UNIT-1

FUNDAMENTALS OF JAVA PROGRAMMING

1.1 INTRODUCTION TO JAVA

Java is a high-level, object-oriented programming language that is designed to be portable, simple, and secure. It was created in the mid-1990s by James Gosling at Sun Microsystems (which is now owned by Oracle Corporation) and has since become one of the most popular programming languages in use today.

One of the key features of Java is its platform independence. Java programs can run on any computer system that has a Java Virtual Machine (JVM) installed, which means that developers can write code once and run it on any platform. This makes Java an ideal language for developing web applications and mobile apps.

Another important feature of Java is its object-oriented programming (OOP) model. Everything in Java is an object, which makes it easy to create reusable code and maintain large codebases. Java also has strong type checking, which helps to prevent common programming errors.

Java has a vast array of libraries and frameworks that make it easy to write complex programs quickly. It is widely used for developing enterprise-level applications, web applications, mobile apps, games, and much more.

In order to get started with Java programming, you will need to download and install the Java Development Kit (JDK) from Oracle's website. Once you have the JDK installed, you can use an Integrated Development Environment (IDE) such as Eclipse, IntelliJ IDEA, or NetBeans to write and debug your code.

1.2 HISTORY OF JAVA AND FEATURES

Java was created by James Gosling and his team of developers at Sun Microsystems (which was later acquired by Oracle Corporation) in the mid-1990s. The project was initially called "Oak," but was later renamed to Java.

The original goal of the Java project was to create a platform-independent language that could be used to develop software for a wide range of devices, from small embedded systems to large mainframe computers. The language was designed to be simple, secure, and portable, with a strong emphasis on object-oriented programming.

Java was first released to the public in 1995, and quickly gained popularity due to its platform independence and ease of use. Its popularity continued to grow in the late 1990s, as it became the preferred language for developing web applications and applets.

In 2006, Sun Microsystems released Java as open source software under the GNU General Public License. This move helped to further increase the popularity of Java, as it allowed developers to modify and distribute the language freely.

Today, Java is one of the most widely used programming languages in the world, with millions of developers using it to create a wide range of applications, from web applications and mobile apps to enterprise-level software and games. The language continues to evolve, with new features and improvements being added in each new release.

1.2.1 FEATURES OF JAVA:

Java is a powerful and versatile programming language with many features that make it popular among developers. Some of the key features of Java include:

1. **Platform Independence:** Java programs are compiled into platform-independent bytecode that can be run on any platform that has a Java Virtual Machine (JVM) installed. This makes Java programs highly portable.

2. **Object-Oriented Programming:** Java is a pure object-oriented programming language. Everything in Java is an object, which makes it easy to write reusable code and build complex applications.

3. **Robust and Secure:** Java was designed to be a secure and robust language. It has many built-in security features such as a security manager, which ensures that Java programs can run safely even in untrusted environments.

4. **Multithreading:** Java supports multithreading, which allows multiple threads of execution to run concurrently within a single program. This is especially useful for applications that need to perform multiple tasks at the same time.

5. **High Performance:** Java is known for its high performance and scalability. The JVM optimizes Java bytecode at runtime, which helps to improve the performance of Java applications.

6. **Rich Libraries:** Java has a vast array of libraries and frameworks that make it easy to write complex programs quickly. These libraries cover everything from network programming and database connectivity to user interface design and graphics.

7. **Architecture-neutral:** Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed. In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

8. **Dynamic:** Java is a dynamic language. It supports the dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++. Java supports dynamic compilation and automatic memory management (garbage collection).

9. **Distributed:** Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

10. **Compiled and Interpreted:** Java can be considered both a compiled and an interpreted language because its source code is first compiled into a binary byte-code. This byte-code runs on the Java Virtual Machine (JVM), which is usually a software-based interpreter.

Overall, Java is a versatile, reliable, and powerful programming language that is used to develop a wide range of applications, from small applets to large enterprise systems.

1.3 JAVA VIRTUAL MACHINE AND BYTE CODE

The Java Virtual Machine (JVM) is a key component of the Java platform. It is a software that provides an environment in which Java bytecode can be executed. The JVM interprets the bytecode and translates it into machine language that can be executed by the computer's processor.

Java bytecode is a highly optimized set of instructions that is produced when Java source code is compiled. It is a platform-independent format that can be executed on any system that has a JVM installed. This means that a Java program can be developed on one platform and then run on any other platform without modification.

The Java bytecode is designed to be highly optimized for execution on the JVM. It is a low-level representation of the original Java source code, and includes many optimizations such as constant folding, dead code elimination, and method in-lining. These optimizations help to improve the performance of Java programs.

The JVM is responsible for interpreting and executing the bytecode. It provides a number of important features, including memory management, garbage collection, and security. The JVM also includes a just-in-time (JIT) compiler, which can dynamically compile frequently used bytecode into machine code for faster execution.

Overall, the JVM and bytecode are important components of the Java platform. They provide a platform-independent way to execute Java programs, while also offering a high degree of performance, security, and reliability.

1.4 TYPES OF JAVA PROGRAM

There are several types of Java programs that can be developed, depending on the requirements of the project. Here are some of the most common types of Java programs:

1. **Console Programs:** These are text-based programs that run in a command-line interface (CLI). They are often used for simple tasks such as data input/output, system administration, or automation.

2. **Desktop Applications:** These are graphical user interface (GUI) programs that run on a desktop computer. They can be used for a wide range of purposes, such as word processing, video editing, or image manipulation.

3. **Web Applications:** These are server-side programs that run on a web server and provide services to clients via a web browser. They can be used for a wide range of purposes, such as e-commerce, social media, or content management.

4. **Mobile Applications:** These are programs that run on mobile devices such as smartphones and tablets. They can be used for a wide range of purposes, such as gaming, communication, or productivity.

5. **Applets:** These are small Java programs that run within a web browser. They are often used for interactive web applications, such as games or multimedia presentations.

6. **Enterprise Applications**: These are large-scale programs that are designed to run within a corporate environment. They can be used for a wide range of purposes, such as customer relationship management, supply chain management, or financial management.

1.5 BASIC CONCEPT OF OOP

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code to manipulate that data. The basic concept of OOP revolves around four principles, known as the "four pillars of OOP":

- **Encapsulation:** Encapsulation in Java is a process of wrapping code and data together into a single unit, for example, a capsule which is mixed of several medicines.

- **Inheritance:** Inheritance is the process of creating new classes by inheriting properties and methods from existing classes. This allows the new classes to reuse code and behavior from the existing classes, reducing code duplication and improving code maintainability.

- **Polymorphism:** Polymorphism is the ability of objects to take on multiple forms. This allows objects to behave in different ways depending on the context in which they are used. Polymorphism is achieved through method overriding and method overloading.

- **Abstraction:** Abstraction is a process of hiding the implementation details and showing only functionality to the user. Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Overall, the four pillars of OOP provide a powerful and flexible way of building software systems that are modular, reusable, and easy to maintain. By encapsulating data, using inheritance and polymorphism to reduce code duplication, and abstracting away unnecessary details, OOP provides a powerful set of tools for building complex systems in a clear and efficient way.

1.6 PROCEDURE ORIENTED V/S OBJECT ORIENTED

Procedure-Oriented Programming (POP) and Object-Oriented Programming (OOP) are two programming paradigms that differ in several ways. Here are some of the key differences between the two:

1. **Data and behavior:** In POP, data and behavior are separate concepts, with data stored in global variables and behavior defined in procedures. In OOP, data and behavior are combined into objects, which encapsulate both.

2. **Encapsulation:** Encapsulation is a key feature of OOP that allows objects to hide their internal state and expose well-defined interfaces. This helps to protect the object's data and ensures that the object's behavior remains consistent. POP does not provide the same level of encapsulation.

3. **Inheritance:** Inheritance is a powerful feature of OOP that allows objects to inherit properties and behavior from parent classes. This can help to reduce code duplication and improve code maintainability. POP does not provide a native way to implement inheritance.

4. **Polymorphism:** Polymorphism is another powerful feature of OOP that allows objects to take on multiple forms. This allows objects to behave in different ways depending on the context in which they are used. POP does not provide a native way to implement polymorphism.

5. **Modularity:** OOP provides a more modular approach to programming, with code organized into objects and classes. This makes it easier to reuse code and maintain large programs. In POP, code is organized into functions or procedures, which can be harder to reuse and maintain in large programs.

6. **Scalability:** OOP is generally more scalable than POP, as it allows for code to be easily extended and modified over time. In contrast, POP can become unwieldy and difficult to maintain as the size of a program grows.

Overall, while POP and OOP have their own strengths and weaknesses, OOP is generally considered to be a more powerful and flexible approach to programming, especially for large and complex systems.

1.7 BASIC PROGRAM OF JAVA

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

- **public:** This is an access modifier that makes the class accessible to other classes in the same package or other packages.
- **class:** This keyword is used to define a new class.
- **HelloWorld:** This is the name of the class we're defining.
- **{}:** This block of code contains the class definition.
- **public:** This is an access modifier that makes the method accessible to other classes in the same package or other packages.
- **static**: This keyword makes the method a class method that can be called without creating an instance of the class.
- **void:** This keyword specifies that the method doesn't return a value.

- **main:** This is the name of the method. It's the entry point of the program and is executed when the program is run.
- **(String[] args):** This is the method's argument list. In this case, it takes an array of strings as an argument.
- **{}:** This block of code contains the method definition.
- **System.out.println("Hello, World!"); :** This line of code prints "Hello, World!" to the console. System.out is a predefined output stream in Java, and println() is a method that prints a string to the console and adds a newline character at the end.

1.8 BASIC DATA TYPES

Java has several primitive data types that represent the basic building blocks of data in the language. These include:

1. **byte:** A byte is a signed 8-bit integer, with a range of -128 to 127.
   Example:byte b = 100;
2. **short:** A short is a signed 16-bit integer, with a range of -32,768 to 32,767.
   Example:short s = 5000;
3. **int:** An int is a signed 32-bit integer, with a range of -2,147,483,648 to 2,147,483,647.
   Example:int i = 100000;
4. **long:** A long is a signed 64-bit integer, with a range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
   Example:long l = 1234567890L;
5. **float:** A float is a single-precision 32-bit floating-point number, with a range of approximately 1.4E-45 to 3.4E+38, and a precision of about 7 decimal digits.
   Example:float f = 3.14f;
6. **double:** A double is a double-precision 64-bit floating-point number, with a range of approximately 4.9E-324 to 1.8E+308, and a precision of about 15 decimal digits.
   Example:double d = 3.14159;
7. **boolean:** A boolean represents a logical value, either true or false.
   Example:char c = 'A';
8. **char:** A char represents a single character in Unicode, with a range of '\u0000' (or 0) to '\uffff' (or 65,535).
   Example:boolean b1 = true;
   boolean b2 = false;

In addition to these primitive data types, Java also has a rich set of reference data types, such as classes, interfaces, and arrays, which are built using the primitive data types.

