VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

(An Autonomous Institution)

(Approved by AICTE, New Delhi & Affiliated to JNTUH, Hyderabad)

B.Tech(CSE) II Year / II Semester (R22)

Lecture Notes

Name of the Faculty	**KISHORE K**
Department	**CSE(Data Science)**
Year & Semester	**B.Tech-II & II Sem**
Subject Name	**OBJECT ORIENTED PROGRAMMING THROUGH JAVA**

DEPARTMENT OF CSE (Data Science)

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

(Approved by AICTE, New Delhi & Affiliated to JNTUH, Hyderabad)

An Autonomous Institution

AZIZ NAGAR, C B POST, HYDERABAD-500075

2024-2025

Java Basics PART-1

History of Java, Java buzzwords, data types, variables, scope and life time of variables, arrays, operators, expressions, control statements, type conversion and casting, simple java program

History of Java Programming Language

Java is an **Object-Oriented Programming Language** was originally developed by **Sun Microsystems** later acquired by **Oracle Corporation** which was initiated by **James Gosling, Mike Sheridan and Patrick Naughton** in June **1991**.

Sun Microsystems released the first public implementation as **Java 1.0** in **1996**. It promised **Write Once, Run Anywhere (WORA)** functionality, providing no-cost run-times on popular platforms.

The team initiated this project to develop a language for digital devices such as set-top boxes, television, etc. Originally C++ was considered to be used in the project but the idea was rejected for several reasons (For instance C++ required more memory). Gosling endeavoured to alter and expand C++ . James Gosling and his team called their project "**Greentalk**" and its file extension was **.gt** and later became to known as "**OAK**".

Why "Oak"?

The name **Oak** was used by **Gosling** after an **oak tree** that remained outside his office. Also, Oak is an image of solidarity and picked as a national tree of numerous nations like the U.S.A., France, Germany, Romania, etc. But they had to later rename it as "**JAVA**" as it was already a trademark by **Oak Technologies.**

The name Java originates from a sort of **espresso bean**. The team came up with this name while having a coffee near their office. Java was created on the principles like **Robust, Portable, Platform Independent, High Performance, Multithread, etc.** and it is widely used in **internet programming, mobile devices, games, e-business solutions, etc.**

Java Versions History

Over the period of nearly 30 years, Java has seen many minor and major versions.

Currently, more than 3 billion devices run on features built in Java.

Following is a brief explanation of versions of java till date.

Sr.No.	Version	Date	Description
1.	JDK Beta	1995	Initial Draft version
2.	JDK 1.0	23 Jan 1996	A stable variant JDK 1.0.2 was termed as JDK 1
3.	JDK 1.1	19 Feb 1997	Major features like **JavaBeans, RMI, JDBC, inner classes** were added in this release.
4.	JDK 1.2	8 Dec 1998	**Swing, JIT Compiler, Java Modules, Collections** were introduced to JAVA and this release was a great success.
5.	JDK 1.3	8 May 2000	HotSpot **JVM**, JNDI, JPDA, JavaSound and support for Synthetic proxy classes were added.
6.	JDK 1.4	6 Feb 2002	Image I/O API to create/read JPEG/PNG image were added. Integrated **XML parser** and XSLT processor (JAXP) and Preferences API were other important updates.
7.	JDK 1.5 or J2SE 5	30 Sep 2004	Various new features were added to the language like foreach, var-args, generics etc.
8.	JAVA SE 6	11 Dec 2006	Notation was dropped to SE and upgrades done to JAXB 2.0, JSR 269 support and JDBC 4.0 support added.
9.	JAVA SE 7	7 Jul 2011	Support for dynamic languages added to JVM. Another enhancements included

			string in switch case, compressed 64 bit pointers etc.
10.	JAVA SE 8	18 Mar 2014	Support for functional programming added. **Lambda expressions, streams, default methods, new date-time** APIs introduced.
11.	JAVA SE 9	21 Sep 2017	**Module system** introduced which can be applied to JVM platform.
12.	JAVA SE 10	20 Mar 2018	Unicode language-tag extensions added. Root certificates, threadlocal handshakes, support for heap allocation on alternate memory devices etc were introduced.
13.	JAVA SE 11	5 Sep 2018	Dynamic class-file constants, Epsilon a no-op garbage collector, local-variable support in lambda parameters, Low-overhead heap profiling support added.
14.	JAVA SE 12	19 Mar 2019	Experimental Garbage Collector, JVM Constants API added.
15.	JAVA SE 13	17 Sep 2019	Feature added - Text Blocks (Multiline strings),Enhanced Thread-local handshakes.
16.	JAVA SE 14	17 Mar 2020	Feature added - Records, a new class type for modelling, **Pattern Matching for instanceof**, Intuitive **NullPointerException handling**.
17.	JAVA SE 15	15 Sep 2020	Feature added - **Sealed Classes**, **Hidden Classes**, Foreign Function and Memory API (Incubator).
18.	JAVA SE 16	16 Mar 2021	Feature added as preview - Records, Pattern Matching for switch, Unix Domain Socket Channel (Incubator) etc.

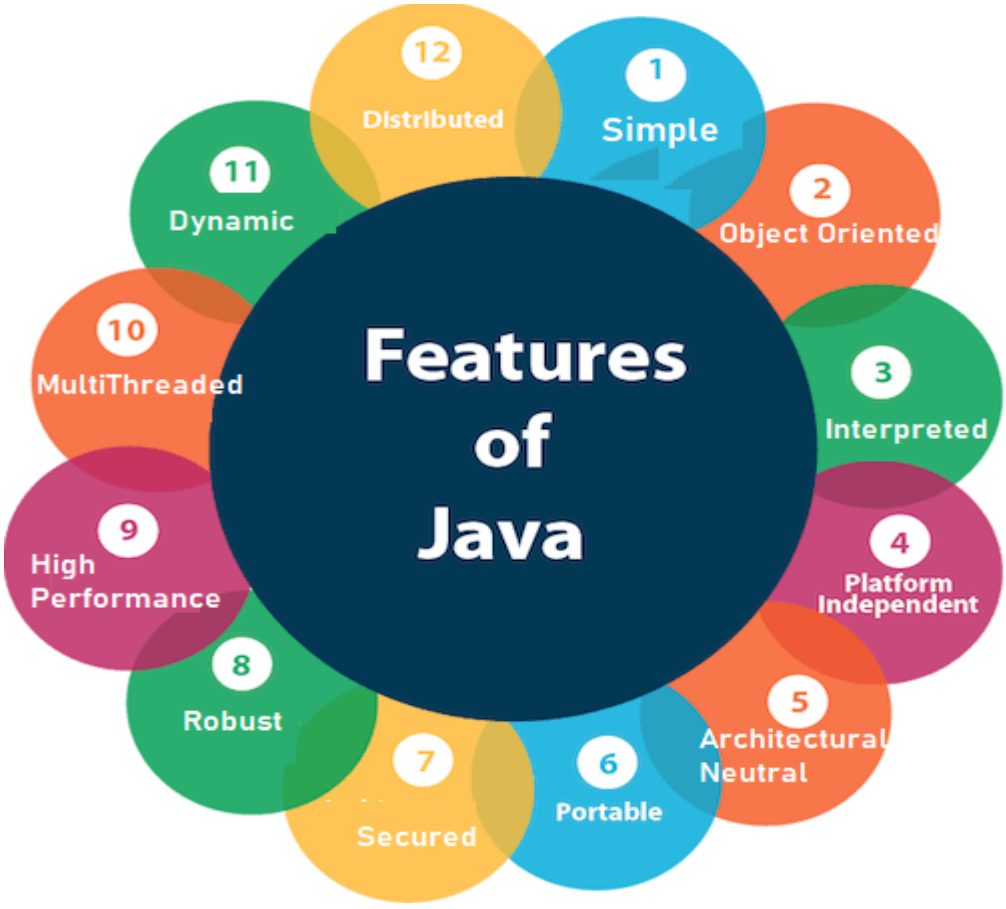
19.	JAVA SE 17	14 Sep 2021	Feature added as finalized - Sealed Classes, Pattern Matching for instanceof, Strong encapsulation of JDK internals by default. New macOS rendering pipeline etc.
20.	JAVA SE 18	22 Mar 2022	Feature added - UTF-8 by Default, Code Snippets in Java API Documentation, Vector API (Third incubator), Foreign Function, Memory API (Second Incubator) etc.
21.	JAVA SE 19	20 Sep 2022	Feature added - Record pattern, Vector API (Fourth incubator), Structured Concurrency (Incubator) etc.
22.	JAVA SE 20	21 Mar 2023	Feature added - Scoped Values (Incubator), Record Patterns (Second Preview), Pattern Matching for switch (Fourth Preview),Foreign Function & Memory API (Second Preview) etc.
23.	JAVA SE 21	19 Sep 2023	Feature added - String Templates (Preview), Sequenced Collections, Generational ZGC, Record Patterns, Pattern Matching for switch etc.
24.	Java SE 22	19 Mar 2024	Feature added - Region Pinning for G1 garbage collector, foreign functions and memory APIs , multi-file source code programs support, string templates, vector apis (seventh incubator), unnamed variables, patterns, stream gatherers (first preview) etc.
25.	Java SE 23	17 Sep 2024	Feature added - Primitive types in patterns, class file APIs, vector APIs (Eighth incubator), ZDC, generation mode by default etc.

Java Platform Editions

1. Java Platform, Standard Edition **(Java SE)** –A platform for developing and deploying portable code for desktop and server environments.
2. Java Platform, Enterprise Edition **(Java EE)** –A platform for developing and running web and enterprise applications.
3. Java Platform, Micro Edition **(Java ME)** –A flexible environment for applications that run on mobile and embedded devices.
4. **Java Card** - It is the tiniest of Java platforms targeted for embedded devices like SIM cards, Banking Cards, Identity Cards, Healthcare Cards, Passports and Several IoT Products.

Java Buzzwords

The primary objective of **Java programming** language creation was to make it simple, portable and secure programming language. Apart from this, there are some excellent features which play an important role in the popularity of this language. The features of Java are also known as Java Buzzwords.



A list of the most important features of the Java language is given below.

1. Simple
2. Object-Oriented
3. Interpreted
4. Platform Independent
5. Architectural Neutral
6. Portable
7. Secured
8. Robust
9. High Performance
10. Multithreaded
11. Dynamic
12. Distributed

1. Simple

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun Microsystem, Java language is a simple programming language because:

- Java syntax is based on C++ (so easier for programmers to learn it after C++).
- Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
- There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

2. Object-Oriented

Java is an object-oriented programming language. Everything in Java is an object.

Object-Oriented Programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

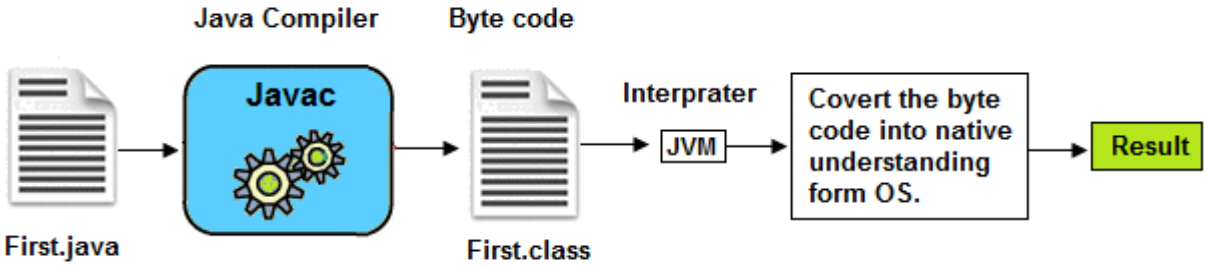
Basic concepts of OOPs are:

1. Object
2. Class
3. Encapsulation
4. Abstraction
5. Inheritance
6. Polymorphism

3. Interpreted

Java is a Compile and Interpreted Language. Compiler converts (translates) source code (.java file) into bytecode (.class file).

A bytecode is a binary code that is understood and interpreted by Java Virtual Machine (JVM) on the underlying operating system.



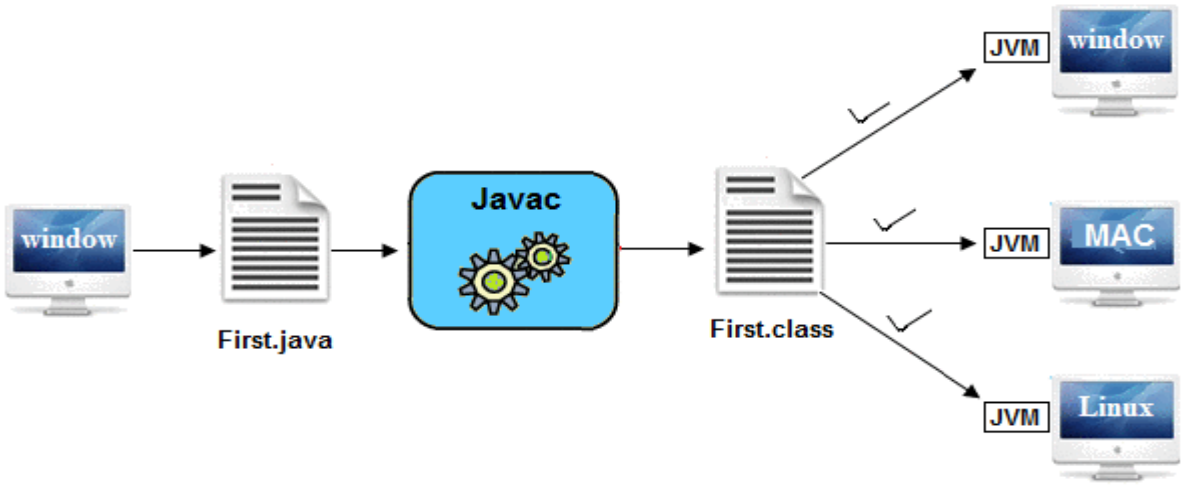
4. Platform Independent

Java is platform independent because it is different from other languages like **C, C++,** etc. which are compiled into platform specific machines.

