cannot be restarted.

Here is an example that demonstrates the life cycle of a thread in Java:

```
public class MyThread implements Runnable {
  public void run() {
    System.out.println("Thread started running");
    try {
      Thread.sleep(5000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("Thread finished running");
  }
}

public class Main {
  public static void main(String[] args) {
    Thread t = new Thread(new MyThread());
    System.out.println("Thread created");
    t.start();
    System.out.println("Thread started");
    try {
      Thread.sleep(2000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("Thread state: " + t.getState());
  }
}
```

While the thread is sleeping, we print the state of the thread using the getState() method, which will show us that the thread is in the Waiting state.

Once the thread has finished running, it enters the Terminated state and cannot be restarted.

Here are some key points to keep in mind when working with threads in Java:

- Java provides a Thread class that can be used to create and manage threads.
- You can create a thread by implementing the Runnable interface and passing an instance of the implementation to a Thread object.
- Threads can be used to perform tasks concurrently and improve the performance of a program.
- Proper synchronization mechanisms should be used to avoid race conditions and other concurrency issues.

Q. Differentiate between multi-tasking and multi-threading.

Multitasking	**Multithreading**
In multitasking, users are allowed to perform many tasks by CPU.	In multithreading, many threads are created from a process.
Here, the processes share separate memory.	Here, processes are allocated the same memory.
It involves in multiprocessing.	It does not involve in multiprocessing.
In multitasking, the CPU is provided in order to execute many tasks at a time.	In multithreading, a CPU is provided in order to execute many threads of a process at a time.

In multitasking, processes don’t share the same resources.	While in multithreading, each process shares the same resources.
Multitasking is slow compared to multithreading.	While multithreading is faster.
It helps in developing efficient programs.	It helps in developing efficient operating systems.

Q. What do you mean by resuming and stoping of threads? Explain it.

Ans:- In Java, threads can be paused and resumed using the suspend() and resume() methods provided by the Thread class. However, these methods have been deprecated in Java 1.2 due to their potential to cause deadlocks and other issues.

Instead of using suspend() and resume(), the recommended way to pause and resume a thread is to use a boolean flag or another signaling mechanism to indicate when the thread should pause or resume.

For example, consider the following code:

```
public class MyThread implements Runnable {
  private volatile boolean running = true;

  public void run() {
    System.out.println("Thread started running");
    while (running) {
      System.out.println("Thread is running");
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    System.out.println("Thread finished running");
  }

  public void stop() {
    running = false;
  }
}

public class Main {
  public static void main(String[] args) {
    MyThread myThread = new MyThread();
    Thread t = new Thread(myThread);
    t.start();
    try {
      Thread.sleep(5000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    myThread.stop();
  }
}
```

Q. What do you mean by stream in Java? Discuss its type in brief.

Ans:- In Java, a stream is a sequence of elements that can be processed in parallel or sequentially. Streams are a feature introduced in Java 8 to enable functional-style operations on collections of data.

There are two types of streams in Java:

1. Stream<T>: This is a stream of objects of type T. It can be a stream of integers, strings, or any other object.
2. IntStream, LongStream, and DoubleStream: These are specialized streams for handling primitive data types. IntStream is a stream of integers, LongStream is a stream of long integers, and DoubleStream is a stream of doubles.

Streams support various operations such as filtering, mapping, and reducing, which can be used to perform complex operations on data with very little code. Streams can also be used to perform operations in parallel, which can lead to significant performance gains on multi-core systems.

Some of the commonly used stream operations are:

1. filter(): This operation filters out elements from a stream based on a given condition.
2. map(): This operation applies a function to each element in the stream and returns a new stream.
3. reduce(): This operation reduces the elements of a stream to a single value based on a given operation.
4. collect(): This operation collects the elements of a stream into a collection or other data structure.
5. forEach(): This operation applies a function to each element in the stream.

Streams are also lazy, which means that they don't start processing elements until they are needed. This allows for more efficient processing of large data sets.

Overall, streams are a powerful tool for processing data in Java, and they can make code more concise, readable, and maintainable.