Classes: A class is a blueprint or template for creating objects that encapsulate data and behavior. A class can contain fields (variables) to store data, and methods (functions) to perform operations on that data. Objects are instances of a class, and each object has its own set of values for the fields defined in the class. Here's an example of a simple class definition in Java:

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

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

public String getName() {
  return name;
  }

  public int getAge() {
   return age;
  }
}
```

In this example, the Person class has two fields, name and age, and a constructor that takes arguments to initialize those fields. It also has two methods, getName() and getAge(), which return the values of those fields.

Interfaces: An interface is a collection of abstract methods that define a set of behaviors that a class can implement. In Java, interfaces are used to define contracts between different classes or components in a system. A class can implement one or more interfaces, and by doing so, it agrees to provide implementations for all the methods defined in those interfaces. Here's an example of a simple interface definition in Java:

```
public interface Drawable {
 public void draw();
}
```

In this example, the Drawable interface defines a single abstract method, draw(), which has no implementation. A class that implements this interface must provide an implementation for the draw() method.

Arrays: An array is a collection of elements of the same data type, stored in contiguous memory locations. In Java, arrays are used to store and manipulate collections of data. Arrays can be of any data type, including primitive types and objects. Here's an example of how to create and use an array of integers in Java:

```
int[] numbers = {1, 2, 3, 4, 5};
System.out.println(numbers[0]);
System.out.println(numbers[2]);
numbers[3] = 10;
System.out.println(numbers[3]);
```

In this example, we create an array of integers and initialize it with some values. We then access individual elements of the array using their index, and modify the value of one of the elements.

1.9 OPERATORS

In Java, operators are symbols that perform operations on one or more operands (values, variables, or expressions). There are several types of operators in Java, including:

Arithmetic operators: Arithmetic operators are used to perform arithmetic operations on numerical values. The basic arithmetic operators in Java are:

+ (addition)
- (subtraction)
* (multiplication)
/ (division)
% (modulus)

EXAMPLE:

```
int a = 10;
int b = 5;

// addition
```

```
int sum = a + b; // sum = 15

// subtraction
int difference = a - b; // difference = 5

// multiplication
int product = a * b; // product = 50

// division
int quotient = a / b; // quotient = 2

// modulus
int remainder = a % b; // remainder = 0
```

Assignment operators: Assignment operators are used to assign values to variables. The basic assignment operator in Java is =. There are also compound assignment operators that combine arithmetic operations with assignment, such as +=, -=, *=, /=, and %=.

EXAMPLE:

```
int a = 10;
int b = 5;

// basic assignment
a = b; // a = 5

// compound assignment
a += b; // equivalent to a = a + b; a = 10 + 5 = 15
b -= a; // equivalent to b = b - a; b = 5 - 15 = -10
```

Comparison operators: Comparison operators are used to compare two values and return a boolean value indicating the result. The basic comparison operators in Java are:

== (equal to)

!= (not equal to)

< (less than)

> (greater than)

<= (less than or equal to)

>= (greater than or equal to)

EXAMPLE:

```
int a = 10;
int b = 5;

// equal to
boolean isEqual = (a == b); // isEqual = false

// not equal to
boolean notEqual = (a != b); // notEqual = true

// greater than
boolean greaterThan = (a > b); // greaterThan = true

// less than or equal to
boolean lessThanOrEqual = (a <= b); // lessThanOrEqual = false
```

Logical operators: Logical operators are used to combine boolean values and return a boolean result. The basic logical operators in Java are:

&& (logical AND)

|| (logical OR)

! (logical NOT)

EXAMPLE:

```
boolean x = true;
boolean y = false;

// logical AND
boolean andResult = (x && y); // andResult = false

// logical OR
boolean orResult = (x || y); // orResult = true

// logical NOT
boolean notResult = !x; // notResult = false
```

Bitwise operators: Bitwise operators are used to perform operations on the binary representations of numerical values. The basic bitwise operators in Java are:

& (bitwise AND)

| (bitwise OR)

^ (bitwise XOR)

~ (bitwise NOT)

<< (left shift)

>> (right shift)

>>> (unsigned right shift)

EXAMPLE:

```
int a = 60; // 0011 1100
int b = 13; // 0000 1101

// bitwise AND
int andResult = a & b; // andResult = 0000 1100

// bitwise OR
int orResult = a | b; // orResult = 0011 1101

// bitwise XOR
int xorResult = a ^ b; // xorResult = 0011 0001

// bitwise NOT
int notResult = ~a;
// notResult = 1100 0011

// left shift
int leftShiftResult = a << 2;
 // leftShiftResult = 1111 0000

// right shift
int rightShiftResult = a >> 2;
// rightShiftResult = 0000 1111

// unsigned right shift
int unsignedRightShiftResult = a >>> 2;
// unsignedRightShiftResult = 0000 1111
```

Conditional operator: The conditional operator (also known as the ternary operator) is a shorthand way of writing an if-else statement. It has the following syntax:

condition ? value1 : value2

If the condition is true, the operator returns value1, otherwise it returns value2.

EXAMPLE:

```
int a = 10;
int b = 5;

// conditional operator
int max = (a > b) ? a : b; // max = 10
```

Increment or Decrement Operator: The Java increment or decrement operators require only one operand. Increment or decrement operators are used to perform increment and decrement operations.

Example :

```
int x=10;
x++ // 10 (11)
++x // 12
x-- //12 (11)
--x //10
```

UNIT-2

CONTROL FLOW AND ARRAY

2.1 VARIABLE AND TYPES OF VARIABLE

Variable is a named storage location that holds a value of a particular type. Variables are used to store and manipulate data within a program. Java variables can be classified into different types based on their characteristics and usage.

Here are the common types of variables in Java:

1.Local Variables:

Local variables are declared within a method, constructor, or a block of code.

They are accessible only within the scope where they are declared.

Local variables must be initialized before they are used.

Their lifespan is limited to the block in which they are declared.

EXAMPLE:

```
public class LocalVariableExample
{
  public static void main(String[] args)
 {
  int age = 25; // local variable
System.out.println("Age: " + age);
  }
}
```

2.Instance Variables (Non-Static Variables):

Instance variables are declared within a class but outside of any method, constructor, or block.

Each instance of a class has its own copy of instance variables.

They are initialized with default values if not explicitly assigned.

Instance variables are accessible throughout the class and can be accessed using the object of the class.

EXAMPLE:

```
public class InstanceVariableExample
{
String name; // instance variable


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

```
        InstanceVariableExample obj = new InstanceVariableExample();
        obj.name = "John"; // accessing instance variable using object
        System.out.println("Name: " + obj.name);
    }
}
```

3.Class Variables (Static Variables):

Class variables are declared with the static keyword within a class but outside of any method, constructor, or block.

Unlike instance variables, there is only one copy of each class variable that is shared among all instances of the class.

Class variables are initialized with default values if not explicitly assigned.

They can be accessed using the class name itself, without creating an instance of the class.

EXAMPLE:

```
public class ClassVariableExample
{
 static int count; // class variable

 public static void main(String[] args)
 {
ClassVariableExample.count = 10; // accessing class variable using class name
System.out.println("Count: " + ClassVariableExample.count);
    }
}
```

In addition to the above classifications, variables can also be categorized based on their data types:

Primitive Variables: These variables store simple data types such as int, double, boolean, etc. They hold the actual value.

EXAMPLE:

```
public class PrimitiveVariableExample
{
   public static void main(String[] args)
 {
 int age = 25; // primitive variable
 double height = 1.75;
```

```
boolean isStudent = true;


System.out.println("Age: " + age);
System.out.println("Height: " + height);
System.out.println("Is Student: " + isStudent);
 }
}
```

Reference Variables: These variables hold references to objects in memory rather than the actual object itself. They are used for accessing object properties and invoking methods.

EXAMPLE:

```
public class ReferenceVariableExample
{
 public static void main(String[] args)
{
String name = "John"; // reference variable
int[] numbers = {1, 2, 3, 4, 5};


System.out.println("Name: " + name);
  System.out.println("Numbers: " + Arrays.toString(numbers));
  }
}
```

2.2 TYPE CASTING AND CONVERSION

2.2.1 TYPE CASTING

Type casting refers to the process of converting a value from one data type to another. Java supports two types of casting: implicit casting (widening) and explicit casting (narrowing).

1.Implicit Casting (Widening):

This type of casting takes place when two data types are automatically converted. It is also known as Implicit Conversion. This happens when the two data types are compatible and also when we assign the value of a smaller data type to a larger data type.

EXAMPLE:

```
public class ImplicitCastingExample
{
   public static void main(String[] args) {
  int myInt = 10;
```

```
 double myDouble = myInt; // Implicit casting from int to double
System.out.println("myInt: " + myInt);
System.out.println("myDouble: " + myDouble);
  }
}
```

In this example, the **myInt** variable of type **int** is implicitly casted to a double when assigned to the **myDouble** variable. This is allowed because a double can hold a larger range of values than an int.

2.Explicit Casting (Narrowing):

The process of conversion of higher data type to lower data type is known as narrowing typecasting. It is not done automatically by Java but needs to be explicitly done by the programmer, which is why it is also called explicit typecasting.

EXAMPLE:

```
public class ExplicitCastingExample
{
 public static void main(String[] args) {
 double myDouble = 10.5;
 int myInt = (int) myDouble; // Explicit casting from double to int
 System.out.println("myDouble: " + myDouble);
System.out.println("myInt: " + myInt);
  }
}
```

In this example, the **myDouble** variable of type **double** is explicitly casted to an **int** using the **(int)** notation. This is allowed, but it may result in the loss of precision, as the decimal portion of the **double** value is truncated.

2.2.2 CONVERSION

Type conversion is simply the process of converting data of one data type into another. This process is known as type conversion, typecasting, or even type coercion. The Java programming language allows programmers to convert both primitive as well as reference data types.

Java provides several methods to convert values between different data types.

Here are a few examples:

1.Converting String to int:

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

```
 String numberString = "123";
int number = Integer.parseInt(numberString);
System.out.println("Number: " + number);
 }
}
```

2.Converting int to String:

```
public class IntToStringExample
{
 public static void main(String[] args)
{
   int number = 123;
 String numberString = String.valueOf(number);
System.out.println("Number: " + numberString);
 }
}
```

3.Converting String to double:

```
public class StringToDoubleExample
{
  public static void main(String[] args)
{
  String numberString = "10.5";
 double number = Double.parseDouble(numberString);
 System.out.println("Number: " + number);
  }
}
```

2.3 WRAPPER CLASS

Wrapper classes are used to provide a way to treat primitive data types as objects. Each primitive data type has a corresponding wrapper class in Java. Wrapper classes are mainly used for converting primitive types into objects, enabling them to be used in scenarios that require objects.

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Int

long	Long
float	Float
double	Double

2.3.1 Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Wrapper class Example: Primitive to Wrapper:

```
public class WrapperExample1
{
public static void main(String args[])
{
//Converting int into Integer
int a=20;
Integer i=Integer.valueOf(a);//converting int into Integer explicitly
Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally

System.out.println(a+" "+i+" "+j);
}}
```

Output:

20 20 20

2.3.2 Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing.

Wrapper class Example: Wrapper to Primitive

```
public class WrapperExample2
{
public static void main(String args[])
{
//Converting Integer to int
Integer a=new Integer(3);
int i=a.intValue();//converting Integer to int explicitly
int j=a;//unboxing, now compiler will write a.intValue() internally
```

```
System.out.println(a+" "+i+" "+j);
}}
```

Output:

3 3 3

EXAMPLE:

```
public class WrapperExample
 {
  private int value;

 public WrapperExample(int value) {
this.value = value;
 }
    public int getValue() {
  return value;
 }
  public void setValue(int value) {
    this.value = value;
  }
 public static void main(String[] args) {
WrapperExample wrapper = new WrapperExample(10);
  System.out.println("Initial value: " + wrapper.getValue());

wrapper.setValue(20);
   System.out.println("Updated value: " + wrapper.getValue());
 }
}
```

In this example, we have a **WrapperExample** class that acts as a wrapper for an integer value. It has a private instance variable **value** which holds the wrapped value.

The class provides getter and setter methods (**getValue**() and **setValue**()) to access and modify the wrapped value. In the **main()** method, we create an instance of **WrapperExample**, set an initial value of 10, and then update it to 20. Finally, we print the initial and updated values to the console.