The Java platform differs from most other platforms. It has two components:

1. Runtime Environment
2. API(Application Programming Interface)

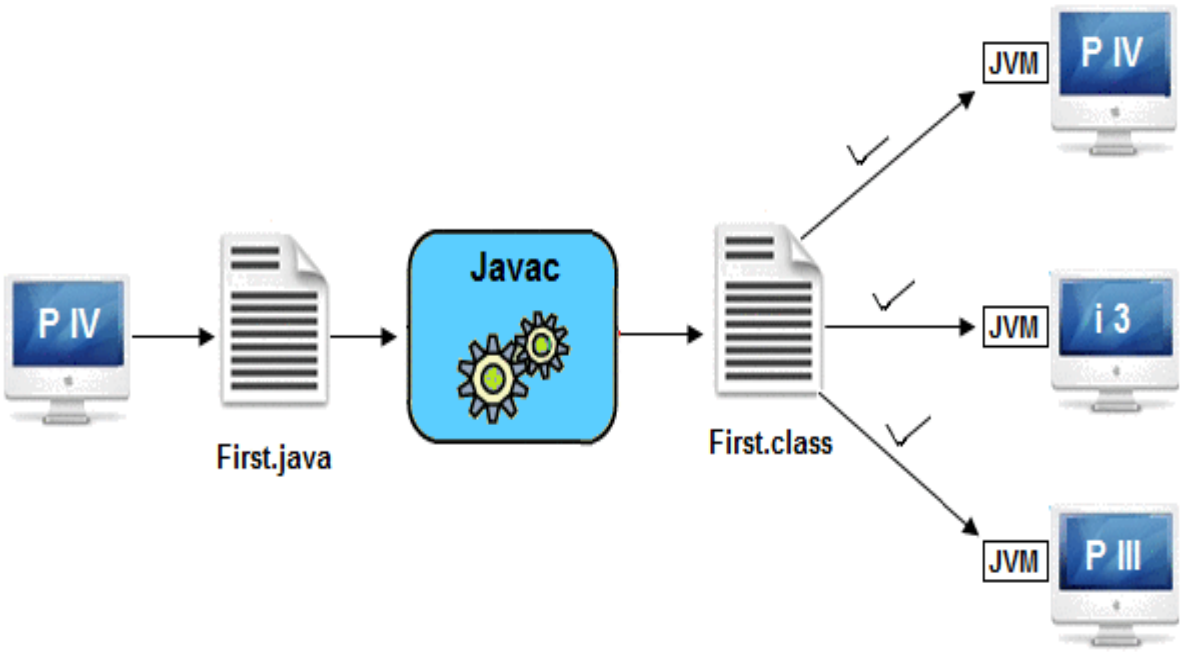
Java code can be executed on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., **Write Once and Run Anywhere (WORA).**



5. Architectural 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.



6. Portable

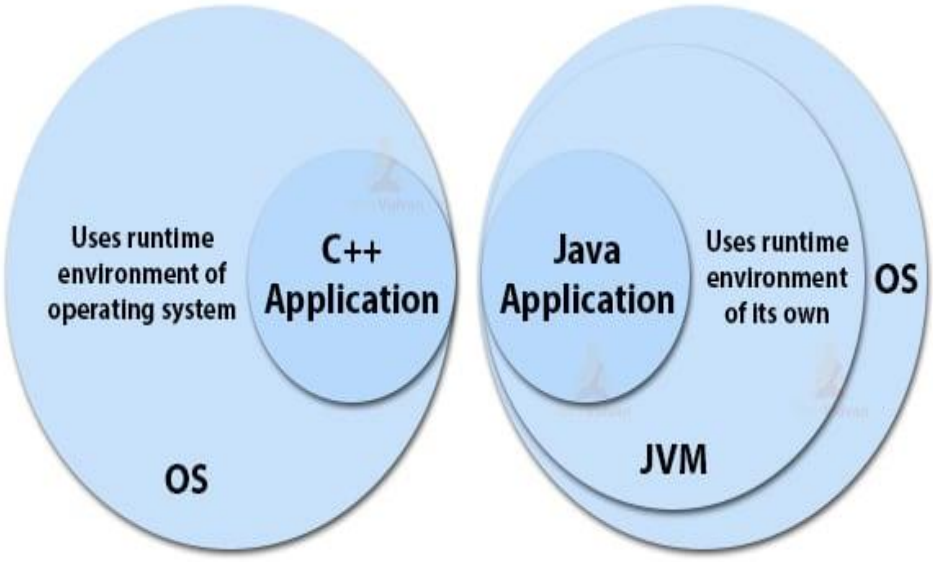
Java is portable because it facilitates you to carry the Java bytecode to any platform ie different operating systems and architectures. It doesn't require any implementation.

- Java code is compiled into bytecode that can run on any device with a Java Virtual Machine (JVM).
- The JVM is platform-dependent, meaning that a different JVM is designed for each operating system.
- The bytecode is platform-independent, meaning that it can run on different operating systems.
- This allows developers to write code once and run it on many different devices.

7. Secured

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- No Explicit Pointer
- Java Programs run inside a Virtual Machine Sandbox



- **Classloader:** Classloader in Java is a part of the Java Runtime Environment (JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access rights to objects.
- **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

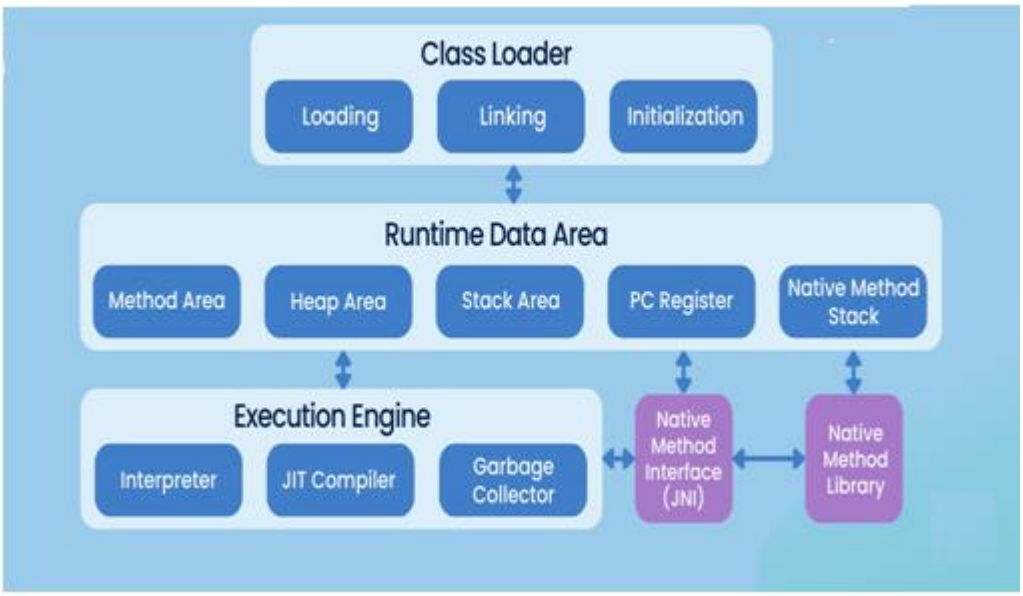
8. Robust

The Java is robust language ie strong because:

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

9. High Performance

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C, C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.



10.Multithreaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

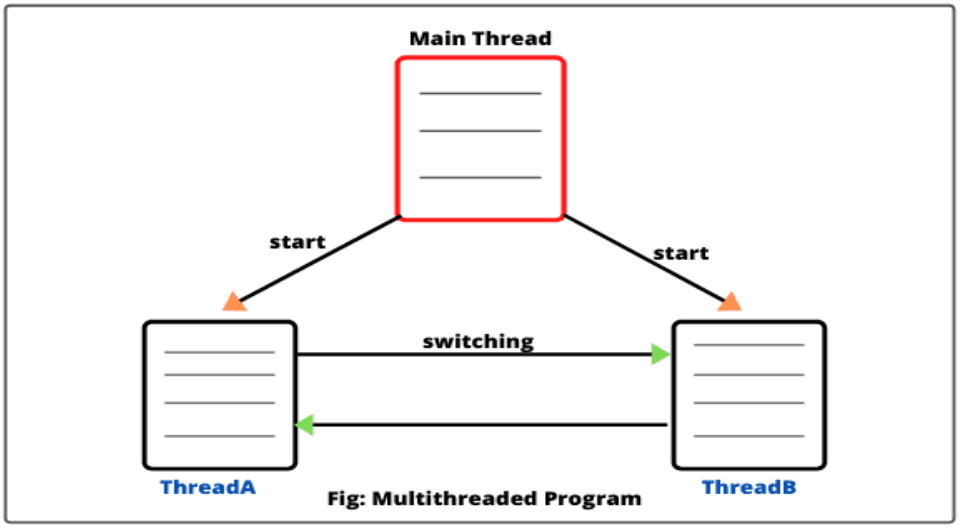


Fig: Multithreaded Program

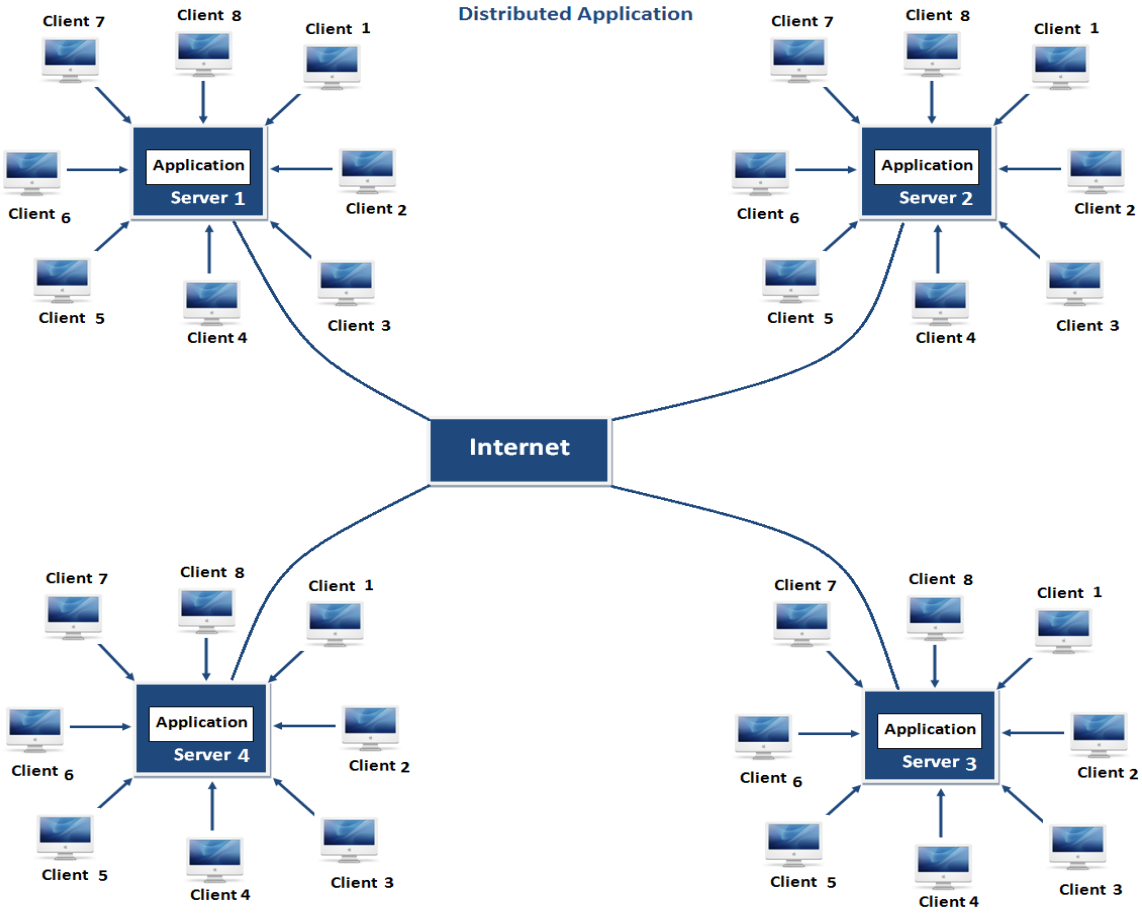
11.Dynamic

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).

12.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.



Basic Requirement to Run Java Program

For executing any Java program, the following software or application must be properly installed.

- Install the JDK if we do not have installed it, download the JDK and install it.
  https://www.oracle.com/in/java/technologies/downloads/
- Set path of the jdk/bin directory
- Create the Java program.
- Compile and run the Java program.

Creating Hello World Example

Let's create the hello world Java program:

```
class Simple
{
   public static void main(String args[])
   {
    System.out.println("Hello Java");
   }
}
```

Test it Now

Save the above file as **Simple.java**.

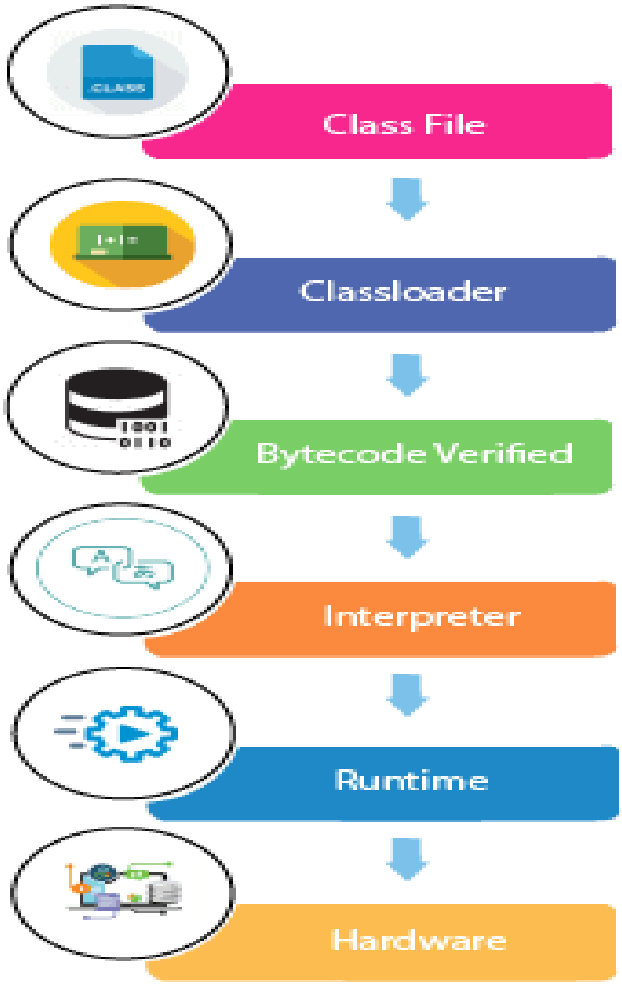
To compile:	javac Simple.java
To execute:	java Simple

Output:

```
Hello Java
```

What happens at runtime?

At runtime, the following steps are performed:



Discussion about JDK, JRE, and JVM

1. Java Development Kit (JDK)
2. Java Runtime Environment (JRE)
3. Java Virtual Machine (JVM)

We must understand the differences between JDK, JRE, and JVM before proceeding further to Java.

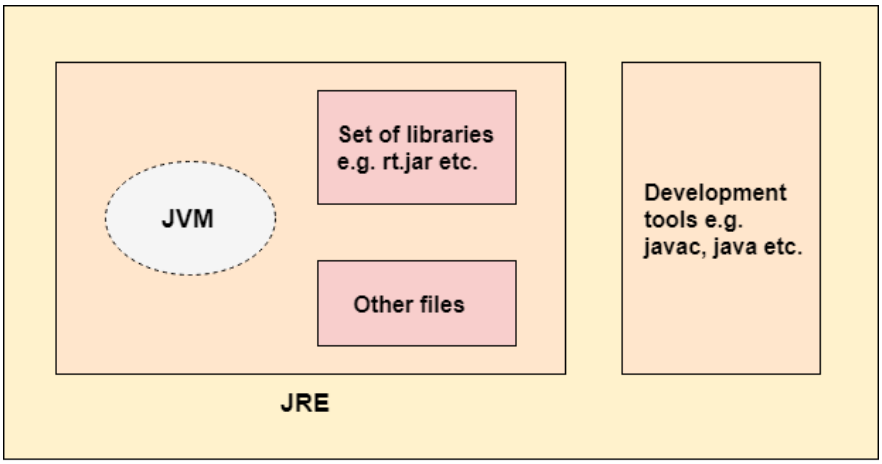
JDK

JDK is an acronym for **Java Development Kit**. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains **JRE + development tools**.

JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

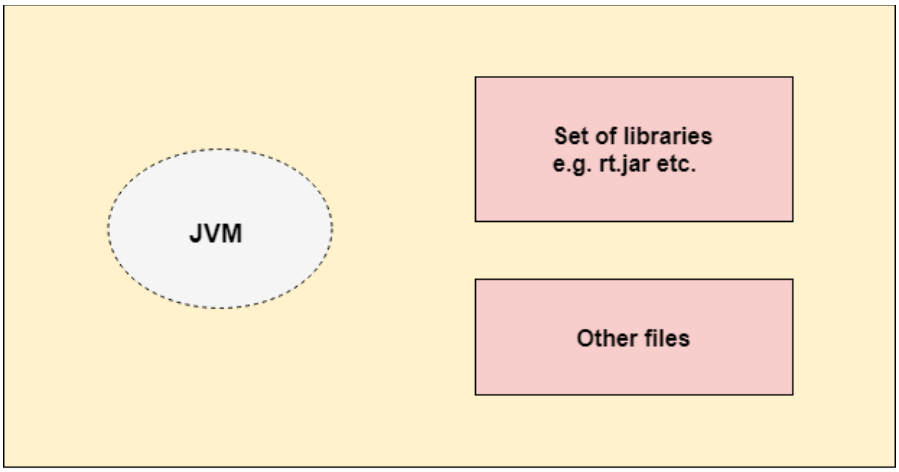
- **Standard Edition Java Platform**
- **Enterprise Edition Java Platform**
- **Micro Edition Java Platform**

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.



JRE

JRE is an acronym for **Java Runtime Environment**. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.