Q. Write the difference between character stream and byte stream.

Ans:-

Character stream	**Byte stream**
Character stream is used to read and write a single character of data.	Byte stream is used to read and write a single byte (8 bits) of data.
Reading or writing a character or text based I/O such as text file, text documents, XML, HTML etc.	Reading or writing to binary data such as exe file, image file, LLL formats file like .zip, .class, .exe etc.
Input and output character streams are called readers and writers respectively.	Input and output byte streams are simply called input streams and output streams respectively.
The abstract class reader and writer and their derived classes in java.io package provide support for character streams.	The abstract class InputStream and OutputStream and their derived classes in java.io package provide support for byte streams.

Q. Explain input and output console in Java.

Ans:- In Java, the console is the standard input/output device for the command-line environment. It is typically used to read input from the user and display output to the user.

Java provides two classes for handling input and output to the console:

1. System.in: This is an InputStream object that represents the standard input stream. It is used to read input from the console.
2. System.out: This is a PrintStream object that represents the standard output stream. It is used to display output to the console.

To read input from the console, you can use the Scanner class, which provides methods for reading different types of input such as strings, integers, and doubles. Here's an example of how to use the Scanner class to read a string from the console:

```
import java.util.Scanner;

public class Main {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter your name: ");
    String name = scanner.nextLine();
    System.out.println("Hello, " + name + "!");
  }
}
```

To display output to the console, you can use the System.out stream along with the print() or println() methods. Here's an example:

```
public class Main {
  public static void main(String[] args) {
    System.out.println("Hello, world!");
  }
}
```

Q. Write short notes on:

(a). Thread synchronization

Ans:- In Java, thread synchronization refers to the coordination of multiple threads to ensure that they access shared resources in a safe and predictable manner. Without proper synchronization, multiple threads may access shared resources simultaneously, leading to race conditions and other types of concurrency bugs.

(b). Alive() and join()

Ans:- alive() and join() are two methods in Java that are used to manage threads.

1. alive(): This method is used to check if a thread is still alive or has completed its execution. It returns true if the thread is still running, and false if the thread has completed its execution.

Here's an example of how to use the alive() method:

```
public class MyThread extends Thread {
  public void run() {
    // do some work
  }
}

public class Main {
  public static void main(String[] args) {
    MyThread thread = new MyThread();
    thread.start();

    // wait for the thread to finish
    while (thread.isAlive()) {
      System.out.println("Thread is still running...");
    }
    System.out.println("Thread has completed.");
  }
}
```

2. join(): This method is used to wait for a thread to complete its execution before continuing with the rest of the program. When you call the join() method on a thread, the calling thread will block until the target thread has completed its execution.

Here's an example of how to use the join() method:

```
public class MyThread extends Thread {
  public void run() {
    // do some work
  }
}

public class Main {
  public static void main(String[] args) throws InterruptedException {
    MyThread thread = new MyThread();
    thread.start();

    // wait for the thread to finish
    thread.join();

    System.out.println("Thread has completed.");
  }
}
```

Q. What do you mean by JDBC? Explain it.

Ans:- JDBC stands for Java Database Connectivity. It is a Java API that allows Java programs to interact with relational databases such as MySQL, Oracle, and SQL Server.

JDBC provides a set of classes and interfaces that allow Java programs to perform the following database-related tasks:

1. Establish a connection to a database
2. Send SQL statements to the database for execution
3. Retrieve and process the results of SQL queries

4. Manage transactions

To use JDBC in a Java program, you first need to load the JDBC driver for the specific database you want to connect to. Once the driver is loaded, you can establish a connection to the database using a connection string that specifies the database URL, username, and password.

Once you have a connection to the database, you can create a statement object and use it to send SQL statements to the database for execution. You can execute SQL queries to retrieve data from the database, or execute SQL update statements to modify data in the database.

JDBC also provides support for transactions, which allow multiple SQL statements to be executed as a single atomic unit. This ensures that all statements in the transaction are either executed successfully or rolled back if an error occurs.

Overall, JDBC is a powerful API that allows Java programs to interact with relational databases and perform a wide range of database-related tasks.