2.4 DECISION AND CONTROL STATEMENTS

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

Java provides three types of control flow statements.

1.Decision Making statements

- if statements
- switch statement

2.Loop statements

- do while loop
- while loop
- for loop
- for-each loop

3.Jump statements

- break statement
- continue statement

1.Decision Making statements

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

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

Simple if statement:

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

Syntax:

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

EXAMPLE:

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

Output:

x + y is greater than 20

if-else statement

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

Syntax:

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

EXAMPLE:

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

```
}
```

Output:

x + y is greater than 20

if-else-if ladder:

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

Syntax:

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

EXAMPLE:

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

Output:

Delhi

Nested if-statement:

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

Syntax:

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

EXAMPLE:

```
public class Student {
public static void main(String[] args) {
String address = "Delhi, India";

if(address.endsWith("India")) {
if(address.contains("Meerut")) {
System.out.println("Your city is Meerut");
}else if(address.contains("Noida")) {
System.out.println("Your city is Noida");
}else {
System.out.println(address.split(",")[0]);
}
}else {
System.out.println("You are not living in India");
}
}  }
```

Output:

Delhi

Switch Statement:

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

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied. It is optional, if not used, next case is executed.
- While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

Syntax:

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

EXAMPLE:

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

```
case 1:
System.out.println("number is 1");
break;
default:
System.out.println(num);
}
}
}
```

Output:

2

2. Loop statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

- for loop
- while loop
- do-while loop

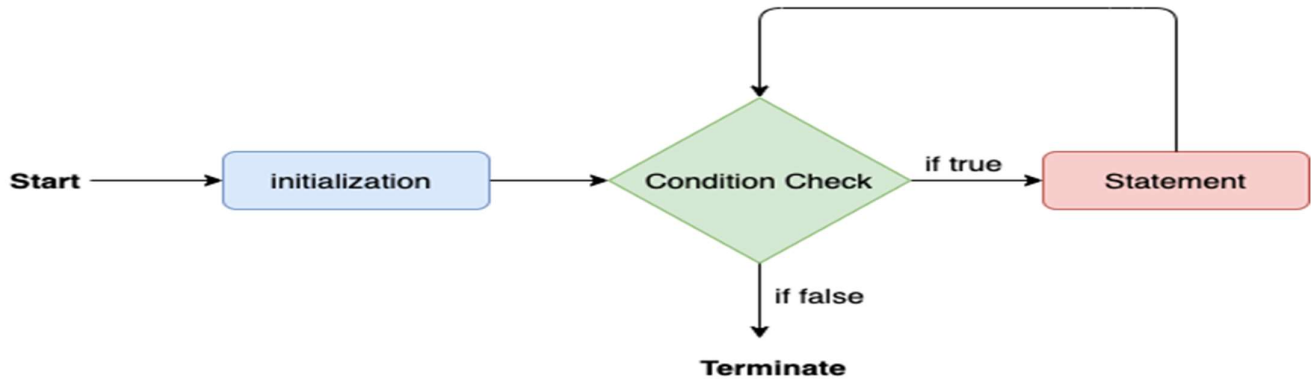
for loop

for loop is similar to C and C++. It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

Syntax:

```
for(initialization, condition, increment/decrement) {
//block of statements
}
```

The flow chart for the for-loop is given below.



EXAMPLE:

```
public class Calculattion {
public static void main(String[] args) {
// TODO Auto-generated method stub
int sum = 0;
for(int j = 1; j<=10; j++) {
sum = sum + j;
}
System.out.println("The sum of first 10 natural numbers is " + sum);
}
}
```

Output:

The sum of first 10 natural numbers is 55

for-each loop

Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable.

Syntax:

```
for(data_type var : array_name/collection_name){
//statements
}
```

EXAMPLE:

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
String[] names = {"Java","C","C++","Python","JavaScript"};
System.out.println("Printing the content of the array names:\n");
for(String name:names) {
System.out.println(name);
}
}
}
```

Output:

Printing the content of the array names:

Java

C

C++

Python

JavaScript

while loop

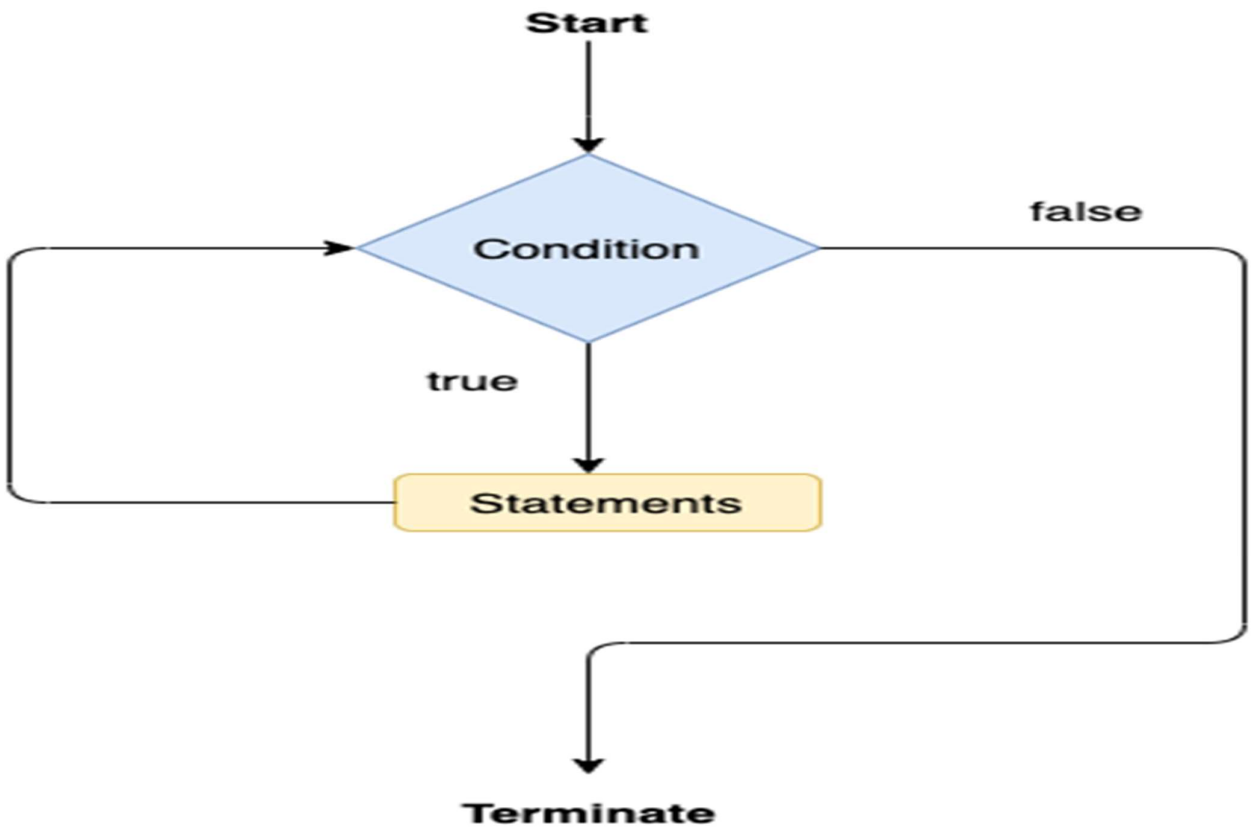
The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

Syntax:

```
while(condition){
//looping statements
}
```

The flow chart for the while loop is given in the following image.



EXAMPLE:

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
while(i<=10) {
System.out.println(i);
i = i + 2;
}
 }
 }
```

Output:

Printing the list of first 10 even numbers

0

2

4

6

8

10

do-while loop

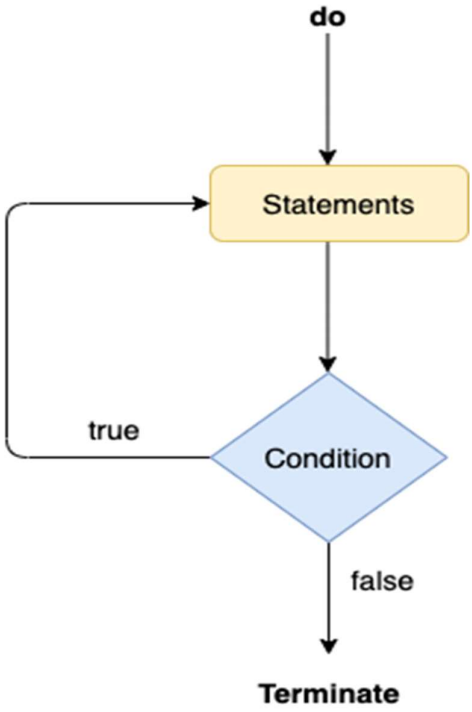
The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

It is also known as the exit-controlled loop since the condition is not checked in advance.

Syntax:

```
do
{
//statements
} while (condition);
```

The flow chart of the do-while loop is given in the following image.



EXAMPLE:

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
do {
System.out.println(i);
i = i + 2;
}while(i<=10);
}  }
```

Output:

Printing the list of first 10 even numbers

0

2

4

6

8

10

3. Jump statements

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

Break Statement

As the name suggests, the break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop.

The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

EXAMPLE:

```
public class BreakExample {


public static void main(String[] args) {
// TODO Auto-generated method stub
for(int i = 0; i<= 10; i++) {
System.out.println(i);
if(i==6) {
break;
} } } }
```

Output:

0

1

2

3

4

5

6

Continue Statement

Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

EXAMPLE:

```
public class ContinueExample {
```

```
public static void main(String[] args) {
// TODO Auto-generated method stub

for(int i = 0; i<= 2; i++) {

for (int j = i; j<=5; j++) {

if(j == 4) {
continue;
}
System.out.println(j);
}
}
}
}
```

Output:

0
1
2
3
5
1
2
3
5
2
3
5

2.5 ARRAY AND TYPES OF ARRAY

Array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

Single Dimensional Array

A single-dimensional array is the most common type of array. It stores elements in a linear fashion, where each element is accessed using its index.

Syntax:

dataType[] arr; (or)

dataType []arr; (or)

dataType arr[];

EXAMPLE:

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

Output:

10

20

70

40

50

Multidimensional Array

A multidimensional array is an array of arrays. It can have two or more dimensions and is represented as a matrix-like structure. Commonly used multidimensional arrays include 2D arrays and 3D arrays.

Syntax:

```
dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];
```

EXAMPLE:

```
class Testarray3{
public static void main(String args[]){
//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
//printing 2D array
for(int i=0;i<3;i++){
for(int j=0;j<3;j++){
  System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}
}}
```

Output:

1 2 3

2 4 5

4 4 5

Dynamic Array (ArrayList):

Unlike regular arrays, the ArrayList class provides dynamic arrays that can grow or shrink at runtime. It is part of the Java Collections Framework and offers additional functionalities such as adding, removing, and searching elements.

EXAMPLE:

```
import java.util.ArrayList;


// Declaration and initialization of an ArrayList
ArrayList<Integer> numbers = new ArrayList< Integer >();


// Adding elements to the ArrayList
```

```
numbers.add(1);
numbers.add(2);
numbers.add(3);

// Accessing elements of the ArrayList
int firstElement = numbers.get(0);  // Accessing the first element
int size = numbers.size();          // Retrieving the size of the ArrayList
```

Other Types of Arrays:

Java also supports arrays of other data types such as boolean, char, double, etc.

EXAMPLE:

```
// Declaration and initialization of a char array
char[] characters = {'a', 'b', 'c'};

// Declaration and initialization of a boolean array
boolean[] flags = {true, false, true};