JRE

JVM

JVM (**Java Virtual Machine**) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

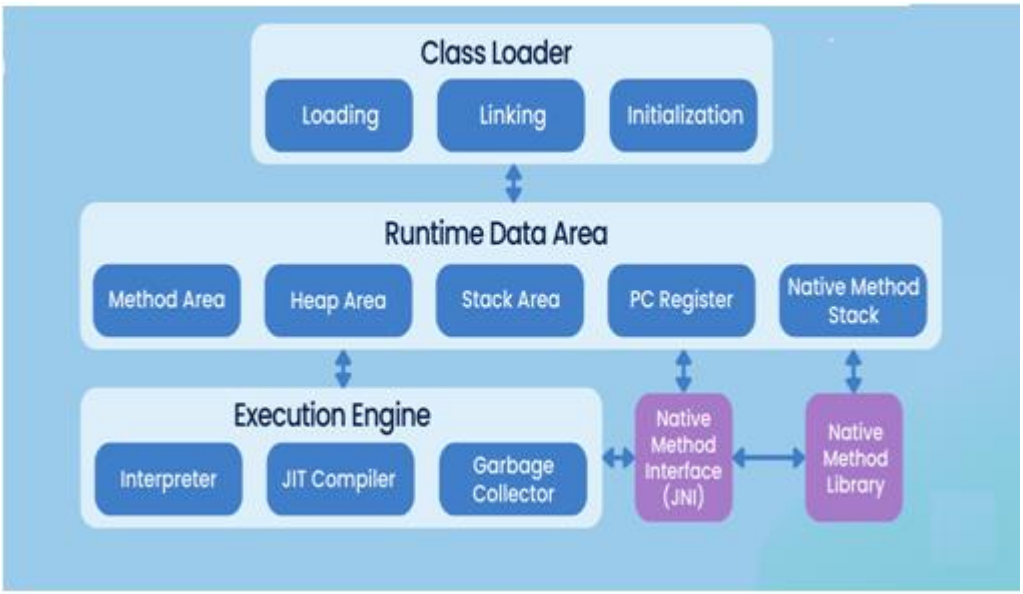
JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each **OS** is different from each other. However, Java is platform independent. There are three notions of the JVM: *specification*, *implementation*, and *instance*.

The JVM performs the following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM Architecture

Let's understand the internal architecture of JVM. It contains classloader, memory area, execution engine etc.



How to set path in Java

The path is required to be set for using tools such as javac, java, etc.

If you are saving the Java source file inside the JDK/bin directory, the path is not required to be set because all the tools will be available in the current directory.

However, if you have your Java file outside the JDK/bin folder, it is necessary to set the path of JDK.

There are two ways to set the path in Java:

1. Temporary
2. Permanent

1) How to set the Temporary Path of JDK in Windows

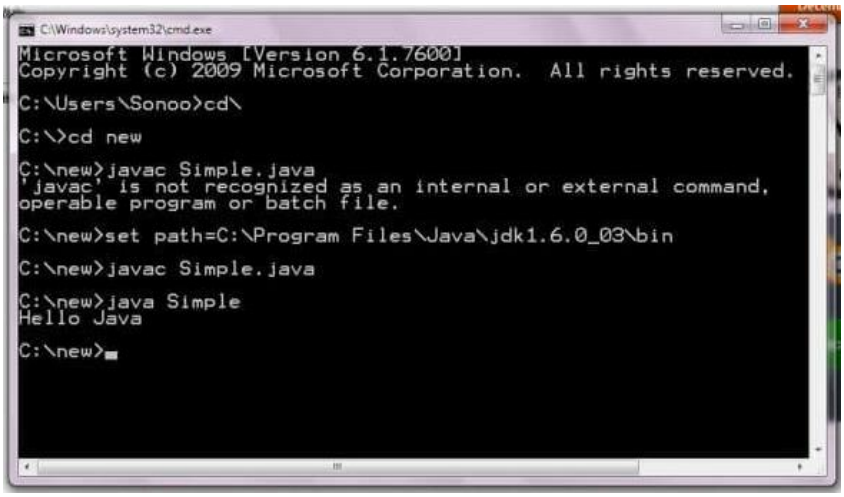
To set the temporary path of JDK, you need to follow the following steps:

- Open the command prompt
- Copy the path of the JDK/bin directory
- Write in command prompt: set path=copied_path

For Example:

set path=C:\Program Files\Java\jdk1.6.0_23\bin

Let's see it in the figure given below:



2) How to set Permanent Path of JDK in Windows

For setting the permanent path of JDK, you need to follow these steps:

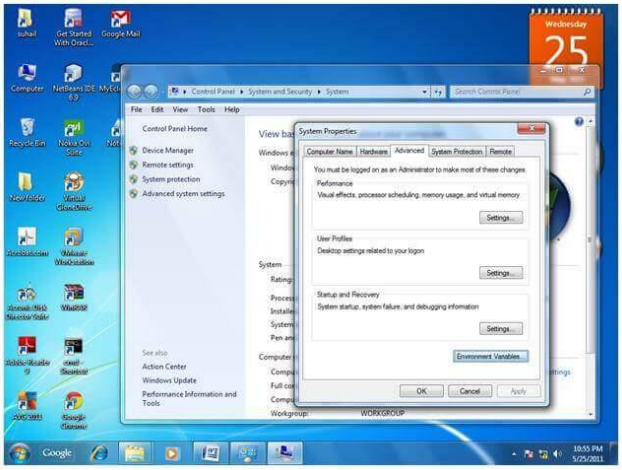
- Go to MyComputer properties -> advanced tab -> environment variables -> new tab of user variable -> write path in variable name -> write path of bin folder in variable value -> ok -> ok -> ok

For Example:

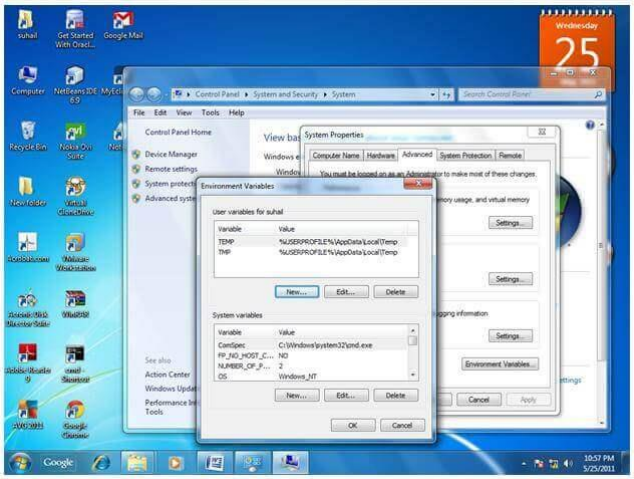
1) Go to MyComputer properties


2) Click on the advanced tab

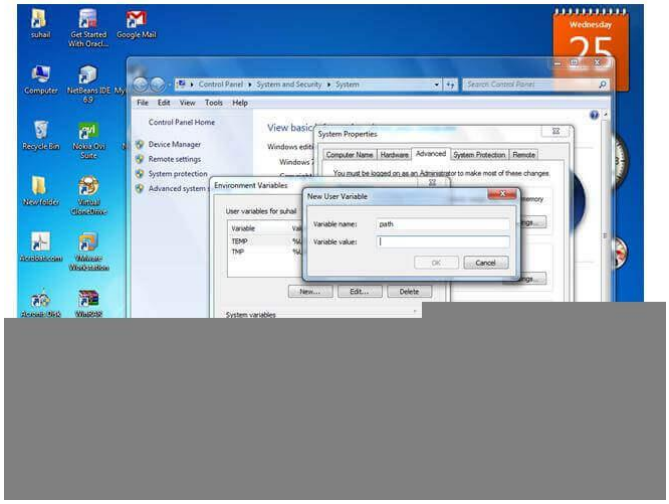
3) Click on environment variables



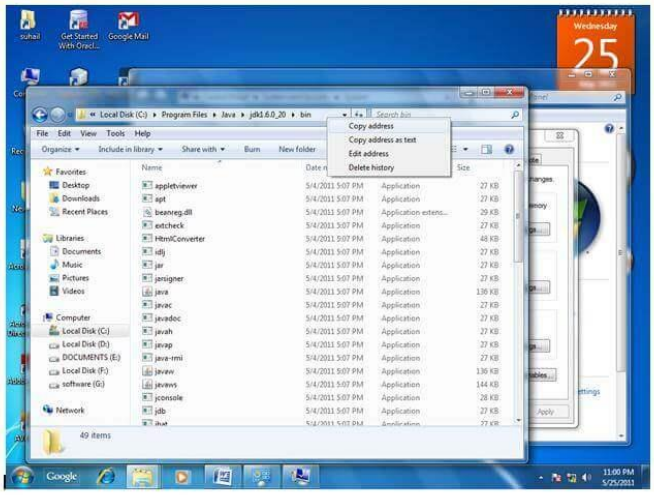
4) Click on the new tab of user variables



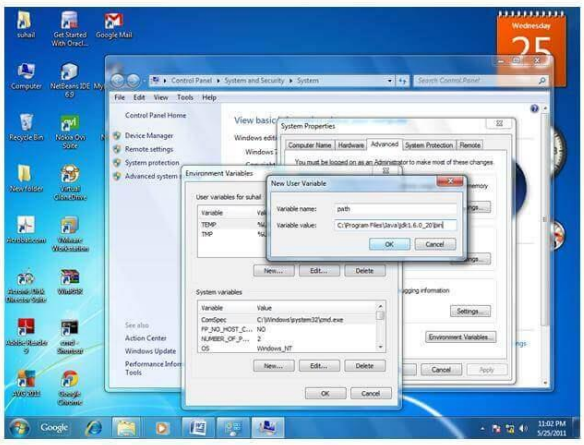
5) Write the path in the variable name



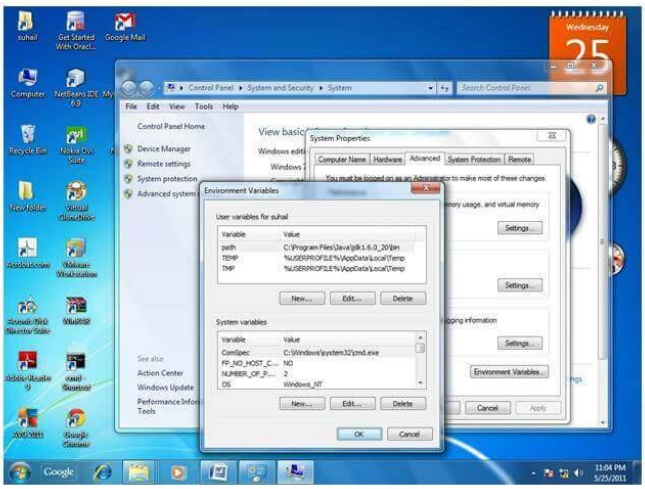
6) Copy the path of bin folder



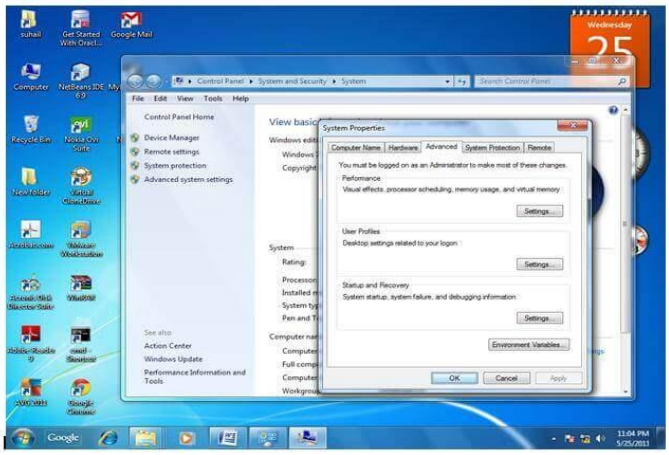
7) Paste path of bin folder in the variable value



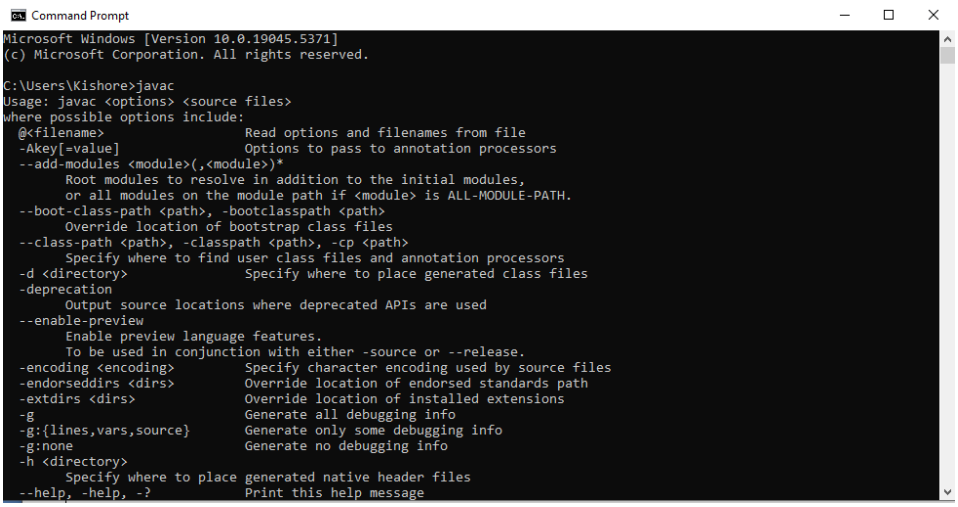
8) Click on ok button



9) Click on ok button



Now your permanent path is set. You can now execute any program of java from any drive.



Java Keywords

Java keywords are reserved words with predefined meanings in the Java programming language. They are part of the syntax and cannot be used as identifiers, such as variable names, class names, or method names.

List of Java Keywords

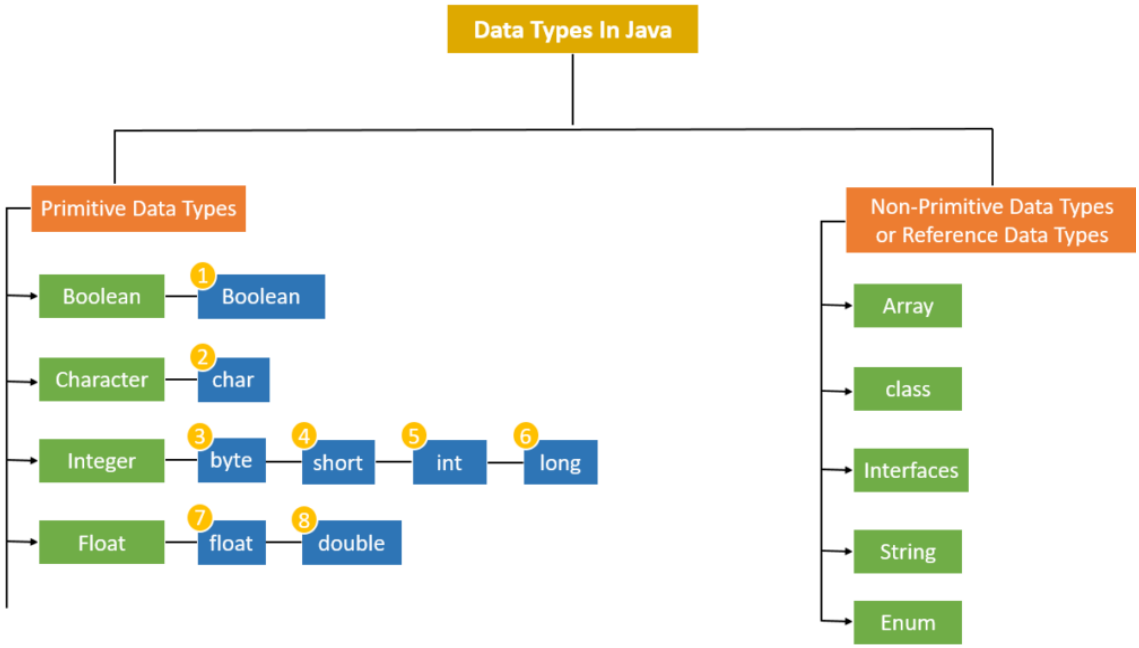
abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Data Types in Java

Data types specify the different sizes and values that can be stored in the variable.

There are two types of data types in Java:

1. **Primitive Data Types**
2. **Non-Primitive Data Types**.