// Declaration and initialization of a double array
double[] decimals = {1.2, 3.4, 5.6};
```

Arrays in Java have a fixed size, meaning you need to specify the size when declaring them (except for ArrayList, which dynamically adjusts its size). However, you can use other data structures like ArrayList or LinkedList if you need a resizable collection.

2.6 GARBAGE COLLECTION

Garbage collection in Java is the process of automatically reclaiming memory occupied by objects that are no longer in use. It helps manage memory efficiently by freeing up resources that are no longer needed by the program.

Here's an overview of how garbage collection works in Java:

Object Lifecycle:

In Java, objects are created using the new keyword and reside in the heap memory. When an object is no longer referenced by any part of the program, it becomes eligible for garbage collection.

How can an object be unreferenced?

There are many ways:

1) By nulling a reference:

```
Employee e=new Employee();
e=null;
```

2) By assigning a reference to another:

Employee e1=new Employee();

Employee e2=new Employee();

e1=e2;//now the first object referred by e1 is available for garbage collection

3) By anonymous object:

new Employee();

EXAMPLE:

```
public class TestGarbage1{
 public void finalize(){System.out.println("object is garbage collected");}
public static void main(String args[]){
TestGarbage1 s1=new TestGarbage1();
TestGarbage1 s2=new TestGarbage1();
  s1=null;
 s2=null;
System.gc();
 }
}
```

Output:

object is garbage collected

object is garbage collected

2.7 COMMAND LINE ARGUMENTS

Command line arguments in Java are the parameters that are passed to a Java program when it is executed from the command line or terminal. These arguments provide a way to customize the behavior of the program without modifying the source code.

Here's how you can access command line arguments in Java:

Command Line Argument Format:

In Java, command line arguments are passed as strings, separated by spaces. Each argument is treated as a separate element in the args array.

Accessing Command Line Arguments:

The command line arguments are available to the Java program through the args parameter of the main() method. The args parameter is an array of strings.

```
public class CommandLineArguments {
  public static void main(String[] args) {
  // Accessing command line arguments
        for (int i = 0; i <args.length; i++) {
   System.out.println("Argument " + i + ": " + args[i]);}}}
```

In the above example, the main() method accepts an array of strings named args. The program uses a for loop to iterate over the args array and prints each argument along with its index.

Running a Java Program with Command Line Arguments:

To run a Java program with command line arguments, open the command prompt or terminal, navigate to the directory containing the compiled .class file, and use the java command followed by the program name and the arguments.

java CommandLineArguments arg1 arg2 arg3

In the above command, CommandLineArguments is the name of the Java program, and arg1, arg2, and arg3 are the command line arguments that will be passed to the program.

Example of command-line argument

```
class CommandLineExample{
public static void main(String args[]){
System.out.println("Your first argument is: "+args[0]);
}  }
```

Output:

Your first argument is: sonoo

UNIT-3

OBJECT ORIENTED PROGRAMMING CONCEPTS

3.1 CLASS AND OBJECT

In the real world, you'll often find many individual objects all of the same kind.

All the objects that have similar properties and similar behavior are grouped together to form a class.

In other words we can say that a class is a user defined data type and objects are the instance variables of class. There may be thousands of other bicycles in existence; all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycls an instance of the class of objects known as bicycles. A class is the blueprint from which individual objects are created.

The following Bicycle class is one possible implementation of a bicycle:

```
class Bicycle {

int cadence = 0;

int speed = 0;

int gear = 1;

void changeCadence(int newValue) {

cadence = newValue;

}

void changeGear(int newValue) {

gear = newValue;

}

void printStates() { System.out.println("cadence:"+cadence+"speed:"+speed+" gear:"+gear);

} }
```

The fields cadence, speed, and gear represent the object's state, and the methods (changeCadence, changeGear, speedUp etc.) define its interaction with the outside world.

You may have noticed that the Bicycle class does not contain a main method. That's because it's not a complete application; it's just the blueprint for bicycles that might be used in an application.

It is possible to define a class within another class; such classes are known as nested classes. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B is known to A, but not outside of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.

There are two types of nested classes: static and non-static. A static nested class is one which has the static modifier applied. Because it is static modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

The most important type of nested class is the inner class is the inner class. An inner class is a non-static nested class. It has access to al of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do. Thus, an inner class is fully within the scope of its enclosing class.

The following program illustrates how to define and use an inner class. The class named Outer has one instance variable named outer_x, one instance method named test(), and defines one inner class called Inner.

```
class Outer {
int outer_x = 100;
void test() {
Inner inner = new Inner();
inner.display();
}
class Inner {
void display() { System.out.println("display : outer_x =" + outer_x);
}
}
}
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
outer.test();
}
}
```

Output from this application is shown here:

display: outer_x = 100

In the program, an inner class named Inner is defined within the scope of class Outer. Therefore, any code in class Inner can directly access the variable outer_x. an instance method named display() is defined inside Inner. This method displays outer_x on the standard output stream. The main() method of InnerClassDemo creates an instance of class Inner and the display() method is called.

It is important to realize that class Inner is known only within the scope of class Outer. The java compiler generates an error message if any code outside of class Outer attempts to instantiate class Inner. Generalizing, a nested class is no different than any other program element: it is known only within its enclosing scope.

As explained, an inner class has access to all of the members of its enclosing class, but the reverse is not true. Members of the inner class are known only within the scope of the inner class and may not be used by the outer class.

3.2 CONSTRUCTOR AND TYPES OF CONSTRUCTORS

In Java, a constructor is a special method that is used to initialize objects of a class. It is called automatically when an object is created using the **new** keyword. Constructors have the same name as the class and do not have a return type, not even **void.**

3.2.1 TYPES OF CONSTRUCTORS

1.Default Constructor:

A default constructor is provided by Java if you do not explicitly define any constructor in your class. It has no parameters and does not perform any initialization. Its purpose is to initialize the object with default values (e.g., setting numeric properties to 0 and reference properties to **null**).

EXAMPLE:

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

2. Parameterized Constructor:

A parameterized constructor is a constructor that takes one or more parameters. It allows you to initialize the object with specific values at the time of creation.

EXAMPLE:

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


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

In the example above, the Person class has a parameterized constructor that takes name and age as parameters and initializes the corresponding instance variables.

3.Copy Constructor:

A copy constructor is a constructor that creates a new object by copying the state of another object of the same class. It is useful when you want to create a new object with the same values as an existing object.

EXAMPLE:

```
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, the Student class has a copy constructor that takes another Student object as a parameter and copies its name and age values to create a new Student object.

3.3 METHOD AND METHOD OVERLOADING

Method is a collection of statements that are grouped together to perform a specific task. Methods are used to encapsulate code and provide reusability and modularity in your programs. They are defined inside classes and can be called to perform the actions they define.

Syntax:

```
modifier returnType methodName(parameterList) {
// Method body
}
```

Example:

```
public class Calculator {
 public int add(int num1, int num2) {
  return num1 + num2;
}
}
```

3.3.1 METHOD OVERLOADING

Method overloading is a feature in Java that allows you to define multiple methods with the same name but different parameters within the same class. The methods must have different parameter types, different numbers of parameters, or both.

USES OF METHOD OVERLOADING:

Method overloading when a couple of methods are needed with conceptually similar functionality with different parameters.

Memory can be saved by implementing method overloading.

EXAMPLE:

```
public class MathUtils {
  public int add(int num1, int num2) {
   return num1 + num2;
  }

  public double add(double num1, double num2) {
   return num1 + num2;
  }
}
```

3.4 THIS KEYWORD

'this' keyword is a reference to the current instance of a class. It is used within an instance method or constructor to refer to the object on which the method or constructor is being invoked**.**

Here are a few common uses of the **this** keyword:

1.Accessing Instance Variables:

You can use this to refer to the current object's instance variables when there is a naming conflict between instance variables and method parameters or local variables.

```
public class Person {
  private String name;

   public void setName(String name) {
  // Using "this" to refer to the instance variable
   this.name = name;
   }
}
```

In this example, the **this.name** refers to the instance variable name, while name alone refers to the method parameter. It helps differentiate between the two variables and assign the value to the instance variable.

2.Invoking Another Constructor:

In a constructor, you can use this to invoke another constructor of the same class. It is useful when you want to reuse the initialization logic of one constructor in another constructor.

```
public class Car {
 private String brand;
private String color;
 private int year;

 public Car(String brand, String color) {
this.brand = brand;
this.color = color;
  }

   public Car(String brand, String color, int year) {
this(brand, color);  // Invoking the two-argument constructor using "this"
this.year = year;
}
}
```

In this example, the second constructor invokes the first constructor using **this(brand, color).** It allows the common initialization code to be shared between the constructors.

3.Returning The Current Instance:

You can use this to return the current instance of the object from a method. It is commonly used for method chaining or fluent interfaces.

```
public class Person {
private String name;

 public Person withName(String name) {
  this.name = name;
return this;
  } }
```

In this example, the **withName** method sets the name and returns the current instance (**this**). It allows method calls to be chained together for a more concise and readable code.

The “this” keyword always refers to the current instance. □

The “this” keyword helps in removing the ambiguity of references and allows to refer to the current instance within the running program.

Key\word this is used with variables as "this. variable" and specifies the number of member variable of this instance of that class.

3.5 STATIC KEYWORD

The **static keyword** is used to declare members (variables and methods) that belong to the class itself rather than to instances (objects) of the class. It allows you to access these members without creating an instance of the class.

Here are some key uses of the **static** keyword:

1.Static Variables:

When a variable is declared as static, it is called a static variable or a class variable. The static variable is shared among all instances of the class. There is only one copy of the static variable that is shared across all objects of the class.

```
public class Counter {
private static int count; // static variable


  public Counter() {
  count++; // Accessing and modifying the static variable
  }


   public static void printCount() { // static method
   System.out.println("Count: " + count); // Accessing the static variable
}
}
```

In this example, the **count** variable is declared as **static**, meaning it is shared among all instances of the **Counter** class. Each time a new **Counter** object is created, the constructor is called and increments the **count** variable. The **printCount** method is also declared as **static** and can be called without creating an object.

2.Static Methods:

When a method is declared as static, it is called a static method. Static methods belong to the class rather than individual objects. They can be invoked directly on the class itself, without the need to create an instance.

```
public class MathUtils {
public static int add(int a, int b) { // static method
  return a + b; }
        }
```

In this example, the **add** method is declared as **static**. It can be called using the class name, without creating an instance of the **MathUtils** class.

```
int sum = MathUtils.add(5, 3);
System.out.println(sum);
```

3.Static Block:

A static block is a block of code enclosed in curly braces {} that is executed when the class is loaded into memory. It is used to initialize static variables or perform any other static initialization tasks.

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

  static {
   x = 10;
System.out.println("Static block executed");
}
}
```

In this example, the static block sets the value of the static variable **x** to 10. It is executed only once when the class is loaded, before any other static or instance members are accessed.

3.6 STRING CLASS AND ITS METHODS

The **String** class represents a sequence of characters. It is a built-in class and provides various methods to manipulate and work with **strings**.

Here are some commonly used methods of the **String** class:

1.Length: The **length**() method returns the length (number of characters) of a string.

```
String str = "Hello, World!";
int length = str.length();
```

Output: length = 13

2.Concatenation: The **concat()** method concatenates two strings and returns a new string.

```
String str1 = "Hello";
String str2 = "World";
String result = str1.concat(str2);
```

Output: result = "HelloWorld"

Alternatively, you can use the + operator for string concatenation.

```
String result = str1 + str2;
```

Output result = "HelloWorld"

3.Substring: The **substring()** method returns a substring of the original string based on the specified starting and ending indexes.

```
String str = "Hello, World!";
String substr = str.substring(7, 12);
```

Output: substr = "World"

4.Comparison: The **equals()** method checks if two strings have the same content.