Data Type	Default Value	Default size
char	'u0000'	2 bytes or 16 bits
boolean	FALSE	1 byte or 2 bytes
byte	0	1 byte or 8 bits
short	0	2 bytes or 16 bits
int	0	4 bytes or 32 bits
long	0	8 bytes or 64 bits
float	0.0f	4 bytes or 32 bits
double	0.0d	8 bytes or 64 bits

1. **Arrays:**
   A collection of similar types of data. For example, **int arr[] = new int[5];**.
2. **Classes:**
   User-defined data types. For example, **String**, **ArrayList**, etc.
3. **Interfaces:**
   Like classes but only contain method signatures.
   For example, **Comparable**, **Serializable**, etc.
4. **String :**

In Java, a string is a sequence of characters. It's a data type used to represent text rather than numeric data. Strings in Java are immutable, meaning once a string object is created, its contents cannot be changed.

Example :

```
// Declare String without using new operator
String s = "Welcome to VJIT !";
// Declare String using new operator
String s1 = new String("Welcome to CSE-Data Science !");
```

5. **Enum:**

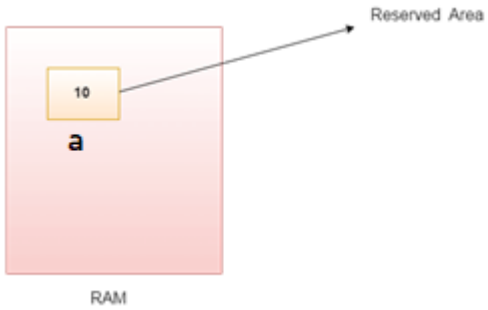
An enum is a special data type used to define a collection of constants. It allows you to create a set of named constants that represent a finite set of possibilities, typically related to some specific type or category. Enums are declared using the **enum** keyword.

Example :

```
enum Day
{
  SUNDAY,
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY
}
```

Variables

A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location.



int a=10; //Here a is variable

Scope and Life Time of Variables

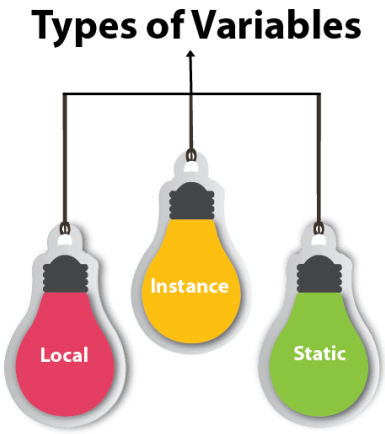
Scope of a variable refers to in which areas or sections of a program can the variable be accessed and

Lifetime of a Variable refers to how long the variable stays alive in memory.

Types of Variables

There are three types of variables in Java:

1. **Local Variables**
2. **Instance Variables**
3. **Static Variables**



1. Local Variables

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

File Name: **LocalVariableDemo.java**

```
public class LocalVariableDemo
{
   public static void main(String[] args)
   {
      //defining a Local Variable
      int num = 10;
      System.out.println(" Variable: " + num);
   }
}
```

Output:

```
Variable: 10
```

2. Instance Variables

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static.

It is called an instance variable because its value is instance-specific and is not shared among instances.

File Name: **InstanceVariableDemo.java**

```
import java.io.*;
public class InstanceVariableDemo
{
   //Defining Instance Variables
   public String name;
   public int age=19;
 //Creadting a default Constructor initializing Instance Variable
   public InstanceVariableDemo()
   {
      this.name = "Deepak";
   }
   public static void main(String[] args)
   {
      // Object Creation
     InstanceVariableDemo obj = new InstanceVariableDemo();
      System.out.println("Student Name is: " + obj.name);
      System.out.println("Age: "+ obj.age);
   }
}
```

Output:

```
Student Name is: Deepak
Age: 19
```

3. Static Variables

A variable that is declared as static is called a static variable also known as **class variables**. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

StaticVariableDemo.java

```
class Student
{
   //static variable
  static int age;
}
```

```
public class StaticVariableDemo
{
  public static void main(String args[])
{
      Student s1 = new Student();
      Student s2 = new Student();
      s1.age = 24;
      s2.age = 21;
      Student.age = 23;
      System.out.println("S1 age is: " + s1.age);
      System.out.println("S2  age is: " + s2.age);
  }
}
```

Output:

```
S1  age is: 23
S2  age is: 23
```

Operators in Java

Java operators are the symbols that are used to perform various operations on variables and its values.

Types of Operators

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Unary Operators
6. Bitwise Operators
7. Comparison Operators
8. Ternary Operator

Arithmetic Operators in Java

Arithmetic Operators are particularly used for performing arithmetic operations on given data or variables.

Operators	Operations
+	Addition
-	Subtraction
x	Multiplication
/	Division
%	Modulus

Assignment Operators

Assignment Operators are mainly used to assign the values to the variable in the program.

Operators	Examples	Equivalent to
=	X = Y;	X = Y;
+=	X += Y;	X = X + Y;
-=	X -= Y;	X = X - Y;
*=	X *= Y;	X = X * Y;
/=	X /= Y;	X = X / Y;
%=	X %= Y;	X = X % Y;

Relational Operators

Relational operators are assigned to check the relationships between two particular operators.

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

Logical Operators

Logical Operators in Java check whether the expression is true or false.

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

Unary Operators

Unary Operators in Java are used in only one operand.

Operators	Description
+	Unary Plus
-	Unary Minus
++	Increment operator
--	Decrement Operator
!	Logical complement operator

Bitwise Operators

Bitwise Operators in Java are used to assist the performance of the operations on individual bits.

Operators	Descriptions
~	Bitwise Complement
<<	Left shift
>>	Right shift
>>>	Unsigned Right shift
&	Bitwise AND
^	Bitwise exclusive OR

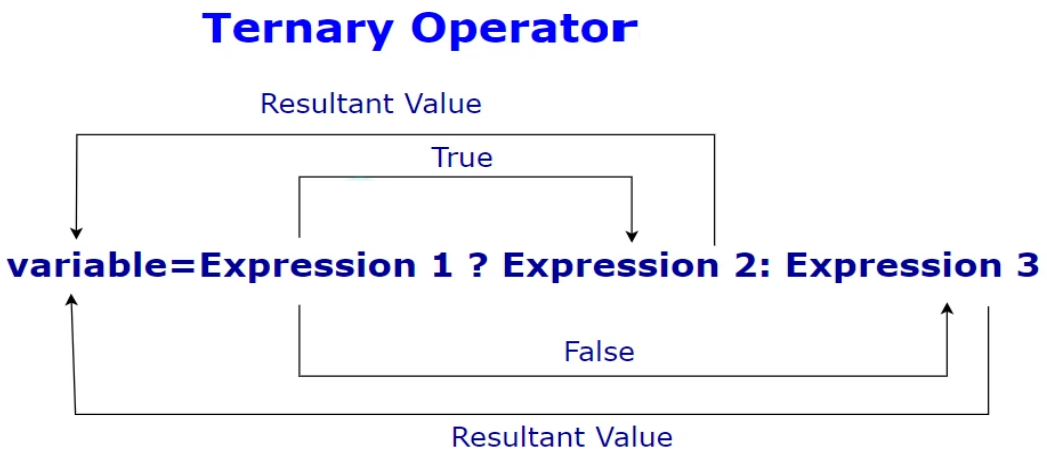
Comparison Operators

To compare two values (or variables), comparison operators are used. A comparison's return value is either true or false. These are referred to as "Boolean values."

Operators	Operations
==	Equal to
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Ternary Operator

The only conditional operator that accepts three operands is the ternary operator in Java. Java programmers frequently use it as a one-line alternative to the if-else expression.



Operator Precedence in Java

Category	Operator	Associativity
Primary	() [] -> . ++ -- new typeof sizeof checked unchecked default(T)	Left to Right
Unary	+ - ! ~ ++ -- (type)* & await	Right to Left
Multiplicative	* / %	Left to Right
Additive	+ -	Left to Right
Shift	<< >>	Left to Right
Relational and Type Testing	<<= >>= is as	Left to Right
Equality	== !=	Left to Right
Bitwise AND	&	Left to Right
Bitwise XOR	^	Left to Right
Bitwise OR	|	Left to Right
Logical AND	&&	Left to Right
Logical OR	||	Left to Right
Ternary	?:	Right to Left
Assignment and Lambda Expression	= += -= *= /= %= >>= <<= &= ^= |= =>	Right to Left

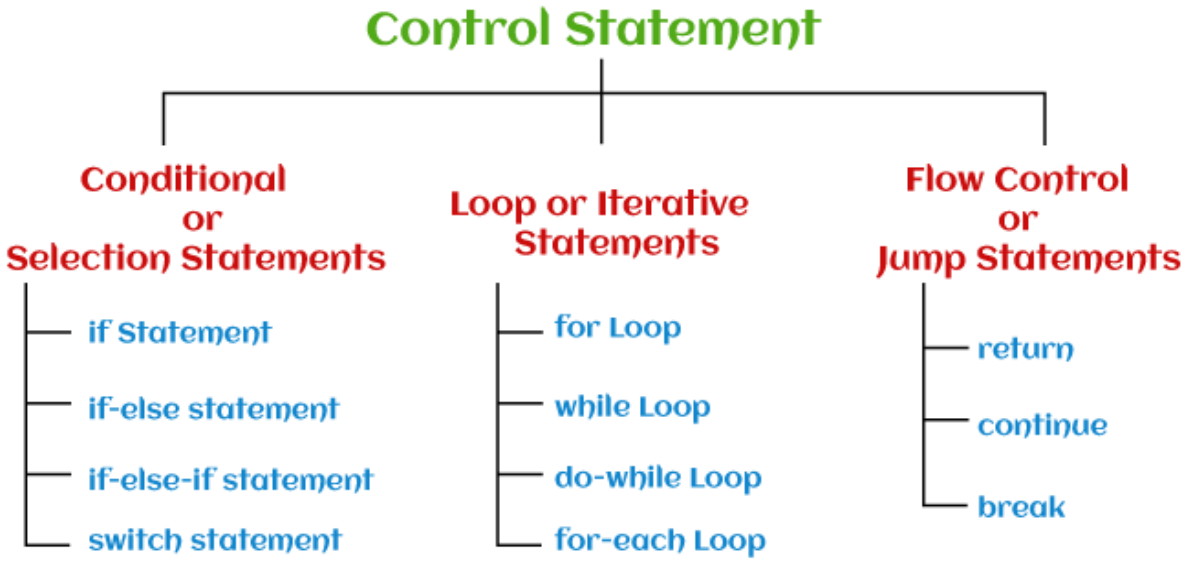
Expressions in Java

An expression in Java is a construct that evaluates to a single value. This value can be a number, a string, an object, or any other data type depending on the components of the expression. Expressions can include variables, literals, method calls, and operators.

```
double a = 2.2, b = 3.4, result;
result = a + b - 3.4;
```

Control Statements in Java

Control statements decide the flow (order or sequence of execution of statements) of a Java program. In Java, statements are parsed from top to bottom. Therefore, using the control flow statements can interrupt a particular section of a program based on a certain condition.



Types of Control Flow Statements

There are different types of control statements in Java for different conditions. We can divide control statements in Java into three major types:

1. **Conditional Statements**
2. **Looping Statements**
3. **Jumping Statements**

Conditional Statements

Conditional statements in Java are similar to making a decision in real life, where we have a situation and based on certain conditions we decide what to do next.

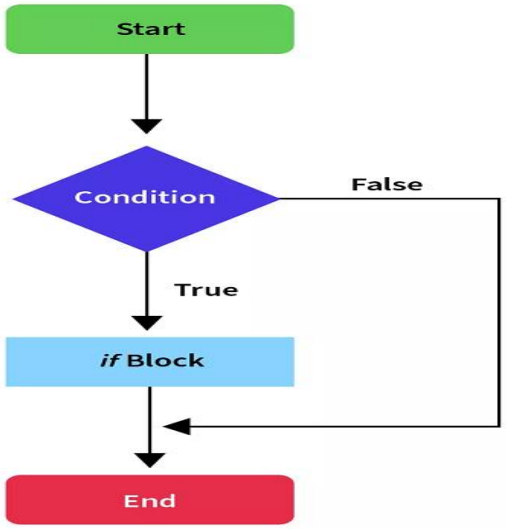
There are four types of decision-making statements in Java:

1. if Statement

These are the simplest and yet most widely used control statements in Java. The if statement is used to decide whether a particular block of code will be executed or not based on a certain condition.

If the condition is true, then the code is executed otherwise not.

Let's see the execution flow of the if statement in a flow diagram:



In the above flow diagram, we can see that whenever the condition is true, we execute the if block otherwise we skip it and **continue the execution with the code following the if block.**

Syntax:

```
if(condition)
{
   // block of code to be executed if the condition is true
}
```

For example:

```
// Java program to illustrate If statement
class Test
 {
   public static void main(String args[])
   {
      int i = 10;
      // using if statement
      if (i < 15)
         System.out.println("10 is less than 15");
      System.out.println("Outside if-block");
      // both statements will be printed
   }
}
```

Output

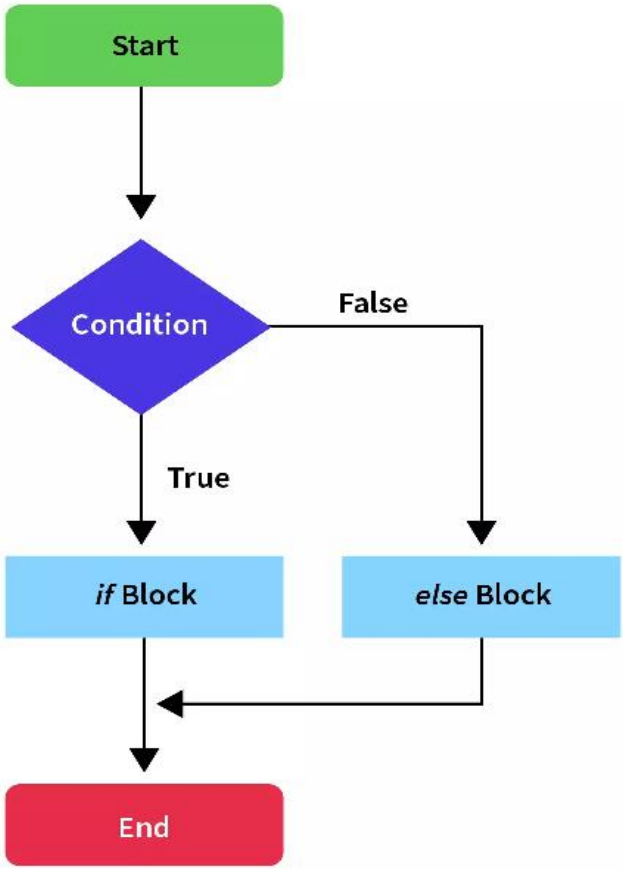
```
10 is less than 15

Outside if-block
```

2. if-else Statement

The if statement is used to execute a block of code based on a condition. But if the condition is false and we want to do some other task when the condition is false, how should we do it?

That's where else statement is used. In this, if the condition is true then the code inside the if block is executed otherwise the else block is executed.

Let's see the execution flow of the if-else statement in a flow diagram:



The above flow diagram is similar to the if statement, with a difference that whenever the condition is false, we execute the else block and then continue the normal flow of execution.

The syntax and execution flow of the if-else statement is as follows:

Syntax:

```
if (condition)
{
    // If block executed when the condition is true
}
else
{
    // Else block executed when the condition is false
}
```

Now, if the condition is false, the else block is executed and the if block code is skipped. This is one example of controlling the flow of a program through control statements in java.

For example:

```
// Java Program to demonstrate
// if-else statement
public class Test
 {

   public static void main(String[] args)
  {

      int n = 10;

      if (n > 5)
     {
         System.out.println("The number is greater than 5.");
      }
     else
     {
         System.out.println("The number is 5 or less.");
      }
   }
}
```

Output

```
The number is greater than 5.
```

3. Nested if-else Statement

Java allows us to **nest control statements within control statements**.

Nested control statements mean an if-else statement inside other if or else blocks. It is similar to an if-else statement but they are defined inside another if-else statement.

Let us see the syntax of a specific type of nested control statements where an if-else statement is nested inside another if block.

Syntax:

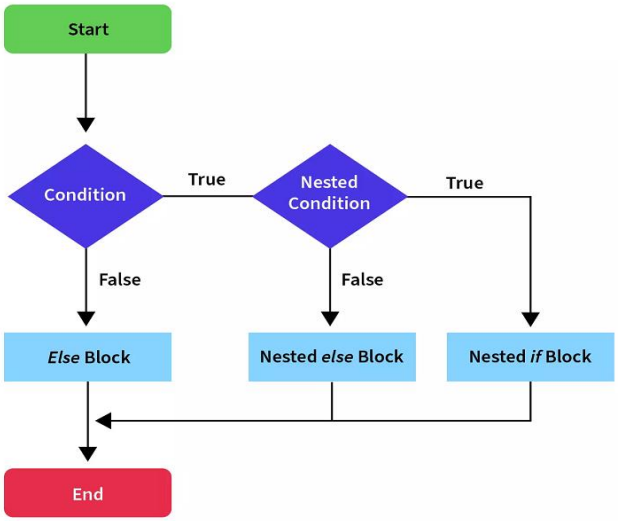
```
if (condition)
{
   // If block code to be executed if condition is true
   if (nested condition)
  {
       // If block code to be executed if nested condition is true
    }
    else
   {
       // Else block code to be executed if nested condition is false
    }
}
else
{
   // Else block code to be executed if condition is false
}
```

Explanation:

Here, we have specified another if-else block inside the first if block. In the syntax, you can see the series of the blocks executed according to the evaluation of condition and nested condition.

Using this, we can nest the control flow statements in Java to evaluate multiple related conditions.

Let's see the execution flow of the above-mentioned nested-if-else statement in a flow diagram:



Example:

Let's say that we want to know which floor of the mall a person needs to go to do shopping based on the age and gender of the person. Let's see the implementation of this example through nested-if:

```
int age = 20;
String gender = "male";
if (age > 18)
 {
   // person is an adult
   if (gender == "male")
   {
   // person is a male
   System.out.println("You can shop in the men's section on the 3rd Floor" );
   }
  else
  {
   // person is a female
   System.out.println("You can shop in the women's section on 2nd Floor");
   }
} else {
   // person is not an adult
   System.out.println("You can shop in the kid's section on 1st Floor");
}
```

Output:

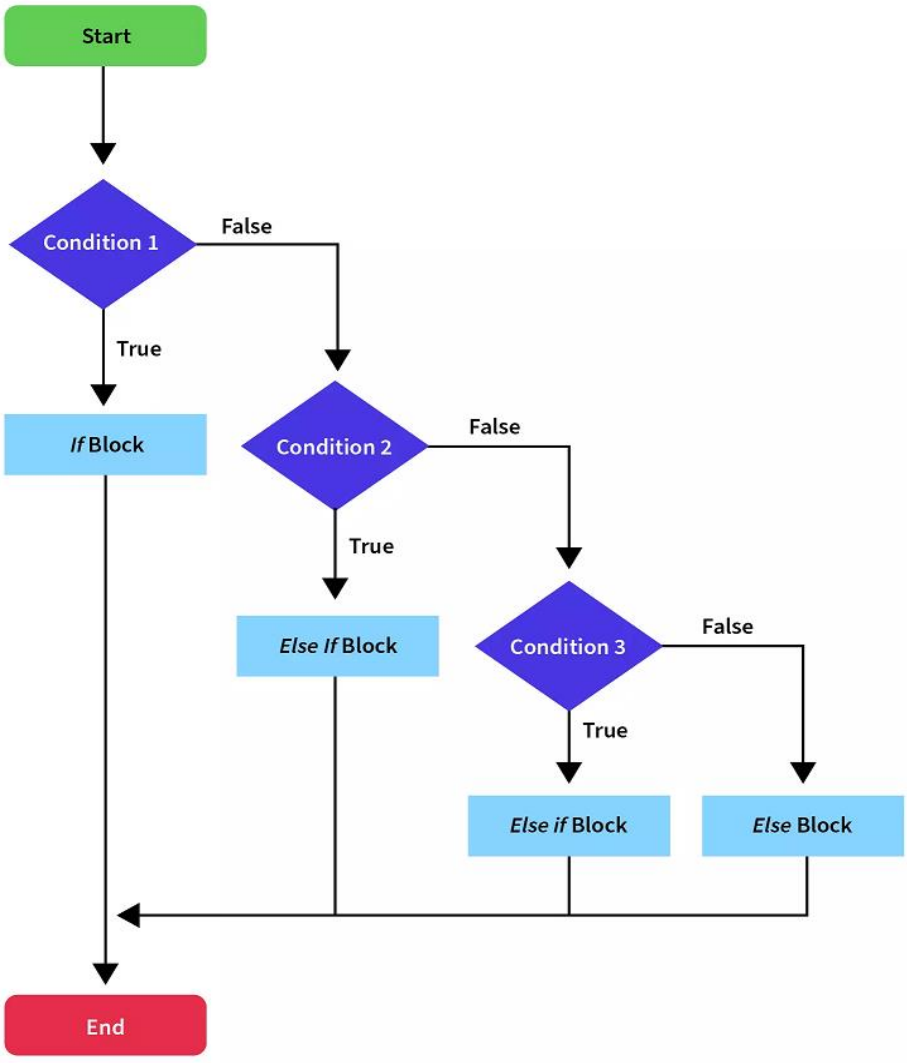
```
You can shop in the men's section on the 3rd Floor
```

4. if-else-if Ladder

In this, the if statement is followed by multiple else-if blocks. We can create a decision tree by using these control statements in Java in which the block where the condition is true is executed and the rest of the ladder is ignored and not executed.

If none of the conditions is true, the last else block is executed, if present.

Let's see the execution flow of the if-else ladder in a flow diagram:



As you can see in the above flow diagram of the if-else ladder, we execute the if block if the condition is true, otherwise if the condition is false, instead of executing the else block, we check other multiple conditions to determine which block of code to execute.

If none of the conditions are true, the last else block, if present, is executed.

The syntax and execution flow of the if-else-if ladder statement is as follows:

Syntax:

```
if(condition1)
{
   // Executed only when the condition1 is true
}
      else if(condition2)
     {
       // Executed only when the condition2 is true
     }
.
.
else
{
   // Executed when all the conditions mentioned above are true
}
```

Example:

Let's say that we have a browser-specific code, where we want to execute a code depending upon the browser the user is using. Let's try to implement this in the code:

```
String browser = "chrome";
if(browser == "safari")
 {
   System.out.println("The browser is safari");
}
else if(browser == "edge")
{
   System.out.println("The browser is edge");
}
else if(browser == "chrome")
{
   System.out.println("The browser is chrome");
}
else
 {
   System.out.println("Not a supported browser");
}
```

Output:

```
The browser is chrome
```

Explanation:

As you can see, the first two conditions are not true. Hence, the first two blocks are not executed. The third condition is true and hence the third block of code is executed giving the output: The browser is chrome. Following this block, all other blocks (the last else block) are ignored.

Here we can write the browser-specific implementation in the equivalent block, and if none of the supported browsers are encountered then we can also give an error in the last else block.

5. switch Statement

Switch statements are almost similar to the if-else-if ladder control statements in Java. It is a multi-branch statement. It is a bit easier than the if-else-if ladder and also more user-friendly and readable.

The switch statements have an expression and based on the output of the expression, one or more blocks of codes are executed.

These blocks are called cases. We may also provide a default block of code that can be executed when none of the cases are matched similar to the else block.

The syntax and execution flow of the switch statement is as follows:

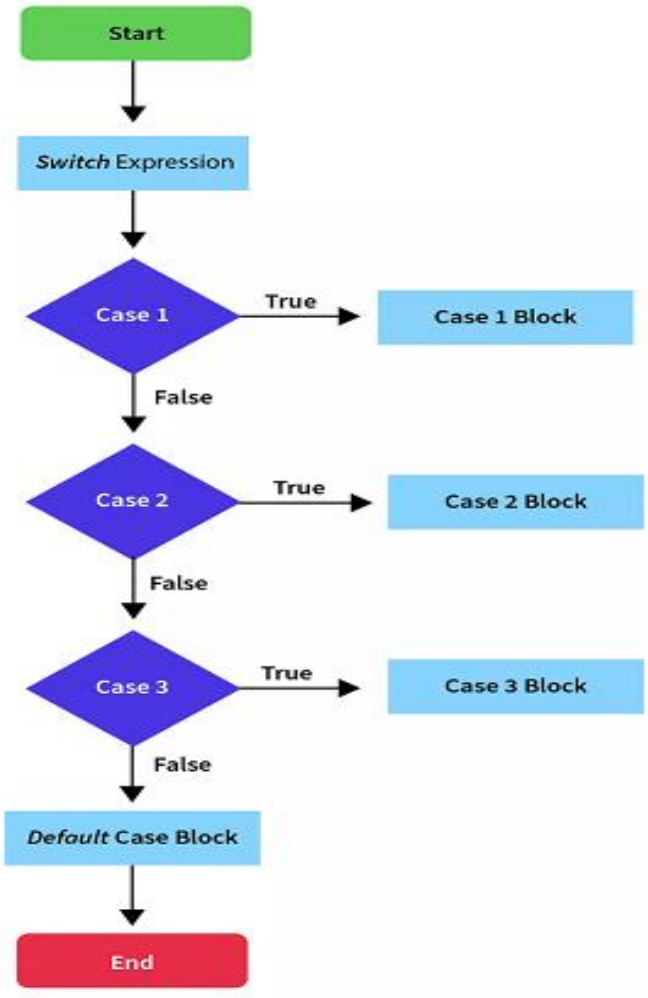
Syntax:

```
switch (expression)
{
  case value1:
     //code block of case with value1
     break;
  case value2:
     //code block of case with value2
     break;
  .
  .
  case valueN:
     //code block of case with valueN
     break;
  default:
    //code block of default value
}
```

There are certain points that one needs to be remembered while using switch statements:

- The expression can be of type String, short, byte, int, char, or an enumeration.
- We cannot have any duplicate case values.
- The default statement is optional.
- Usually, the break statement is used inside the switch to terminate a statement sequence.
- The break statement is optional. If we do not provide a break statement, the following blocks will be executed irrespective of the case value. This is known as the trailing case.

Let's see the execution flow of the switch statement in a flow diagram:



In the above flow diagram, we have a switch expression and we match the output of the expression through a series of case blocks.

Whichever case matches the output, its block is executed and execution skips to the end of the switch; otherwise, if none of the cases matches, the default block is executed.

Here, we have multiple case statements and each case code block is followed by a break statement to stop the execution to that case only.

Example:

If we take the previous example, we can easily implement that using switch statements:

```
String browser = "chrome";
switch (browser)
{
   case "safari":
      System.out.println("The browser is Safari");
      break;
   case "edge":
      System.out.println("The browser is Edge");
      break;
   case "chrome":
      System.out.println("The browser is Chrome");
      break;
   default:
      System.out.println("The browser is not supported");
}
```

Output:

```
The browser is chrome
```

In this example, the case with the value “chrome” is matched and hence its block is executed. If we do not give a break statement in this block, the trailing blocks(default block in this example) will also be executed.

Also, it's worthwhile to notice that the switch statement made the code more readable and cleaner.

Let’s see the output if we omit the break statement:

Code:

```
String browser = "chrome";
switch (browser)
{
   case "safari":
      System.out.println("The browser is Safari");
   case "edge":
```

```
        System.out.println("The browser is Edge");
    case "chrome":
        System.out.println("The browser is Chrome");
    default:
        System.out.println("The browser is not supported");
}
```

Output:

```
The browser is Chrome
The browser is not supported
```

We can see that the default block is also executed. It is due to the following reason: Once a case value is matched, all the following blocks of code are executed until a break statement or the end of the switch statement is encountered.

This is not the expected behavior and hence we should use the break statement. We'll learn more about the break statement in the next sections.

Looping Statements

Java provides a set of looping statements that **executes a block of code repeatedly** while some condition evaluates to true. Looping control statements in Java are used to traverse a collection of elements, like arrays.

Java provides the following looping statements:

1. while Loop

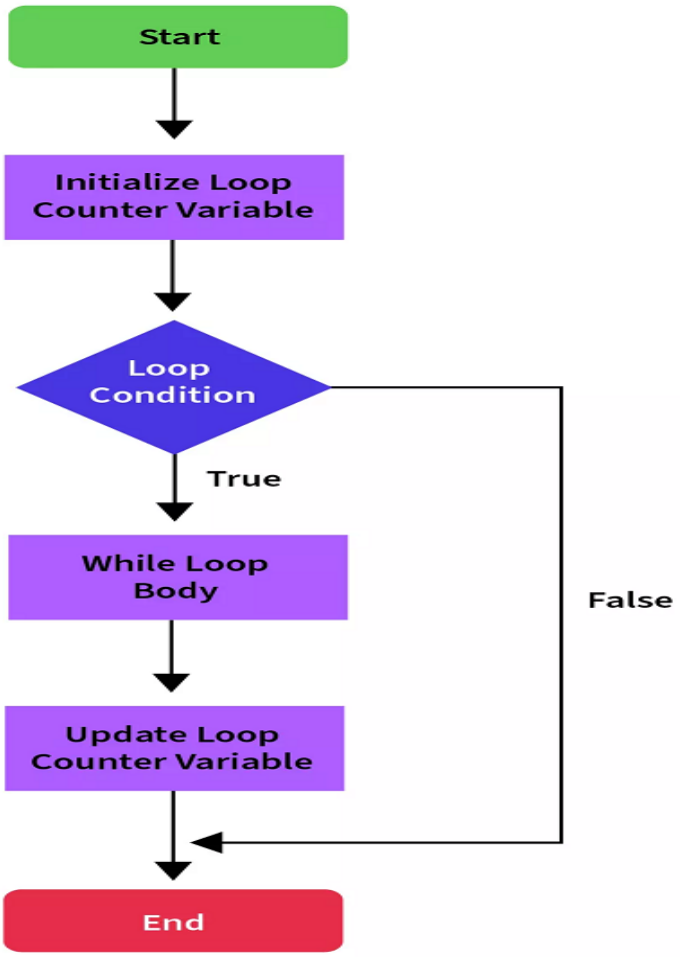
The while loop statement is the simplest kind of loop statement. It is used to iterate over a single statement or a block of statements until the specified boolean condition is false.

The while loop statement is also called the **entry-control looping statement** because the condition is checked prior to the execution of the statement and as soon as the boolean condition becomes false, the loop automatically stops.

You can use a while loop statement if the number of iterations is not fixed.

Normally the while loop statement contains an update section where the variables, which are involved in while loop condition, are updated.

Let's see the execution flow of the while loop statement in a flow diagram:



In the above flow diagram of a while loop:

- We initialize a loop counter variable. After that, we check the loop condition and if it's true, then the body of the loop is executed followed by the update of the counter variable.
- The control then again switches back to the loop condition and the cycle continues till the condition is false and we execute the statements outside the loop body.

The syntax and execution flow of the while loop statement is as follows:

Syntax:

```
while (condition)
{
   // code block to be executed
}
```

Example:

Let's say we want to print the numbers from 10 to 1 in decreasing order. Let's implement this through a while loop.

```
public class WhileLoopDemo
{

 public static void main(String args[])
 {
   int num = 10;
   while (num > 0)
   {
     System.out.println(num);

     // Update Section
     num--;
   }
  }
}
```

Output:

```
10
9
8
7
6
5
4
3
2
1
```

Here, we have used the loop condition as num > 0 and then at each iteration of the loop, we have decreased the value of the num variable by 1.

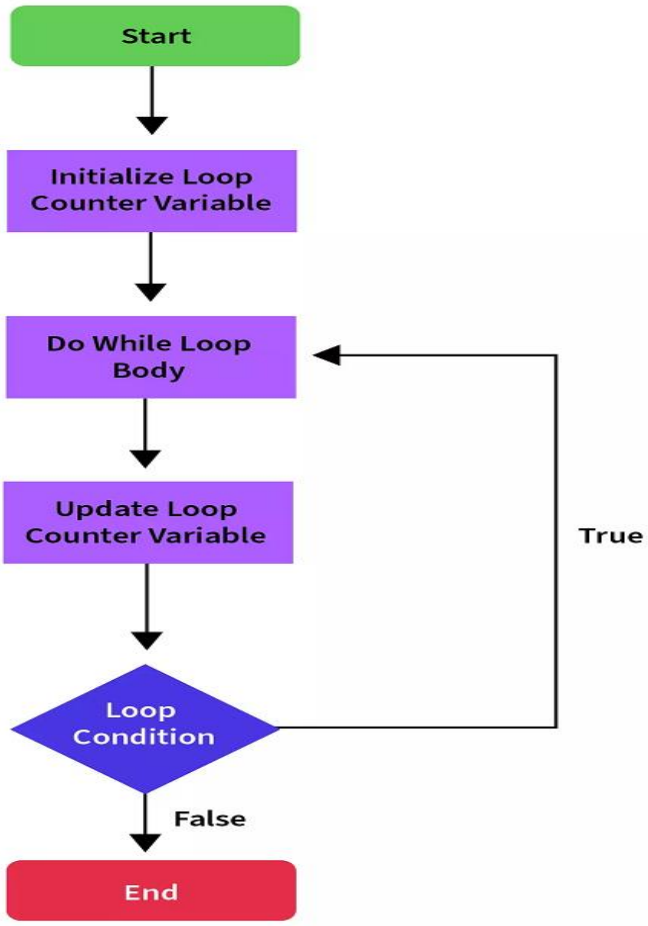
So the loop runs till the value of num becomes 1 from 10, and we get the desired output.