```
String str1 = "Hello";
String str2 = "Hello";
boolean isEqual = str1.equals(str2);
```

Output: isEqual = true

The **equalsIgnoreCase()** method compares strings while ignoring case differences.

5.Conversion: The **toUpperCase**() and **toLowerCase**() methods convert the string to uppercase and lowercase, respectively.

```
String str = "Hello, World!";
String uppercase = str.toUpperCase();
String lowercase = str.toLowerCase();
```

Output: uppercase = "HELLO, WORLD!"

lowercase = "hello, world!"

6.Searching: The **indexOf()** method returns the index of the first occurrence of a specified substring within a string.

```
String str = "Hello, World!";
int index = str.indexOf("World");
```

Output: index = 7

7.Splitting: The **split()** method splits a string into an array of substrings based on a specified delimiter.

```
String str = "Hello,World,Java";
String[] parts = str.split(",");
```

Output: parts = ["Hello", "World", "Java"]

8.Replacement: The **replace()** method replaces all occurrences of a specified character or substring with another character or substring.

```
String str = "Hello, World!";
String replaced =str.replace("Hello", "Hi");
```

Output: replaced = "Hi, World!"

3.7 I/O CLASSES AND FILE HANDLING

I/O (Input/Output) classes are used for reading input from various sources and writing output to different destinations. These classes provide functionality for file handling, reading from and writing to streams, and interacting with the user via the console.

Here are some commonly used I/O classes and file handling operations in Java:

1.File class: The **File** class is used to represent files and directories. It provides methods to create, delete, rename, and query file properties.

```
import java.io.File;

// Creating a File object
File file = new File("path/to/file.txt");

// Checking if the file exists
boolean exists = file.exists();

// Creating a directory
File directory = new File("path/to/directory");
directory.mkdir();
```

2.Stream classes: Java provides different stream classes for reading and writing data. **InputStream** and **OutputStream** are used for reading and writing byte-level data, while Reader and Writer are used for character-level data.

```
import java.io.*;

// Reading from a file using FileInputStream and InputStreamReader
FileInputStream fis = new FileInputStream("file.txt");
InputStreamReader isr = new InputStreamReader(fis);
int data;
while ((data = isr.read()) != -1) {
  // Process the data
}

// Writing to a file using FileOutputStream and OutputStreamWriter
FileOutputStream fos = new FileOutputStream("file.txt");
OutputStreamWriter osw = new OutputStreamWriter(fos);
osw.write("Hello, World!");
osw.close();
```

3.BufferedReader and BufferedWriter: These classes provide buffered reading and writing operations, which can improve performance by reducing the number of I/O operations.

```
import java.io.*;
```

```
// Reading from a file using BufferedReader
FileReader fr = new FileReader("file.txt");
BufferedReader br = new BufferedReader(fr);
String line;
while ((line = br.readLine()) != null) {
 // Process the line
}

// Writing to a file using BufferedWriter
FileWriter fw = new FileWriter("file.txt");
BufferedWriter bw = new BufferedWriter(fw);
bw.write("Hello, World!");
bw.newLine(); // Writes a newline character
bw.close();
```

4.Scanner class: The Scanner class is used for reading user input from the console or parsing formatted data from different sources.

```
import java.util.Scanner;

// Reading user input from the console
Scanner scanner = new Scanner(System.in);
System.out.print("Enter your name: ");
String name = scanner.nextLine();
System.out.println("Hello, " + name);

// Parsing data from a file
File file = new File("data.txt");
Scanner fileScanner = new Scanner(file);
while (fileScanner.hasNext()) {
  String data = fileScanner.nextLine();
 // Process the data
}
fileScanner.close();
```

UNIT-4

INHERITANCE, PACKAGES AND INTERFACES

4.1 BASIC OF INHERITANCE

inheritance is a fundamental feature of object-oriented programming that allows you to create new classes based on existing classes. The class that is being extended is called the superclass or parent class, and the class that inherits from the superclass is called the subclass or child class. Inheritance promotes code reuse and supports the concept of "is-a" relationship between classes.

To define a subclass and establish inheritance, you use the **extends** keyword in the class declaration. The subclass inherits all the fields and methods (except for private members) of the superclass.

Here's an example to illustrate the basics of inheritance in Java:

```
// Superclass
class Animal {
  protected String name;

public Animal(String name) {
 this.name = name;
 }

   public void eat() {
System.out.println(name + " is eating.");
   }
}

// Subclass inheriting from Animal
class Dog extends Animal {
public Dog(String name) {
 super(name);  // Calling the superclass constructor
  }

   public void bark() {
   System.out.println(name + " is barking.");
 }
}
```

In this example, the **Animal** class serves as the superclass, and the **Dog** class is the subclass that extends the **Animal** class.

The **Dog** class inherits the name field and the **eat**() method from the **Animal** class. It also defines its own method, **bark(),** which is specific to dogs.

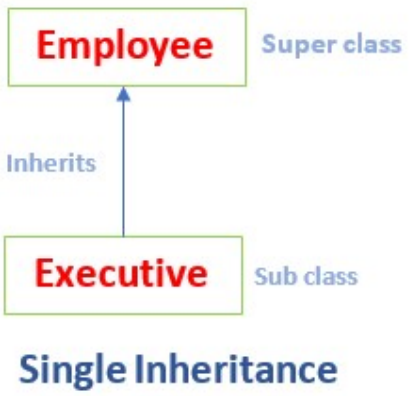
Key points to note about inheritance:

- Subclasses can access non-private members (fields and methods) of the superclass.
- Subclasses can add their own fields and methods or override the behavior of the superclass methods.
- Constructors are not inherited but can be called using the super() keyword to invoke the superclass constructor.
- Java supports single inheritance, which means a class can only extend one superclass. However, it allows for multiple levels of inheritance, where a subclass can itself become a superclass for another subclass.

4.2 TYPES OF INHERITANCE

There are several types of inheritance that you can utilize based on your program's requirements. The different types of inheritance are as follows:

1.Single Inheritance: In single inheritance, a subclass extends a single superclass. It is the most common type of inheritance.

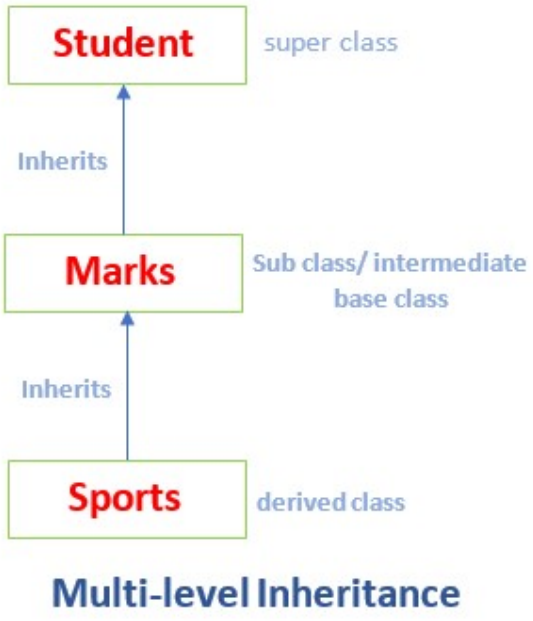


Single Inheritance

```
class Superclass {
 // Superclass members
}

class Subclass extends Superclass {
  // Subclass members
}
```

2.Multilevel Inheritance: In multilevel inheritance, a subclass becomes the superclass for another subclass, creating a chain of inheritance.




```
class Superclass {
  // Superclass members
}

class IntermediateSubclass extends Superclass {
        // Intermediate subclass members
}

class Subclass extends IntermediateSubclass {
  // Subclass members
}
```

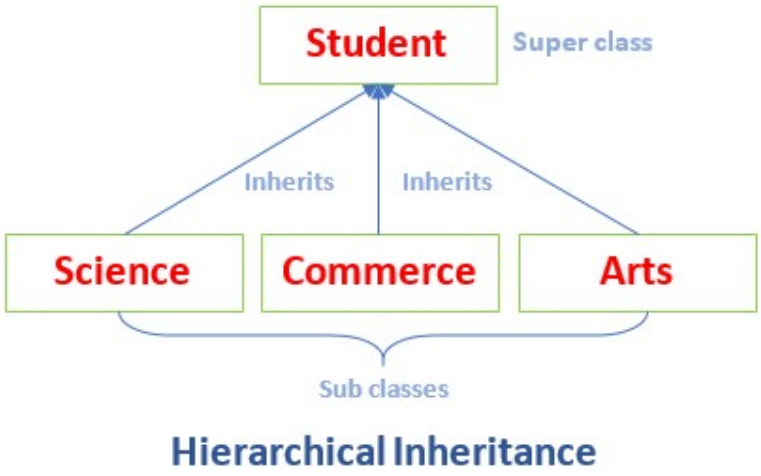
3.Hierarchical Inheritance: In hierarchical inheritance, multiple subclasses extend a single superclass.

```
class Superclass {
// Superclass members
}

class Subclass1 extends Superclass {
        // Subclass1 members
}

class Subclass2 extends Superclass {
```

```
// Subclass2 members
}
```



4.Multiple Inheritance (through Interfaces): Java does not support multiple inheritance of classes, but it allows multiple inheritance of interfaces. A class can implement multiple interfaces, inheriting their method signatures.

```
interface Interface1 {
// Interface1 members
}

interface Interface2 {
  // Interface2 members
}

class MyClass implements Interface1, Interface2 {
 // MyClass members
}
```

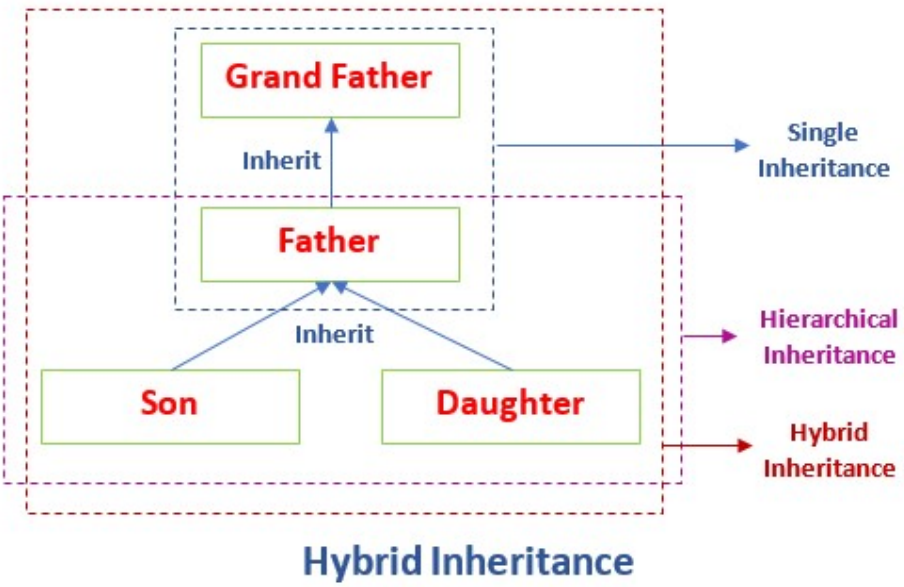
5.Hybrid Inheritance: Hybrid inheritance combines multiple types of inheritance, such as a combination of single inheritance, multilevel inheritance, and multiple inheritance.

```
class Superclass {
  // Superclass members
}

class IntermediateSubclass extends Superclass {
 // Intermediate subclass members
```

```
}

class Subclass extends IntermediateSubclass implements Interface1, Interface2 {
  // Subclass members
}
```



4.3 METHOD OVERRIDING

Method overriding in Java allows a subclass to provide a different implementation of a method that is already defined in its superclass. It is a feature of inheritance that promotes polymorphism, allowing objects of different subclasses to be treated uniformly through their common superclass.

To override a method in a subclass, the following conditions must be met:

- The method in the subclass must have the same name and parameters (method signature) as the method in the superclass.
- The method in the superclass must be marked as public, protected, or have default (package-private) access, to be accessible to the subclass.
- The return type of the method in the subclass must be the same or a subtype (covariant return type) of the return type in the superclass. In Java 5 and later versions, you can override a method with a covariant return type.

Example:

```
class Animal {
  public void makeSound() {
  System.out.println("Animal is making a sound.");
 }
}
class Dog extends Animal {
```

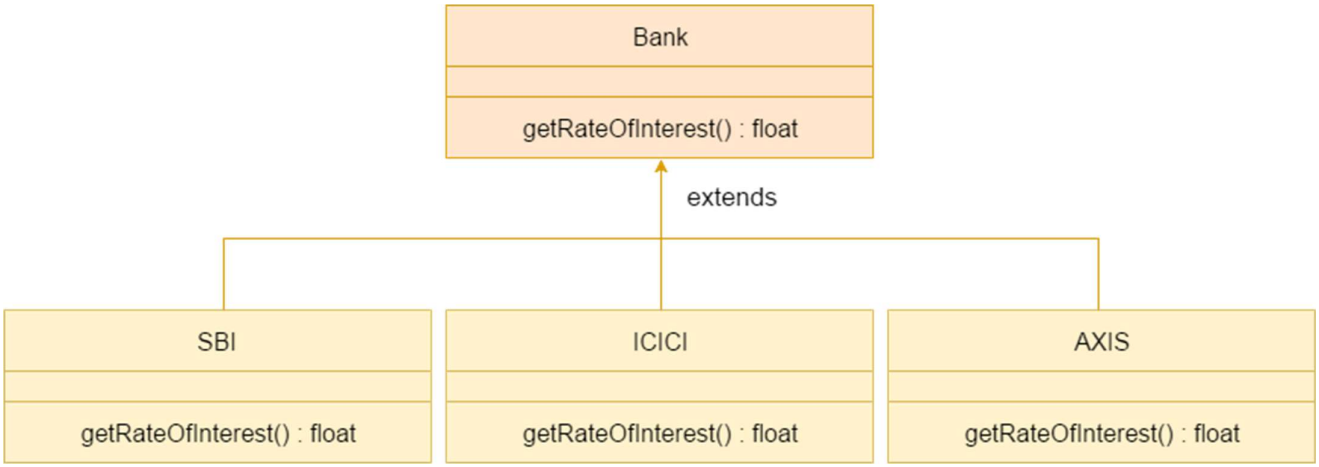
```
@Override
 public void makeSound() {
    System.out.println("Dog is barking.");
 }
}
```

In this example, the **Animal** class has a method called **makeSound**(). The **Dog** class extends the **Animal** class and overrides the **makeSound()** method to provide its own implementation.

When an overridden method is called on an object, the decision on which implementation to invoke is made at runtime based on the actual type of the object (dynamic method dispatch). This allows for polymorphism, where different subclasses can have their own specific implementation of the same method.

Method overriding allows you to provide specialized behavior in subclasses while still adhering to the common interface defined by the superclass. It is a powerful mechanism for achieving code reuse and flexibility in object-oriented programming.

Real Life Example Of Method Overriding:



4.4 SUPER KEYWORD

The **super keyword** is used to refer to the superclass or parent class of a subclass. It is commonly used in two scenarios:

Usage of Java super Keyword

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

1) Super Is Used To Refer Immediate Parent Class Instance Variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
class Animal{
```

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

Output:

black

white

2) Super Can Be Used To Invoke Parent Class Method:

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

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

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

Output:

eating...

barking...

3) Super Is Used ToInvoke Parent Class Constructor:

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

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

Output:

animal is created

dog is created

4.5 DYNAMIC METHOD DISPATCH

Dynamic method dispatch in Java is a mechanism that allows the selection of the appropriate method implementation at runtime, based on the actual type of the object rather than the reference type. It enables polymorphism, where objects of different subclasses can be treated uniformly through their common superclass.

Here's how dynamic method dispatch works in Java:

1.Inheritance Hierarchy: Create a class hierarchy with a common superclass and multiple subclasses that override a method.

```
class Animal {
 public void makeSound() {
```

```
  System.out.println("Animal is making a sound.");
  }
}

class Dog extends Animal {
  @Override
  public void makeSound() {
 System.out.println("Dog is barking.");
 }
}

class Cat extends Animal {
 @Override
 public void makeSound() {
 System.out.println("Cat is meowing.");
}
}
```

2.Object Creation: Create objects of the subclasses and assign them to variables of the superclass type.

```
Animal animal = new Animal();
Animal dog = new Dog();
Animal cat = new Cat();
```

3.Method Invocation: Invoke the overridden method on the objects.

```
animal.makeSound();  // Output: "Animal is making a sound."
dog.makeSound();     // Output: "Dog is barking."
cat.makeSound();     // Output: "Cat is meowing."
```

In this example, we have an **Animal** superclass and two subclasses, **Dog** and **Cat**, that override the **makeSound**() method. We create objects of both subclasses and assign them to variables of the superclass type. When we invoke the **makeSound**() method on these objects, the appropriate overridden method implementation is executed based on the actual type of the object. This is dynamic method dispatch.

At runtime, the JVM determines the actual type of the object being referred to (known as the runtime type), and then the corresponding method implementation is executed. This mechanism allows for treating objects of different subclasses uniformly through their common superclass, promoting code flexibility and polymorphic behavior.

Dynamic method dispatch is a fundamental concept in object-oriented programming and enables the principles of polymorphism and code reuse. It allows for writing more generic code that can be extended and specialized by subclasses while maintaining a consistent interface defined by the superclass.

4.6 FINAL KEYWORD

The final keyword is used to restrict the modification or inheritance of classes, methods, and variables. When applied to different elements, it provides different behaviors:

1.Final Classes: When a class is declared as **final**, it cannot be subclassed. It ensures that the class's implementation cannot be changed by any other class.

```
final class FinalClass {
 // Class members and implementation
}


// Error: Cannot inherit from final class
class Subclass extends FinalClass {
  // Subclass implementation
}
```

2.Final Methods: When a method is declared as **final**, it cannot be overridden by any subclass. This is useful when you want to enforce a specific implementation of a method in a class hierarchy.

```
        class Superclass {
  // Final method
          public final void finalMethod() {
 // Method implementation
  }
}


class Subclass extends Superclass {
  // Error: Cannot override final method
  public void finalMethod() {
    // Method implementation
  }}
```

3.Final Variables: When a variable is declared as **final**, its value cannot be modified once it is assigned. It becomes a constant, and the variable name is written in uppercase by convention.

```
class MyClass {
  // Final variable
public static final int MY_CONSTANT = 10;

  public void myMethod() {
    // Error: Cannot assign a value to final variable
   MY_CONSTANT = 20;
  }
}
```

4.Final Parameters: When a parameter is declared as **final** in a method, its value cannot be changed within the method. It is useful when you want to ensure that the parameter is not modified accidentally.

```
class MyClass {
public void myMethod(final int value) {
   // Error: Cannot assign a value to final parameter
  value = 10;
}
}
```

The **final** keyword provides immutability, security, and performance benefits. By marking a class, method, or variable as **final**, you can ensure that its behavior cannot be modified or overridden, protecting critical functionality or values.

It's important to note that using the **final** keyword should be done judiciously. Overusing it may limit the flexibility and extensibility of your code, so it's recommended to use it when necessary, such as for constants, non-overridable methods, or classes that should not be subclassed.

4.7 ABSTRACT CLASS AND METHOD

An abstract class is a class that cannot be instantiated and is meant to serve as a blueprint for subclasses. It is declared using the abstract keyword. Abstract classes can contain abstract methods, which are declared without an implementation and must be overridden by concrete subclasses.

Here are the key points about abstract classes and methods in Java:

1.Abstract Classes:

- An abstract class is declared using the abstract keyword.
- It may contain abstract methods, concrete methods, instance variables, and constructors.
- Abstract classes cannot be instantiated directly using the new keyword.
- Subclasses of an abstract class must either implement all the abstract methods or be declared as abstract themselves.

- Abstract classes can provide common functionality and define a common interface for their subclasses.

Example:

```
abstract class Shape {
  protected String color;

public Shape(String color) {
this.color = color;
 }

  public abstract double getArea(); // Abstract method

 public void display() {
   System.out.println("Color: " + color);
 }
}
```

2.Abstract Methods:

- An abstract method is declared using the abstract keyword and does not have an implementation.
- Abstract methods are meant to be overridden by concrete subclasses.
- Subclasses must provide an implementation for all the abstract methods inherited from the abstract class.
- Abstract methods do not have a body and end with a semicolon.

Example:

```
abstract class Shape {
 // ...

 public abstract double getArea(); // Abstract method

  // ...
}
```

To Use The Abstract Class And Override The Abstract Method:

```
class Rectangle extends Shape {
 private double length;
private double width;
```

```
  public Rectangle(String color, double length, double width) {
   super(color);
this.length = length;
this.width = width;
 }

    @Override
   public double getArea() {
 return length * width;
 }
}

class Circle extends Shape {
private double radius;

  public Circle(String color, double radius) {
    super(color);
this.radius = radius;
}

 @Override
  public double getArea() {
      return Math.PI * radius * radius;
  }
}
```

In this example, the Shape class is an abstract class that defines an abstract method **getArea**(). The Rectangle and Circle classes extend the Shape class and provide their own implementations for the **getArea**() method.

Abstract classes and methods are useful when you want to provide a common interface and some default behavior for related classes, while still allowing each class to have its specific implementation. They help enforce structure and contracts in your code and facilitate polymorphism.

4.8 INTERFACE

An interface is a reference type that defines a contract of methods that a class implementing the interface must adhere to. It provides a way to achieve abstraction and multiple inheritance of behavior.

1.Declaring Interfaces:

- An interface is declared using the interface keyword.
- It can contain method declarations (without implementations), constant fields, and default methods (with implementations) since Java 8.
- By convention, interface names should be nouns or noun phrases and are typically named using uppercase camel case**.**

```
interface Shape {
    void draw(); // Method declaration

    double getArea(); // Method declaration

    String COLOR = "Red"; // Constant field (implicitly public, static, and final)
}
```

2.Implementing Interfaces:

- A class implements an interface using the **implements** keyword.
- The class must provide implementations for all the methods declared in the interface.
- A class can implement multiple interfaces by separating them with commas.

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

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

    @Override
    public void draw() {
        System.out.println("Drawing a circle.");
    }

    @Override
    public double getArea() {
        return Math.PI * radius * radius;
```

```
    }
}
```

3.Interface Inheritance:

- An interface can extend one or more other interfaces using the extends keyword.
- When an interface extends another interface, it inherits the methods and constant fields of the parent interface.
- A class implementing a child interface must provide implementations for all the methods in the child interface as well as the parent interfaces.