2. do-while Loop

The Java do-while loop statement works the same as the while loop statement with the only difference being that its boolean condition is evaluated post first execution of the body of the loop. Thus it is also called **exit controlled looping statement**.

You can use a do-while loop if the number of iterations is not fixed and the body of the loop has to be executed at least once.

Let's see the execution flow of the do-while loop statement in a flow diagram:



In the above flow diagram of a do-while loop:

- We also initialize a counter variable, but instead of checking the loop condition at the start, the body of the loop is executed.
- After the completion of the loop body, we check the loop condition and continue to execute the loop body till the condition is false when we come out of the loop and execute the rest of the code.

The syntax and execution flow of the do-while loop statement is as follows:

Syntax:

```
do
{
   // code block to be executed
} while (condition);
```

Example:

Let's try to use the same example of printing the number in decreasing order through a do-while loop:

```
public class Main
{

  public static void main(String args[])
 {
    int num = 10;
    do
   {
      System.out.println(num);
      num--;
    } while (num > 0);
  }
}
```

Output:

```
10
9
8
7
6
5
4
3
2
1
```

The implementation and output are almost similar, with the change that, even if the value of the num initially is less than 1. We'll get at least one print statement with the value of the num as the output.

3. for Loop

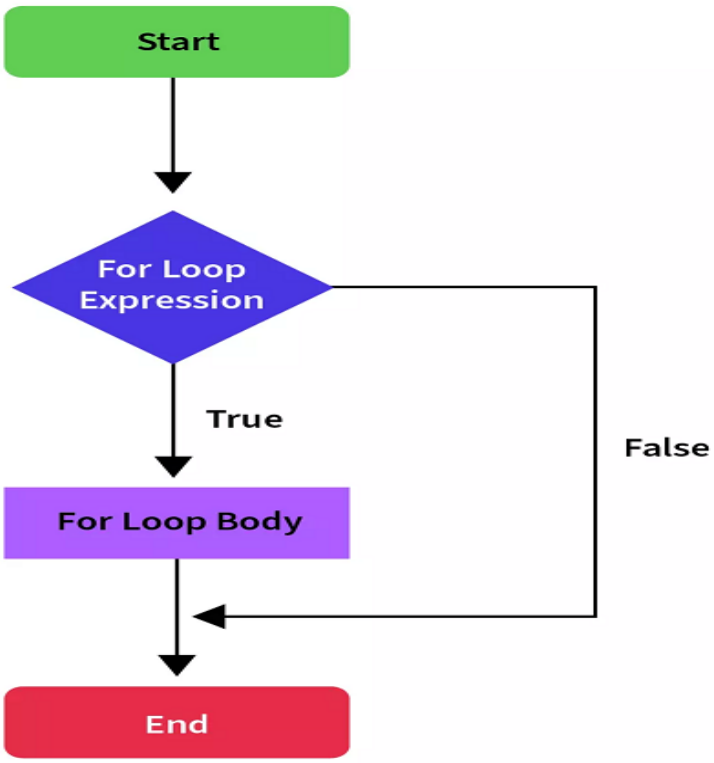
Unlike the while loop control statements in Java, a for loop statement consists of the initialization of a variable, a condition, and an increment/decrement value, all in one line. It executes the body of the loop until the condition is false.

The for loop statement is shorter and provides an easy way to debug structure in Java. You can use the for loop statement if the number of iterations is known.

In a for loop statement, execution begins with the initialization of the looping variable, then it executes the condition, and then it increments or decrements the looping variable.

If the condition results in true then the loop body is executed otherwise the for loop statement is terminated.

Let's see the execution flow of the for loop statement in a flow diagram:



As you can see in the above flow diagram, we have a for loop statement. In this statement, the loop condition is checked, and if the condition is true, the for loop body is executed until the condition is false and we continue with the normal flow of execution.

The syntax and execution flow of for loop statement is as follows:

Syntax:

```
for (initialization; condition; increment/decrement)
{
   // code block to be executed if condition is true
}
```

Example:

```
public class ForLoopDemo
{

  public static void main(String args[])
 {
    for (int num = 10; num > 0; num--) System.out.println(
      "The value of the number is: " + num
    );
  }
}
```

Output:

```
The value of the number is: 10
The value of the number is: 9
The value of the number is: 8
The value of the number is: 7
The value of the number is: 6
The value of the number is: 5
The value of the number is: 4
The value of the number is: 3
The value of the number is: 2
The value of the number is: 1
```

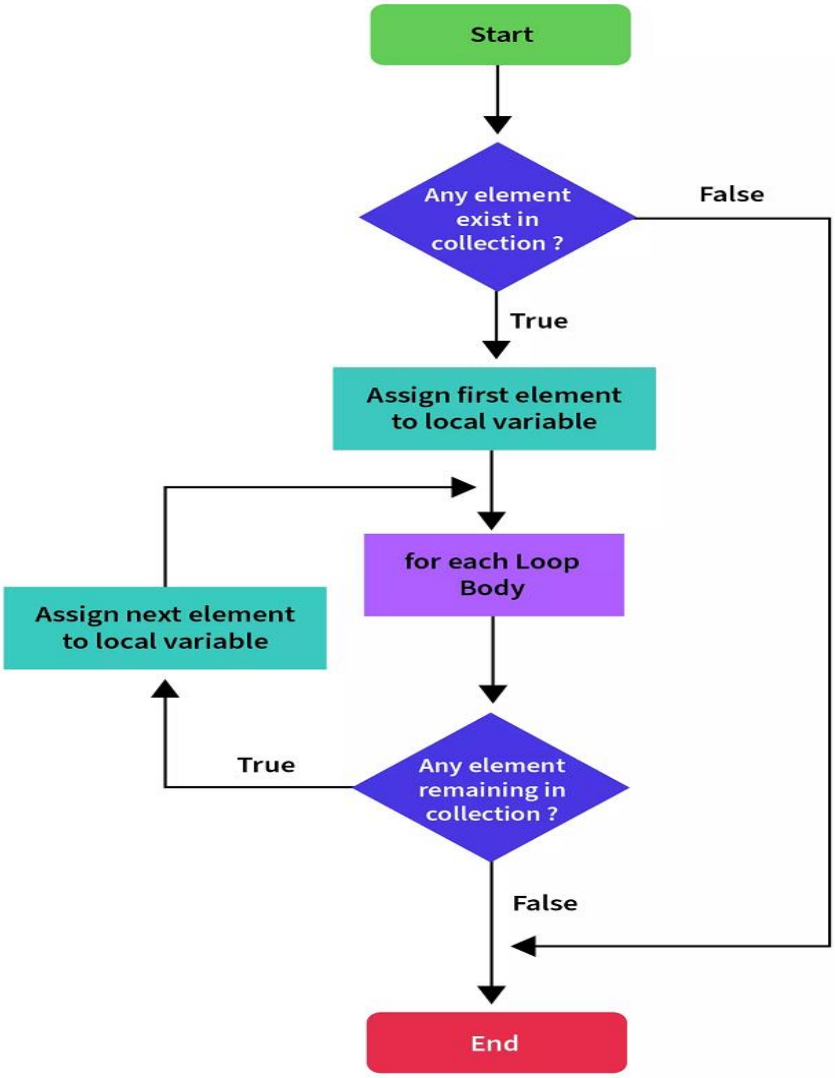
4. for-each Loop

The for-each loop statement provides an approach to traverse through elements of an array or a collection in Java. It executes the body of the loop for each element of the given array or collection. It is also known as the **Enhanced for loop statement** because it is easier to use than the for loop statement as you don't have to handle the increment operation. The major difference between the for and for-each loop is that for loop is a

general-purpose loop that we can use for any use case, the for-each loop can only be used with collections or arrays.

In for-each loop statement, you cannot skip any element of the given array or collection. Also, you cannot traverse the elements in reverse order using the for-each loop control statement in Java.

Let's see the execution flow of the for-each loop statement in a flow diagram:



In the above flow diagram of a for each loop:

- We check if the collection has any elements or not. If it has the elements, then the first element is assigned to the local variable mentioned in the for each expression, and the for each loop body is executed.
- After this, we again check if the collection has any remaining elements and this cycle continues till we have traversed all the elements.

The syntax and execution flow of for each loop statement is as follows:

Syntax:

```
for(dataType variableName : array | collection)
{
   // code block to be executed
}
```

Example:

```
public class ForEachLoopDemo
{

  public static void main(String args[])
 {
    int[] array = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
    System.out.println("Elements of the array are: ");
    for (int elem : array) System.out.println(elem);
  }
}
```

Output:

```
Elements of the array are:
10
9
8
7
6
5
4
3
2
1
```

Jump/Branching Statements

Jump/Branching control statements in Java transfer the control of the program to other blocks or parts of the program and hence are known as the branch or jump statements.

Java provides us the following jump or branching statements:

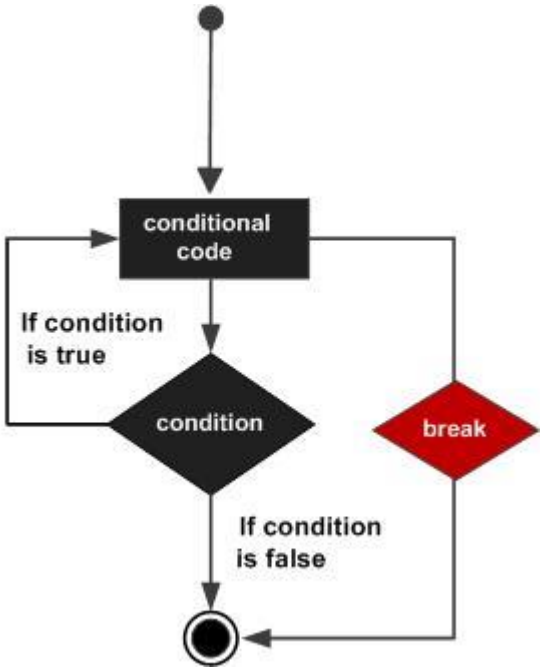
1. Break Statement

The break statement as we can deduce from the name is used to break the current flow of the program. The break statement is commonly used in the following three situations:

- Terminate a case block in a switch statement as we saw in the example of the switch statement in the above section.
- To exit the loop explicitly, even if the loop condition is true.
- Use as the alternative for the goto statement along with java labels, since java doesn't have goto statements.

The break statement cannot be used as a standalone statement in Java. It must be either inside a switch or a loop. If we try to use it outside a loop or a switch, JVM will give an error.

Let's see the execution flow of the break statement in a flow diagram:



In the above flow diagram of a break statement, whenever the loop body encounters a break statement, it stops the current flow of execution and jumps to the first statement out of the loop body.

The syntax of the break statement is as follows:

Syntax:

We have already seen how we use the break inside a switch. Let's see the syntax in the case of a loop:

```
for(condition)
 {
   // body of the loop
   break;
}
while(condition)
 {
   // body of the loop
   break;
}
```

Example 1:

```
for(int index = 0; index < 10; index++)
{
   System.out.println("The value of the index is: " + index);
   if(index == 3)
  {
      break;
   }
}
```

Output:

```
The value of the index is: 0
The value of the index is: 1
The value of the index is: 2
The value of the index is: 3
```

As we can see the loop was terminated even when the condition of the loop was still true. This is how we can use the break statement in a loop.

Example 2:

In case, we have nested loops, the break statement will only break the execution of the loop its part of. For **example**:

```
public class BreakStatementDemo
{

 public static void main(String args[])
 {
   for (int outer_index = 0; outer_index < 2; outer_index++)
   {
    System.out.println("The value of the outer index is: " + outer_index);

    for (int inner_index = 0; inner_index < 10; inner_index++)
    {
     System.out.println("The value of the inner index is: " + inner_index);

     if (inner_index == 3)
     {
      break;
     }
    }

    System.out.println("**********Inner loop ends**********");
   }
 }
}
```

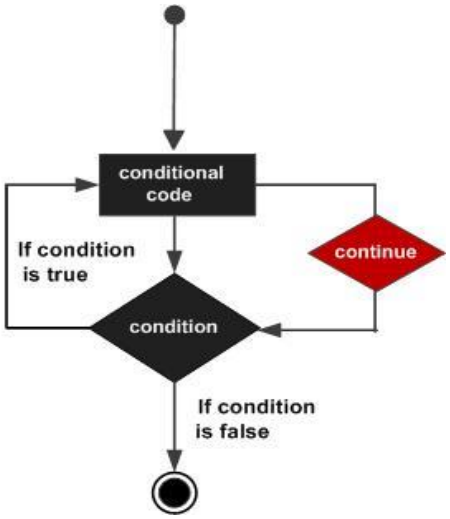
Output:

```
The value of the outer index is: 0
The value of the inner index is: 0
The value of the inner index is: 1
The value of the inner index is: 2
The value of the inner index is: 3
**********Inner loop ends**********
The value of the outer index is: 1
The value of the inner index is: 0
The value of the inner index is: 1
The value of the inner index is: 2
The value of the inner index is: 3
**********Inner loop ends**********
```

2. Continue Statement

Sometimes there are situations where we just want to ignore the rest of the code in the loop body and continue from the next iteration. The continue statement in Java allows us to do just that. This is similar to the **break** statement in the sense that it **bypasses** every line in the loop body after itself, but instead of exiting the loop, it goes to the next iteration.

Let's see the execution flow of the continue statement in a flow diagram:



In the above flow diagram of a continue statement, whenever the continue statement has encountered the rest of the loop body is skipped and the next iteration is executed if the loop condition is true.

The syntax and execution flow of the continue statement is as follows:

Syntax:

```
for(condition)
 {
   // body of the loop
   continue;
   //the statements after this won't be executed
}
while(condition)
 {
   // body of the loop
   continue;
   // the statements after this won't be executed
}
```

Example:

Let's try to print the odd number between 1 to 10 as we did in the example of the for loop, but this time we'll use the continue statement:

```
System.out.println("The odd numbers between 1 to 10 are: ");
for (int number = 1; number <= 10; number++)
 {
    if (number % 2 == 0) continue;
    System.out.println(number);
}
```

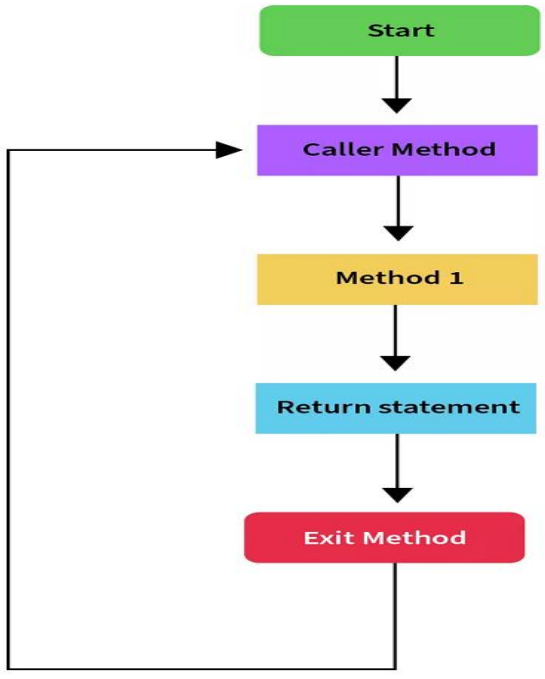
Output:

```
The odd numbers between 1 to 10 are:
1
3
5
7
9
```

3. Return Statement

The return statements are used when we need to return from a method explicitly. The return statement transfers the control back to the caller method of the current method. In the case of the main method, the execution is completed and the program is terminated. Return statements are often used for conditional termination of a method or to return something from the method to the caller method.