```
interface Drawable {
    void draw();
}

interface Shape extends Drawable {
    double getArea();
}

class Circle implements Shape {
    // ...
}
```

Interfaces are useful for achieving abstraction, defining contracts, and enabling polymorphism. They provide a way to define behavior without specifying the implementation details, allowing for flexibility and code reuse. They are widely used in Java to define APIs and ensure a consistent interface across different classes.

4.9 PACKAGES

Packages are used to organize and group related classes, interfaces, and other resources. They provide a way to create a hierarchical structure and prevent naming conflicts between classes with the same name.

1.Package Declaration:

- A package is declared at the top of a Java source file using the package keyword followed by the package name.
- The package declaration must be the first non-comment line in the source file.
- By convention, package names are written in lowercase and use the reverse domain name convention, such as **com.example.myapp.**

package com.example.myapp;

2.Package Structure:

- Packages can be organized into a hierarchical structure, separated by periods (.).
- The package structure reflects the directory structure in which the corresponding Java files are stored.

- For example, the package **com.example.myapp** would typically correspond to the directory structure **com/example/myapp** on the file system.

3.Package Visibility:

- Classes and members within a package have package-level visibility by default.
- Package-level visibility means that they are accessible only within the same package.
- Classes and members marked as **public** can be accessed from other packages.

4.Importing Packages and Classes:

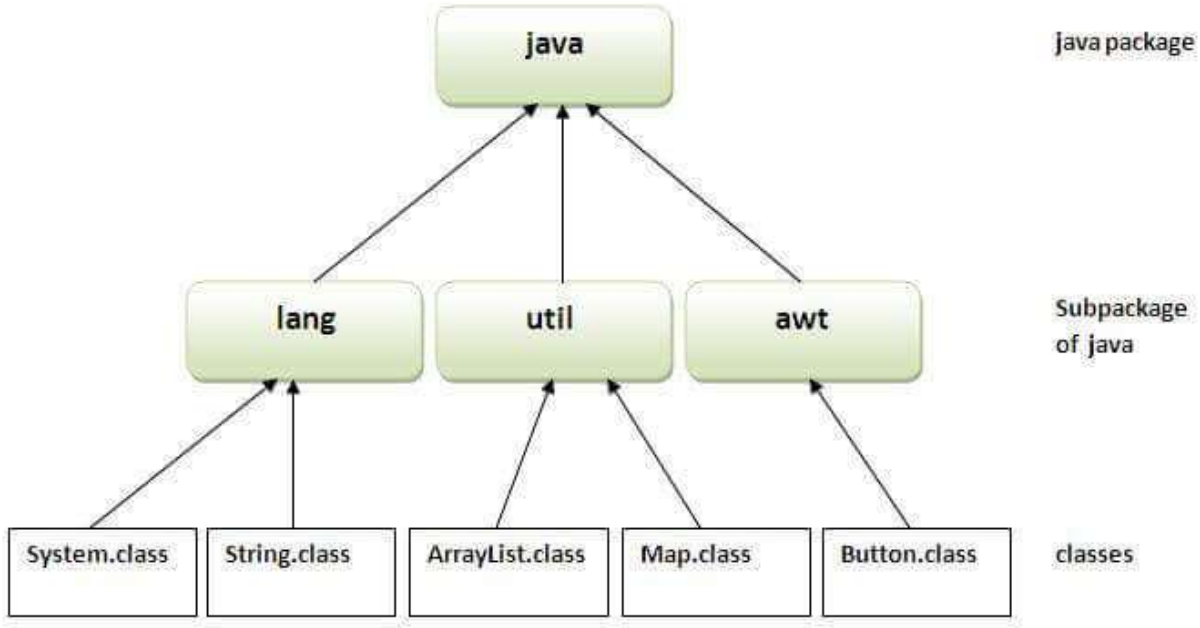
- To use a class from another package, it must be imported using the **import** statement.
- The **import** statement is placed at the top of the Java source file, below the package declaration and above the class declaration.
- Importing a package allows you to use any class within that package without specifying the package name each time.

import com.example.myapp.MyClass;

5.Standard Java Packages:

- Java provides a set of standard packages that contain classes and APIs for various functionalities, such as **java.util**, java.io, **java.lang**, etc.
- These packages are automatically available for use without requiring an import statement.

Packages play a crucial role in organizing and managing larger Java projects. They help avoid naming conflicts, provide encapsulation, and promote code modularity. By grouping related classes together, packages make it easier to understand and maintain code. Additionally, they facilitate code reuse by enabling the use of classes from other packages through import statements.



UNIT-5

EXCEPTION HANDLING AND MULTITHREADED PROGRAMMING

5.1TYPES OF ERROR

There Are Three Types Of Errors:

1.Compile-time Errors:

- Compile-time errors, also known as compilation errors or syntax errors, occur during the compilation phase of the program.
- These errors are caused by violations of the Java syntax rules or incorrect usage of language constructs.
- Examples of compile-time errors include missing semicolons, undefined variables, incompatible types, or using a reserved keyword as an identifier.
- When a compile-time error occurs, the compiler generates an error message, and the program cannot be compiled until the errors are fixed.

2.Runtime Errors:

- Runtime errors, also called exceptions, occur during the execution of the program.
- These errors are caused by exceptional conditions or unexpected situations that arise while the program is running.
- Common examples of runtime errors include division by zero (**ArithmeticException**), accessing an array element with an invalid index (**ArrayIndexOutOfBoundsException**), or attempting to open a non-existent file (**FileNotFoundException**).
- When a runtime error occurs, an exception object is thrown, which can be caught and handled using exception handling mechanisms like try-catch blocks. If unhandled, the program terminates abruptly.

3.Logical Errors:

- Logical errors, also known as semantic errors or bugs, occur when the program's logic or algorithm is incorrect.
- These errors do not cause the program to crash or generate error messages but result in undesired or incorrect behavior.
- Logical errors can be challenging to identify as they do not produce any compile-time or runtime errors. The program runs successfully, but the output or behavior is not as expected.
- Detecting and fixing logical errors typically involves careful code review, debugging, and testing**.**

It is important to note that compile-time errors are caught by the compiler during the compilation process, runtime errors occur during program execution and can be caught and handled using exception handling, while logical errors require careful analysis and debugging to identify and fix.

Understanding the different types of errors in Java is crucial for developing robust and error-free software. Proper testing, code review, and debugging techniques are essential to identify and resolve these errors effectively.

5.2 BASIC CONCEPTS OF EXCEPTION HANDLING

Exception handling in Java is a mechanism that allows you to handle and recover from exceptional situations or errors that occur during the execution of a program. It helps in making the program more robust by providing a structured approach to deal with unexpected or exceptional scenarios.

Exception

- An exception is an event that occurs during the execution of a program and disrupts the normal flow of the program.
- Exceptions represent various types of errors, such as runtime errors, input/output errors, arithmetic errors, and others.
- In Java, exceptions are represented by classes that are derived from the Throwable class or one of its subclasses.

5.3 TRY AND CATCH BLOCK

The try-catch block is used to handle exceptions and provide a mechanism to gracefully handle exceptional situations during the execution of a program. The try-catch block consists of a try block and one or more catch blocks.

```
try {
  // Code that may throw an exception
} catch (ExceptionType1 ex1) {
 // Exception handling code for ExceptionType1
} catch (ExceptionType2 ex2) {
// Exception handling code for ExceptionType2
} catch (Exception ex) {
  // Generic exception handling code for any other exception
}
```

The code that may throw an exception is placed within the try block.

If an exception occurs within the try block, the control immediately transfers to the corresponding catch block based on the type of exception.

Each catch block specifies the type of exception it can handle. If the caught exception matches the type specified in a catch block, that catch block is executed.

If no catch block is found that matches the type of the thrown exception, the exception propagates to the next level of the method call stack or terminates the program if it reaches the main method without being caught.

After executing the catch block, the program continues with the code that follows the try-catch block.

EXAMPLE:

```
try {
  int result = divide(10, 0); // This method may throw an ArithmeticException
```

```
    System.out.println("Result: " + result);
} catch (ArithmeticException ex) {
    System.out.println("Exception caught: " + ex.getMessage());
}

// ...

public int divide(int dividend, int divisor) {
    return dividend / divisor;
}
```

In this example, the divide method is called within the try block, and it may throw an ArithmeticException if the divisor is zero. The catch block following the try block catches the ArithmeticException and executes the exception handling code.

You can have multiple catch blocks to handle different types of exceptions. The catch blocks are evaluated sequentially, and the first catch block that matches the thrown exception type is executed.

It's important to handle exceptions appropriately in the catch block, which may include logging the error, displaying an error message to the user, or taking corrective actions.

By using try-catch blocks, you can handle exceptions and prevent them from causing the program to crash. It provides a mechanism for error handling, allowing your program to gracefully recover from exceptional situations and continue its execution.

5.4 THROW AND THROWS

throw and **throws** are two keywords used in exception handling to deal with exceptions and specify exception propagation.

1.throw Keyword:

- The throw keyword is used to explicitly throw an exception from within a method or a block of code.
- It is followed by an instance of an exception class or a subclass of Throwable.
- When a throw statement is encountered, the normal flow of the program is disrupted, and the specified exception is thrown.
- It allows you to generate and throw exceptions explicitly to handle exceptional scenarios.

```
public void validateAge(int age) {
    if (age < 0) {
    throw new IllegalArgumentException("Age cannot be negative.");
```

```
    }
}
```

In this example, the **validateAge** method throws an **IllegalArgumentException** if the given age is negative. The throw keyword is used to throw the exception, and a custom error message is provided.

2.throws Keyword:

- The throws keyword is used in a method declaration to indicate that the method may throw one or more types of exceptions.
- It specifies that the method is not handling the exception itself but is instead passing the responsibility to the calling method or the JVM.
- Multiple exceptions can be declared using a comma-separated list.

```
public void readFile() throws IOException, FileNotFoundException {
    // Code that may throw IOException or FileNotFoundException
}
```

In this example, the **readFile** method declares that it may throw **IOException** or **FileNotFoundException**. The responsibility of handling these exceptions is passed to the caller of the **readFile** method.

When a method throws an exception using the throws keyword, it must be either caught and handled using a try-catch block by the calling method, or the calling method should also declare the exception using the throws keyword in its own declaration.

It's important to note that throw is used to raise an exception explicitly within a method, whereas **throws** is used to indicate that a method may potentially **throw** an exception, allowing the caller to handle it or propagate it further.

Both **throw** and throws keywords play a crucial role in handling and propagating exceptions in Java programs, ensuring that exceptional scenarios are appropriately handled and managed.

5.5 USER DEFINE EXCEPTION

In Java, you can define your own exceptions by creating a subclass of the **Exception** class or one of its existing subclasses. This allows you to create custom exceptions that represent specific exceptional conditions in your application.

1.Create A Subclass Of The Exception Class Or One Of Its Existing Subclasses:

You can extend the Exception class directly or choose a more specific subclass such as **RuntimeException** or **IOException**, depending on the nature of the exception you want to create.

To create a custom checked exception, extend the Exception class.

To create a custom unchecked exception, extend the **RuntimeException** class.

Example of a custom exception by extending Exception class:

```
public class MyException extends Exception {
```

```
 // Constructor(s) and additional methods can be defined
}
```

Example of a custom exception by extending Exception class subclass:

```
public class MyException extends RuntimeException {
 // Constructor(s) and additional methods can be defined
}
```

2.Customize the exception class:

You can provide constructors to initialize the exception object with a specific error message or any additional information.

You can add additional methods or properties to the exception class to provide more functionality or context.

Example:

```
public class InvalidInputException extends RuntimeException {
private String input;


public InvalidInputException(String message, String input) {
  super(message);
this.input = input;
  }


   public String getInput() {
   return input;
  }
}
```

In this example, a custom unchecked exception called **InvalidInputException** is defined. It extends the **RuntimeException** class and includes an additional property input. It also provides a constructor to initialize the exception object with an error message and the invalid input value.

Once you have defined a custom exception, you can use it in your code by throwing instances of your exception class when necessary. For example:

```
public void processInput(String input) {
   if (input == null) {
   throw new InvalidInputException("Input cannot be null", input);
}
// Process the input
}
```

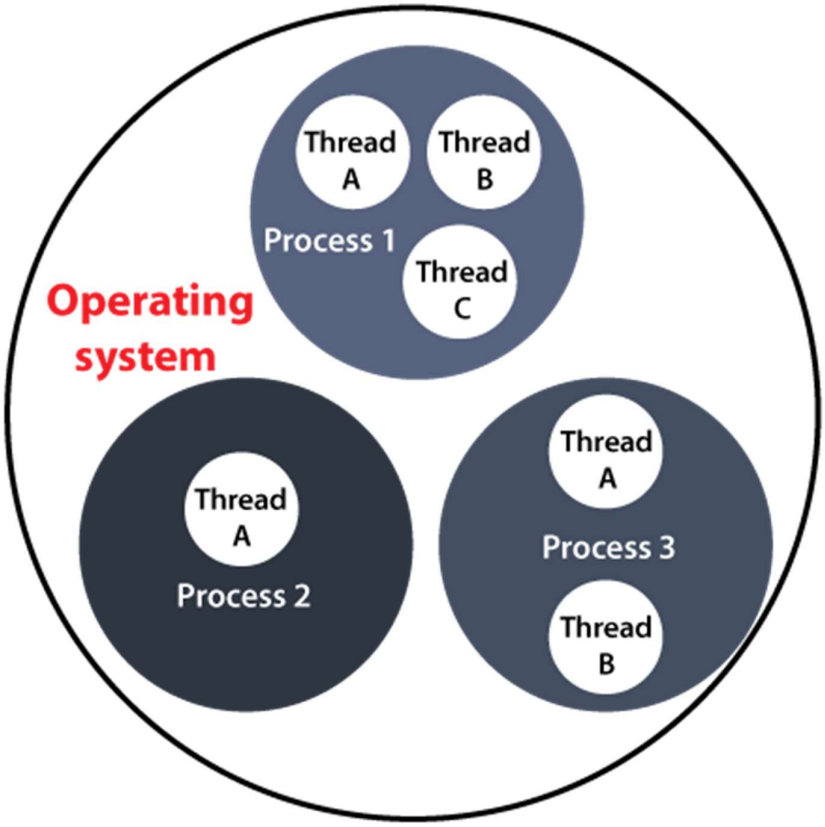
In this example, the **processInput** method throws an instance of the **InvalidInputException** when the input is **null**.

By creating user-defined exceptions, you can represent specific exceptional conditions in your application and provide meaningful error messages or additional information to aid in debugging and error handling. It allows you to design a more robust and expressive exception hierarchy tailored to your application's needs.

5.6 INTRODUCTION OF THREAD

A Thread is a very light-weighted process, or we can say the smallest part of the process that allows a program to operate more efficiently by running multiple tasks simultaneously.

In order to perform complicated tasks in the background, we used the Thread concept in Java. All the tasks are executed without affecting the main program. In a program or process, all the threads have their own separate path for execution, so each thread of a process is independent.



5.7 IMPLEMENTATION OF THREAD

In Java, there are two primary ways to implement threads: by extending the **Thread** class or by implementing the **Runnable** interface. Let's look at both approaches:

1.Extending the Thread class:

In this approach, you create a new class that extends the **Thread** class and override its **run**() method, which represents the code that will be executed in the new thread.

You can then create an instance of your custom thread class and start the thread by calling the **start**() method.

EXAMPLE:

```
class MyThread extends Thread {
    @Override
 public void run() {
 // Code to be executed in the new thread
  System.out.println("Thread is running");
 }
}

public class Main {
  public static void main(String[] args) {
    MyThread thread = new MyThread();
thread.start(); // Starts the execution of the new thread
   }
}
```

2.Implementing the Runnable interface:

In this approach, you create a class that implements the **Runnable** interface and implement the **run**() method defined by the interface.

You then create an instance of your custom class and pass it to a **Thread** object's constructor.

Finally, you call the **start**() method on the **Thread** object to start the new thread.

EXAMPLE:

```
class MyRunnable implements Runnable {
   @Override
   public void run() {
    // Code to be executed in the new thread
  System.out.println("Thread is running");
 }
}

public class Main {
  public static void main(String[] args) {
   MyRunnable runnable = new MyRunnable();
   Thread thread = new Thread(runnable);
```

```
thread.start(); // Starts the execution of the new thread
} }
```

Both approaches create a new thread of execution, and the run() method is called when the thread starts. You can perform any desired operations within the run() method. The start() method initiates the execution of the thread, which will run concurrently with the main thread.

It's important to note that creating and managing threads requires careful consideration of thread synchronization and coordination to avoid potential issues such as race conditions or data inconsistencies. You should use synchronization mechanisms like synchronized blocks or locks when accessing shared resources to ensure thread safety.

Additionally, Java provides other features and utilities for thread management, such as thread synchronization, interruption, thread pools, and concurrent data structures, which can be explored based on specific requirements.

Remember to handle any exceptions thrown within the **run**() method appropriately, as uncaught exceptions in a thread can cause the program to terminate unexpectedly.

5.8 THREAD LIFE CYCLE AND METHOD

The life cycle of a thread in Java refers to its various states and transitions that occur during its execution. A thread goes through several stages from its creation to termination.

The Java thread life cycle consists of the following states:

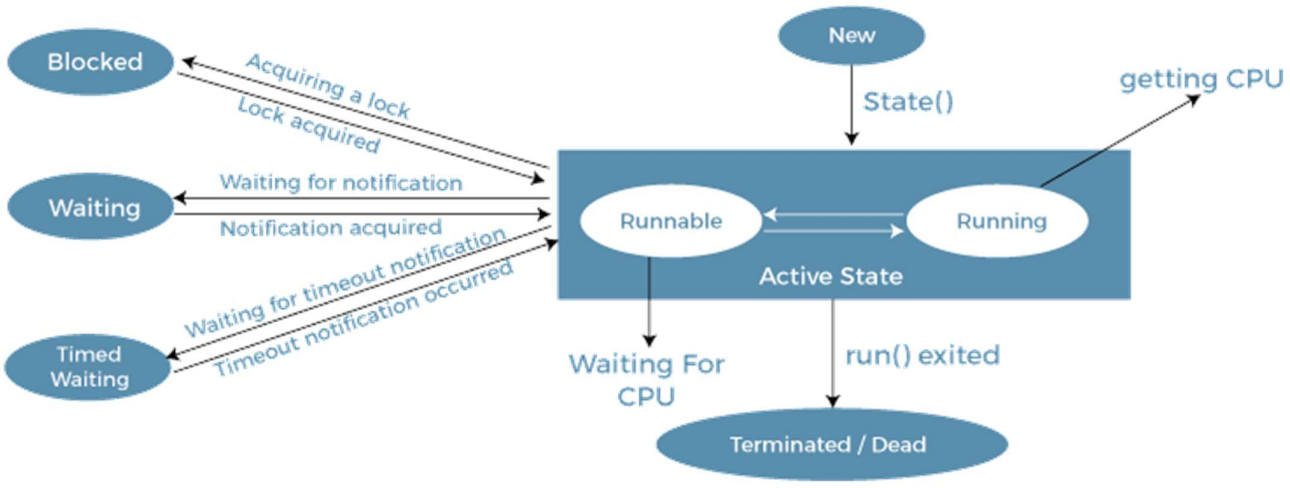
New

Active

Blocked / Waiting

Timed Waiting

Terminated



Life Cycle of a Thread

1.New:

The thread is in the new state when it has been created but has not yet started.

In this state, the thread has been instantiated but has not yet invoked the **start**() method.

2.Runnable:

When the **start**() method is called, the thread enters the runnable state.

In this state, the thread is eligible for execution, but it may or may not be currently executing.

The thread scheduler selects a thread from the runnable pool and executes it.

3.Running:

A thread enters the running state when it starts executing its **run**() method.

In this state, the actual execution of the thread's code is taking place.

The thread remains in this state until it is paused, interrupted, or its execution completes.

4.Blocked/Waiting:

A thread can transition to the blocked or waiting state for various reasons:

Blocked: If the thread is waiting for a monitor lock to enter a synchronized block or method and another thread holds the lock.

Waiting: If the thread is waiting indefinitely for another thread to perform a particular action, such as waiting for a condition to be satisfied or waiting for a **notify**/**notifyAll** signal.

5.Timed Waiting:

Similar to the blocked/waiting state, a thread can also enter a timed waiting state, where it waits for a specific period of time.

This state is reached when the thread calls methods such as **Thread.sleep(), Object.wait()**, or **Thread.join()** with a specified timeout.

6.Terminated:

A thread enters the terminated state when its **run**() method completes execution or when it is terminated prematurely.

Once in the terminated state, the thread cannot be resumed or restarted.

It's important to note that the transitions between these states are managed by the Java Virtual Machine (JVM) and the thread scheduler. The scheduler determines which thread to execute based on factors like thread priorities, fairness, and the scheduling algorithm used by the JVM.

Understanding the life cycle of a thread is essential for proper thread management and synchronization. It helps in designing concurrent applications and handling synchronization issues effectively.

Additionally, Java provides methods and utilities to manage and control threads throughout their life cycle, such as **Thread.sleep()**, **Thread.yield(),Thread.interrupt(),** and synchronization mechanisms like locks, conditions, and barriers.

5.9 MULTITHREADING

Multithreading in Java refers to the concurrent execution of multiple threads within a single program. It allows different threads to run independently, performing tasks simultaneously and improving overall application performance and responsiveness. Java provides built-in features and APIs to support multithreading, making it easier to develop concurrent applications.

1.Thread Creation:

Java provides two primary ways to create threads: by extending the Thread class or by implementing the Runnable interface (as discussed earlier).

The Thread class and Runnable interface provide methods and hooks for thread initialization, execution, and termination.

2.Thread Synchronization:

When multiple threads access shared resources simultaneously, synchronization is required to avoid data inconsistencies and race conditions.

Java provides synchronization mechanisms like synchronized blocks, synchronized methods, and explicit locks (e.g., Lock interface and **ReentrantLock** class) to ensure thread safety and proper synchronization between threads.

3.Thread Scheduling and Priorities:

Java's thread scheduler determines the order and timing of thread execution.

Thread priorities (ranging from 1 to 10) can be assigned to threads to influence their relative execution order.

However, thread priorities do not guarantee precise ordering, and the actual behavior may vary across JVM implementations and platforms.

4.Thread Coordination:

Threads can be coordinated to work together or communicate using synchronization mechanisms like **wait**(), **notify**(), and **notifyAll**().

These methods are used to pause, resume, or wake up threads based on specific conditions, allowing threads to communicate and synchronize their actions.

5.Thread Safety:

Thread safety ensures that shared resources are accessed in a safe and consistent manner by multiple threads.

Proper synchronization, using synchronization blocks or locks, is essential to achieve thread safety and prevent race conditions and data corruption.

6.Executors and Thread Pools:

The **Executor** framework in Java provides higher-level abstractions for managing and controlling threads.

Thread pools can be created using the **ExecutorService** interface, allowing for efficient management and reuse of threads.

7.Concurrent Data Structures:

Java provides concurrent data structures, such as **ConcurrentHashMap** and **ConcurrentLinkedQueue**, which are designed for safe access and manipulation by multiple threads concurrently.

These data structures handle synchronization internally, making them suitable for multithreaded environments.

Multithreading is particularly useful in scenarios where tasks can be divided into smaller units of work that can run independently or when tasks involve blocking operations like I/O or network requests. By utilizing multiple threads, you can maximize resource utilization and enhance the responsiveness and performance of your Java applications.

However, it's crucial to handle synchronization and coordination between threads carefully to avoid synchronization issues, deadlocks, and other concurrency-related problems. Proper design, synchronization mechanisms, and thread safety practices are necessary to ensure the correct and efficient execution of multithreaded programs.