Let's see the execution flow of the return statement in a flow diagram:



As you can see in the above flow diagram, whenever a return statement is encountered anywhere in a method, the execution of the current method is stopped and the flow of the program returns to the caller method of the current method.
The syntax and execution flow of the return statement is as follows:

Syntax:

```
void method()
{
   // body of the method
   return;
}
```

Example:

Let's say we are searching an element in a list, and as soon as we find it, our work is done and we should exit the function. Let's see the implementation of this problem:

```
public class ReturnStatementDemo
{
  public static String search(int key)
  {
    int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    for (int element : numbers)
   {
      if (element == key)
      {
        return "Success";
        // Putting statements post this return statement
        // Will throw compile-time error
      }
    }
    return "Failure";
  }
  public static void main(String[] args)
  {
    System.out.println(search(3));
    System.out.println(search(10));
  }
}
```

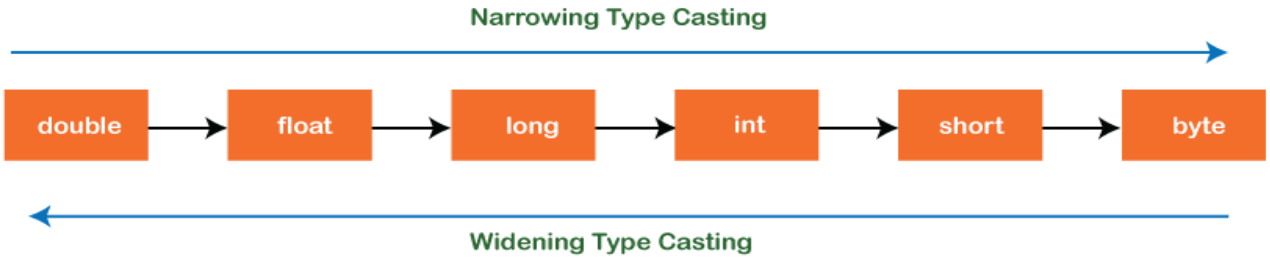
Output:

```
Success
Failure
```

As we can search that the value 3 was in the array and we get the output as "Success", but since value 10 does not exist in the array, we get the "Failure" as the output.

Type Conversion and Casting

Type casting in Java is a fundamental concept that allows developers to convert data from one data type to another. It is essential for handling data in various situations, especially when dealing with different types of variables, expressions, and methods. In Java, type casting is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer.



Type Casting in Java

Rules of Typecasting

Widening Conversion (Implicit)

- No explicit notation is required.
- Conversion from a smaller data type to a larger data type is allowed.
- No risk of data loss.

Narrowing Conversion (Explicit)

- Requires explicit notation using parentheses and casting.
- Conversion from a larger data type to a smaller data type is allowed.
- Risk of data loss due to truncation.

1. Widening Type Casting

Converting a lower data type into a higher one is called **widening** type casting. It is also known as **implicit conversion** or **casting down**. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

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

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

WideningExample.java

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

Output

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

2. Narrowing Type Casting

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

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

NarrowingExample.java

```
public class NarrowingExample
{
   public static void main(String args[])
   {
    double d = 166.66;
    //converting double data type into long data type
    long l = (long)d;
    //converting long data type into int data type
    int i = (int)l;
    System.out.println("Before conversion: "+d);
    //fractional part lost
    System.out.println("After conversion into long type: "+l);
    //fractional part lost
    System.out.println("After conversion into int type: "+i);
    }
}
```

Output

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

Arrays in Java

An array is a collection of similar types of data.

1. Array Declaration

To declare an array in Java, use the following syntax:

```
type[] arrayName;
```

- **type**: The data type of the array elements (e.g., int, String).
- **arrayName**: The name of the array.

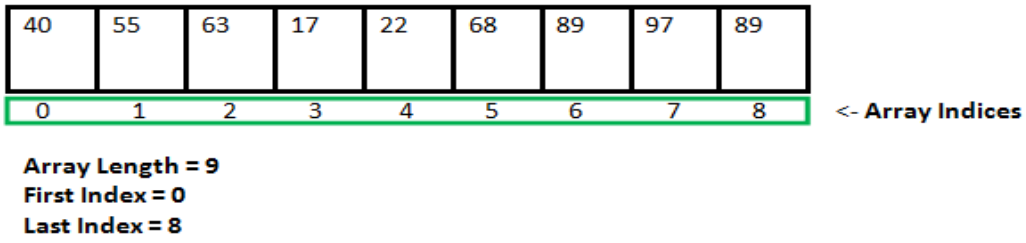
2. Create an Array

To create an array, you need to allocate memory for it using the **new** keyword:

// Creating an array of 9 integers

int[] numbers = new int[9];

This statement initializes the numbers array to hold 5 integers. The default value for each element is 0.



3. Access an Element of an Array

We can access array elements using their index, which starts from 0:

// Setting the first element of the array
numbers[0] = 10;

// Accessing the first element
int firstElement = numbers[0];

The first line sets the value of the first element to 10. The second line retrieves the value of the first element.

4. Change an Array Element

To change an element, assign a new value to a specific index:

// Changing the first element to 20
numbers[0] = 20;

5. Array Length

We can get the length of an array using the length property:

// Getting the length of the array
int length = numbers.length;

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

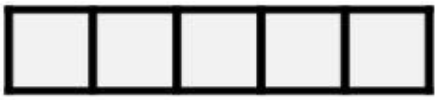
Single-Dimensional Array in Java

A single-dimensional array in Java is a linear collection of elements of the same data type. It is declared and instantiated using the following syntax:

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

Instantiation of an Array in Java

arrayRefVar=new datatype[size];



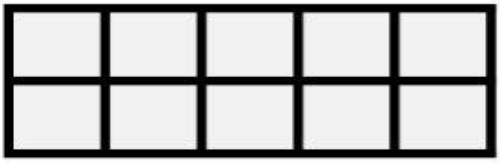
```
int Array = new int[5];
```

Program to calculate sum of array elements

```
public class Sum
 {
  public static void main(String[] args)
  {
    int[] myArray = {1, 5, 10, 25};
    int sum = 0;
    int i;
      // Loop through array elements and get the sum
    for (i = 0; i < myArray.length; i++)
     {
        sum = sum+myArray[i];
      }
    System.out.println("The sum is: " + sum);
  }
}
```

2. Two-Dimensional Array

Two-dimensional arrays store the data in rows and columns:



```
int[][] Array = new int[2][5];
```

In this, the array has two rows and five columns. The index starts from 0,0 in the left-upper corner to 1,4 in the right lower corner.

Program to calculate addition of two matrices

```
public class MatrixAdditionExample
{
    public static void main(String args[])
    {
        //creating two matrices
        int a[][]={{1,3,4},{2,4,3},{3,4,5}};
        int b[][]={{1,3,4},{2,4,3},{1,2,4}};

        //creating another matrix to store the sum of two matrices
        int c[][]=new int[3][3];  //3 rows and 3 columns

        //adding and printing addition of 2 matrices
        for(int i=0;i<3;i++)
        {
            for(int j=0;j<3;j++)
            {
              c[i][j]=a[i][j]+b[i][j];
              System.out.print(c[i][j]+" ");
            }
        System.out.println();//new line
        }
    }
}
```

Output:

```
2 6 8
4 8 6
4 6 9
```

Simple Java Program

Steps to Implement Java Program

Implementation of a Java application program involves the following steps. They include:

1. Creating the program
2. Compiling the program
3. Running the program

1. Creating Programs in Java

We can create a program using Text Editor (Notepad++) or Eclipse IDE

```
// This is a simple Java program.
// FileName : "HelloWorld.java".
class HelloWorld
 {
   // Your program begins with a call to main().
   // Prints "Hello, World" to the terminal window.
   public static void main(String[] args)
   {
      System.out.println("Hello, World");
   }
}
```

File Save: d:\ *HelloWorld.java*

1. **Class Definition**

This line uses the keyword **class** to declare that a new class is being defined.

```
class HelloWorld
 {
   //
   //Statements
}
```

2. ***HelloWorld***

It is an identifier that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace " **{** " and the closing curly brace " **}** ".

3. ***main Method***

In the Java programming language, every application must contain a main method. The main function(method) is the entry point of your Java application, and it's mandatory in a Java program. whose signature in Java is:

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

Explanation of the above syntax

- ***public*** *: So that JVM can execute the method from anywhere.*
- ***static*** *: The main method is to be called without an object. The modifiers are public and static can be written in either order.*
- ***void*** *: The main method doesn't return anything.*
- ***main()*** *: Name configured in the JVM. The main method must be inside the class definition. The compiler executes the codes starting always from the main method.*
- ***String[]*** *: The main method accepts a single argument, i.e., an array of elements of type String.*

Like in C/C++, the main method is the entry point for your application and will subsequently invoke all the other methods required by your program.

The next line of code is shown here. Notice that it occurs inside the main() method.

```
System.out.println("Hello, World");
```

This line outputs the string "Hello, World" followed by a new line on the screen. Output is accomplished by the built-in println( ) method. The **System** is a predefined class that provides access to the system and **out** is the variable of type output stream connected to the console.

Comments

They can either be multiline or single-line comments.

```
// This is a simple Java program.
// Call this file "HelloWorld.java".
```

This is a single-line comment. This type of comment must begin with // as in C/C++. For multiline comments, they must begin from /* and end with */.

Important Points

- *The name of the class defined by the program is HelloWorld, which is the same as the name of the file(HelloWorld.java). This is not a coincidence. In Java, all codes must reside inside a class, and there is at most one public class which contains the main() method.*

- *By convention, the name of the main class(a class that contains the main method) should match the name of the file that holds the program.*
- *Every Java program must have a class definition that matches the filename (class name and file name should be same).*

Compiling the Program

- After successfully setting up the environment, we can open a terminal in Windows and go to the directory where the file – HelloWorld.java is present.
- Now, to compile the HelloWorld program, execute the compiler – javac, to specify the name of the **source** file on the command line, as shown:

```
javac HelloWorld.java
```

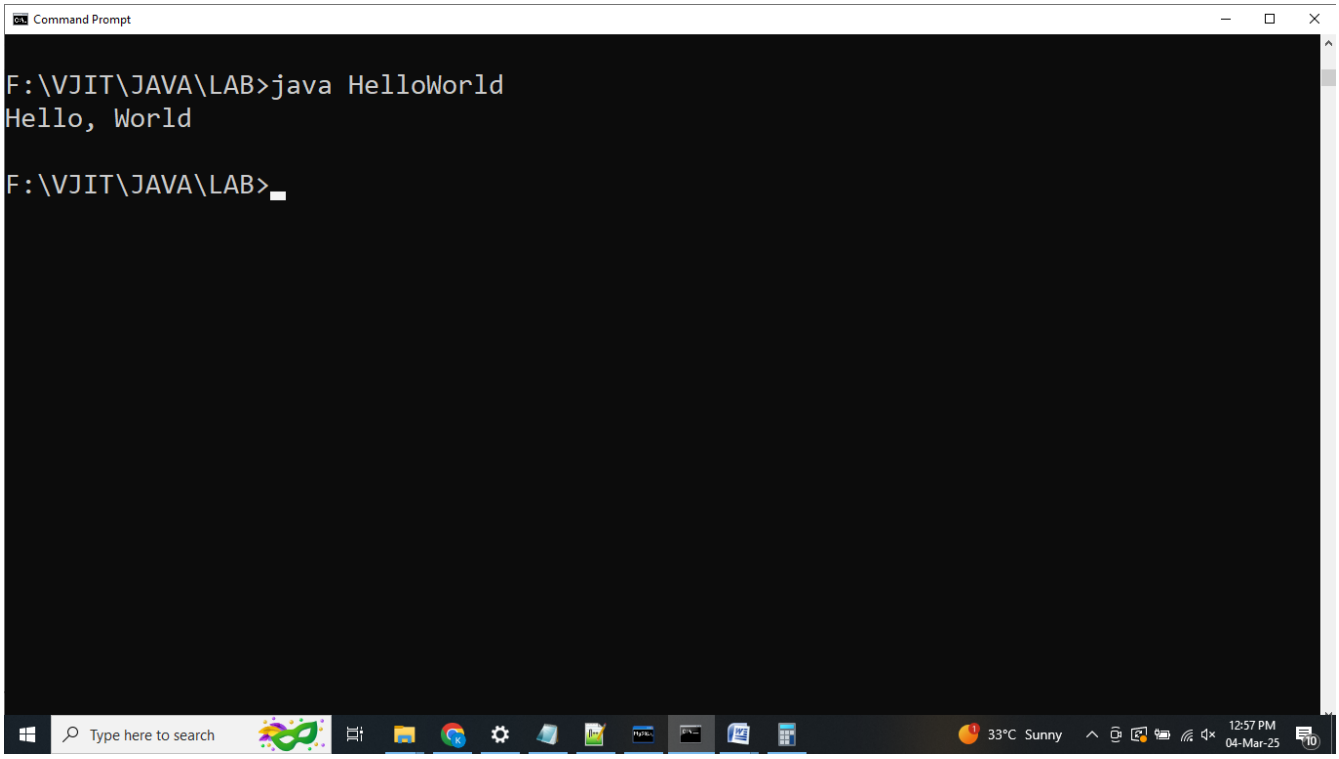
- The compiler creates a HelloWorld.class (in the current working directory) that contains the bytecode version of the program.

Execute the Program

- Now, to execute our program, **JVM** (Java Virtual Machine) needs to be called using Java tool, specifying the name of the **class** file on the command line, as shown:

```
java HelloWorld
```

- This will print "Hello World" to the terminal screen.



Fundamentals of Object Oriented Programming PART-2

Object-Oriented Paradigm, Basic Concepts of Object Oriented Programming, Applications of OOP. Concepts of classes, objects, constructors, methods, access control, this keyword, garbage collection, overloading methods and constructors, parameter passing, recursion, static keyword, nested and inner classes, Strings, Object class.

Object-Oriented Paradigm

The object-oriented paradigm in Java is a programming model that uses objects to represent and manipulate data. It's a widely used approach in software development.

Key concepts of Object-Oriented Programming

- **Encapsulation**

  Wrapping up of data into a single unit i.e Bundles data and the methods place it into a class. This protects data from being accessed by other classes.
- **Abstraction**

  Showing the essential features and hiding their implementation details i.e Exposes only the essential information of an object to the user, hiding their implementation details.
- **Inheritance**

  Acquire the properties from one class to other class i.e allows objects to inherit properties and behaviors from other objects.
- **Polymorphism**

  It means many forms i.e the ability of an object to take on multiple forms, meaning a single method can behave differently depending on the object it is called on, allowing you to treat objects of different types as if they were the same type, as long as they share a common interface or inheritance relationship; essentially, it enables a single action to be performed in different ways depending on the context, making code more flexible and reusable.

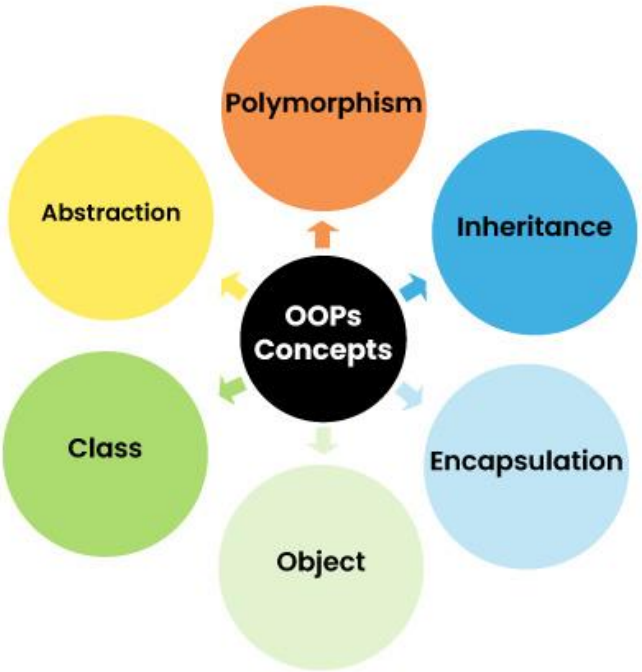
Benefits of Object-Oriented Programming

- It enables developers to write modular, reusable, and maintainable code.
- It allows objects to interact with each other to create powerful applications.
- It helps developers create scalable Java applications.

Basic Concepts of Object Oriented Programming

Object means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by following the basic principles of OOPs

- Object
- Class
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism



- **Object:** *It represents real-world entities.*

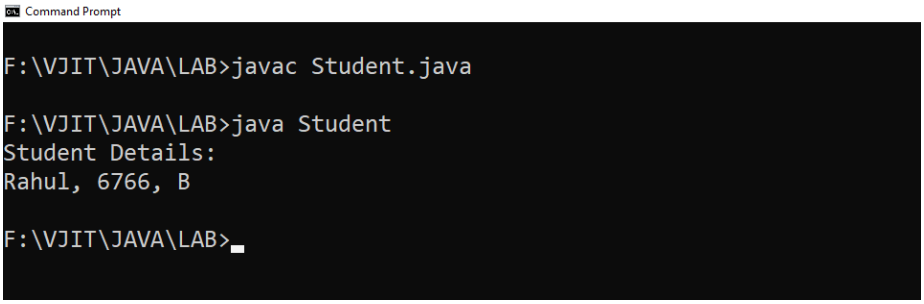
  It has two characteristics (**state and behavior**). Some of the real-world objects are book, mobile, table, computer, etc. An object is a variable of the type **class**, it is a basic component of an object-oriented programming system. A class has the methods and data members (attributes), these methods and data members are accessed through an object. Thus, an **object is an instance of a class**.

An object mainly consists of:

1. **State**: It is represented by the attributes of an object. It also reflects the properties of an object.

2. **Behavior**: It is represented by the methods of an object. It also reflects the response of an object to other objects.
3. **Identity**: It is a unique name given to an object that enables it to interact with other objects.
4. **Method:** A method is a collection of statements that perform some specific task and return the result to the caller.

```
// create a Student class
public class Student
{
        // Declaring attributes
        String name;
        int rollNo;
        String section;

          // initialize attributes
          Student(String name, int rollNo, String section)
          {
            this.name= name;
            this.rollNo = rollNo;
            this.section = section;
          }
         // print details
         public void printDetails()
         {
           System.out.print("Student Details: ");
           System.out.println(name+ ", " + rollNo + ", " + section);
         }

   public static void main(String[] args)
   {
            // create student objects
            Student st = new Student("Rahul", 6766, "B");

            // print student details
            st.printDetails();
  }
}
```



- **Class:** *Blueprint of an objects or Collection of Objects*

In object-oriented programming, a class is a blueprint from which individual objects are created. In Java, everything is related to classes and objects. Each class has its methods and attributes that can be accessed and manipulated through the objects.

class declaration can include the following components in order:

1. **Modifiers**: A class can be public or have default access
2. **Class name:** The class name should begin with the initial letter capitalized by convention.
3. **Body:** The class body is surrounded by braces, { }.

```
// create a Student class
public class Student
{
  // Declaring attributes
  String name;
  int rollNo;
  String section;

  // initialize attributes
  Student(String name, int rollNo, String section)
  {
    this.name= name;
    this.rollNo = rollNo;
    this.section = section;
  }
```

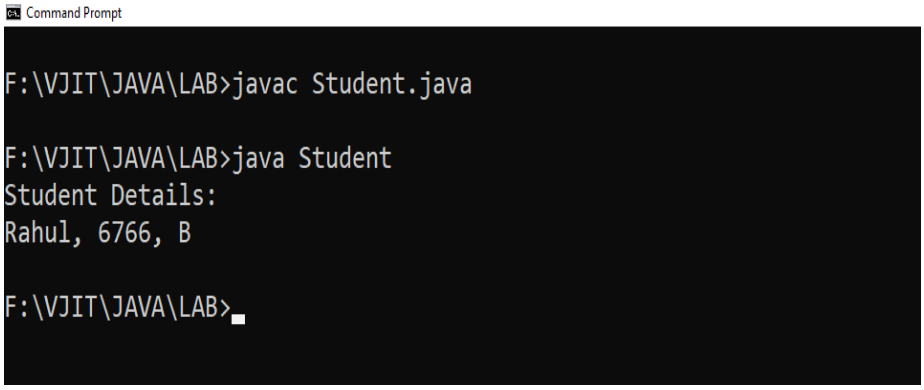
```
    // print details
    public void printDetails()
    {
      System.out.print("Student Details: ");
      System.out.println(name+ ", " + rollNo + ", " + section);
    }
}
```

- **Encapsulation:** *Wrapping up of data into a single unit*

In an object-oriented approach, encapsulation is a process of binding the data members (attributes) and methods together. The encapsulation restricts direct access to important data. The best example of the encapsulation concept is making a class where the data members are private and methods are public to access through an object. In this case, only methods can access those private data.

```
// create a Student class
public class Student
 {
  // Declaring private attributes
  private String name;
  private int rollNo;
  private String section;

  // initialize attributes
  Student(String name, int rollNo, String section)
  {
    this.name= name;
    this.rollNo = rollNo;
    this.section = section;
  }
  // print details
  public void printDetails()
  {
    System.out.println("Student Details: ");
    System.out.println(this.name+ ", " + this.rollNo + ", " + section);
  }
```

```
    public static void main(String[] args)
    {
      // create student objects
      Student st = new Student("Rahul", 6766, "B");

      // print student details
      st.printDetails();

    }
}
```



- **Abstraction:** *Showing the Essential Features and hiding their implementation details*

In object-oriented programming, an abstraction is a technique of hiding internal details and showing functionalities. The abstract classes and interfaces are used to achieve abstraction in Java.

The real-world example of an abstraction is a Car, the internal details such as the engine, process of starting a car, process of shifting gears, etc. are hidden from the user, and features such as the start button, gears, display, break, etc are given to the user. When we perform any action on these features, the internal process works.

```
abstract class Vehicle
{
  public void startEngine()
  {
    System.out.println("Engine Started");
  }
}
```

```
public class Car extends Vehicle
 {
   private String color;
   public Car(String color)
   {
     this.color = color;
   }
   public void printDetails()
   {
     System.out.println("Car color: " + this.color);
   }
   public static void main(String[] args)
   {
     Car car = new Car("White");
     car.printDetails();
     car.startEngine();
   }
}
```

```
F:\VJIT\JAVA\LAB>javac Car.java

F:\VJIT\JAVA\LAB>java Car
Car color: White
Engine Started

F:\VJIT\JAVA\LAB>
```

- **Inheritance :** *Acquire the properties from one class to other class*

In object-oriented programming, inheritance is a process by which we can reuse the functionalities of existing classes to new classes.

We are achieving inheritance by using **extends** keyword. Inheritance is also known as "**is-a**" relationship.

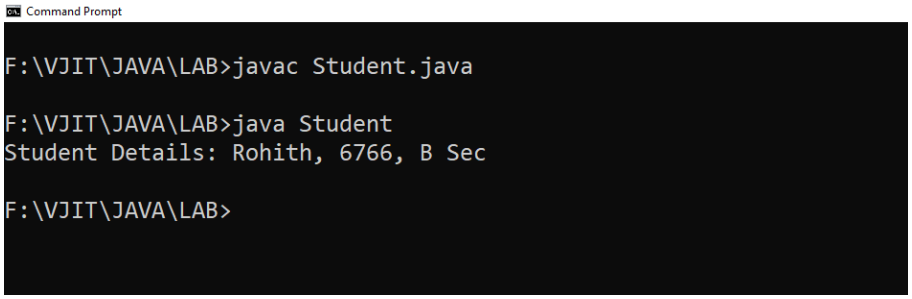
Let us discuss some frequently used important terminologies:

- **Superclass:** The class whose features are inherited is known as superclass (also known as base or parent class).

- **Subclass:** The class that inherits the other class is known as subclass (also known as derived or extended or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

```
// base class for all students
class Person
 {
        String name;
        Person(String name)
        {
          this.name = name;
        }
}
// create a Student class
public class Student extends Person
 {
       // Declaring attributes
       int rollNo;
       String section;
       // initialize attributes
         Student(String name, int rollNo, String section)
          {
                 super(name);
                 this.rollNo = rollNo;
                 this.section = section;
         }
   // print details
   public void printDetails()
   {
     System.out.print("Student Details: ");
     System.out.println(this.name+ ", " + this.rollNo + ", " + section);
   }
```

```
    public static void main(String[] args)
     {
            // create student objects
            Student st = new Student("Rohith", 6766, "B Sec");
             // print student details
            st.printDetails();
     }
}
```



- **Polymorphism**

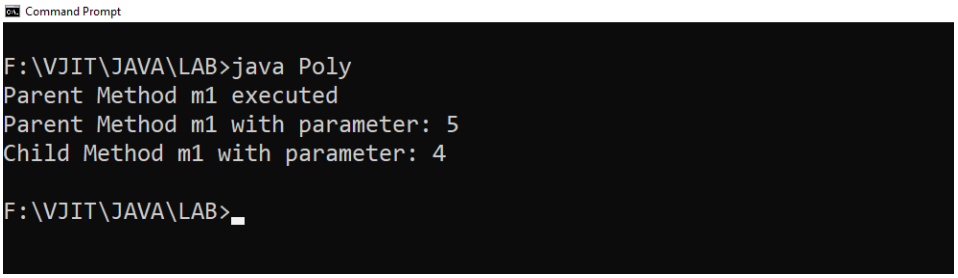
The term "polymorphism" means "many forms". In object-oriented programming, polymorphism is useful when you want to create multiple forms with the same name of a single entity.

Polymorphism in Java is mainly of **2 types** as mentioned below:

1. Method Overloading
2. Method Overriding

1. **Method Overloading:** Also, known as **compile-time** polymorphism, is the concept of Polymorphism where more than one method share the same name with different signature (Parameters) in a class. The return type of these methods can or cannot be same.
2. **Method Overriding:** Also, known as **run-time** polymorphism, is the concept of Polymorphism where method in the child class has the same name, return-type and parameters as in parent class. The child class provides the implementation in the method already written in the parent class.

```
// Method Overloading and Overriding
// Parent Class
class Parent
 {
   // Method Declared
   public void m1()
       {
      System.out.println("Parent Method m1 executed");
   }
   // Method Overloading
   public void m1(int a)
       {
      System.out.println("Parent Method m1 with parameter: " + a);
   }
}

// Child Class
class Child extends Parent
 {
   // Method Overriding
    public void m1(int a)
  {
          System.out.println("Child Method m1 with parameter: " + a);
   }
}
// Main Method
class Poly
 {
   public static void main(String args[])
   {
      Parent p = new Parent();
      p.m1();
      p.m1(5);
      Child c = new Child();
      c.m1(4);
   }
}
```



Applications of OOPs

Java is one of the most prominent programming languages that fully embrace the OOP paradigm. The application of OOP in Java is extensive, spanning various types of software, including desktop applications, web applications, enterprise software, mobile apps, and games. Java's rich set of libraries, frameworks, and tools are built on OOP principles, making it an ideal choice for developing scalable and maintainable applications.

Application Area	Description	Examples
GUI Applications	Creating graphical user interfaces.	JavaFX, Swing applications (e.g., desktop apps)
Web Applications	Building dynamic and interactive web applications.	Servlets, JSP, Spring Framework-based applications
Enterprise Applications	Structuring business logic and data processing.	Banking systems, CRM systems
Mobile Applications	Developing applications for mobile devices.	Android applications
Embedded Systems	Developing applications for embedded devices.	Java ME (Micro Edition) applications
Distributed Systems	Developing systems that run on multiple networked computers.	Java RMI, microservices architectures
Real-World Modeling	Modeling real-world entities as objects.	Simulation systems, games (e.g., flight simulators, RPG games)

Java Constructors

Java constructors are special types of methods that are used to initialize an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Typically, you will use a constructor to give initial values to the instance variables defined by the class or to perform any other start-up procedures required to create a fully formed object.

All classes have constructors, whether you define one or not because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your constructor, the default constructor is no longer used.

Rules for Creating Java Constructors

You must follow the below-given rules while creating Java constructors:

- The name of the constructors must be the same as the class name.
- Java constructors do not have a return type. Even do not use void as a return type.
- There can be multiple constructors in the same class, this concept is known as constructor overloading.
- The access modifiers can be used with the constructors, use if you want to change the visibility/accessibility of constructors.
- Java provides a default constructor that is invoked during the time of object creation. If you create any type of constructor, the default constructor (provided by Java) is not invoked.

Creating a Java Constructor

To create a constructor in Java, simply write the constructor's name (that is the same as the class name) followed by the brackets and then write the constructor's body inside the curly braces (**{}**).

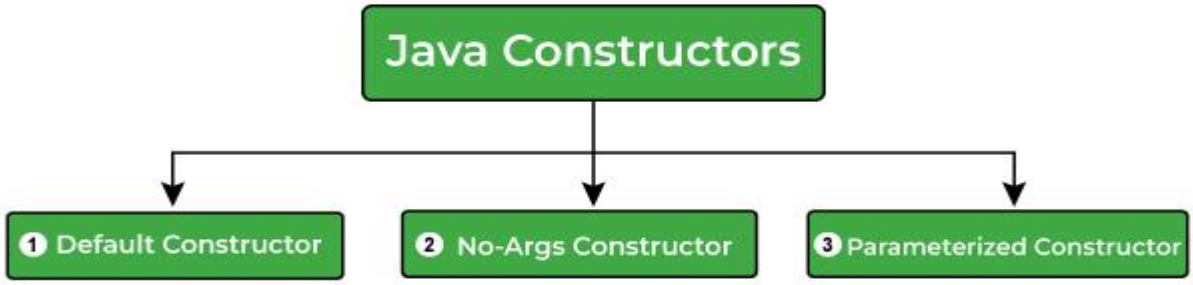
Syntax

```
class ClassName
 {
     ClassName()
     {
        Statements;
     }
}
```

Types of Java Constructors

There are three different types of constructors in Java, we have listed them as follows:

1. Default Constructor
2. No-Args Constructor
3. Parameterized Constructor



1. Default Constructor

If you do not create any constructor in the class, Java provides a default constructor that initializes the object.

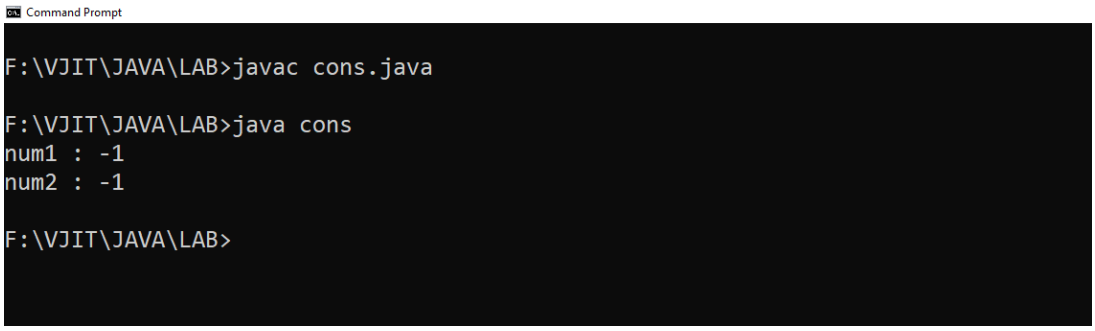
```
public class cons
{
  int num1;
  int num2;

  public static void main(String[] args)
  {
    // a default constructor will invoke here
    cons obj = new cons();
    // Printing the values
    System.out.println("num1 : " + obj.num1);
    System.out.println("num2 : " + obj.num2);
  }
}
```

```
Command Prompt

F:\VJIT\JAVA\LAB>javac cons.java

F:\VJIT\JAVA\LAB>java cons
num1 : 0
num2 : 0

F:\VJIT\JAVA\LAB>
```

2. No-Args (No Arguments) Constructor

As the name specifies, the No-argument constructor does not accept any arguments. By using the No-Args constructor you can initialize the class data members and perform various activities that you want on object creation.

```
public class cons
{
      int num1;
      int num2;
       cons()
       {
           num1 = -1;
           num2 = -1;
       }
      public static void main(String[] args)
      {
       // a default constructor will invoke here
       cons obj = new cons();

       // Printing the values
       System.out.println("num1 : " + obj.num1);
       System.out.println("num2 : " + obj.num2);
      }
 }
```

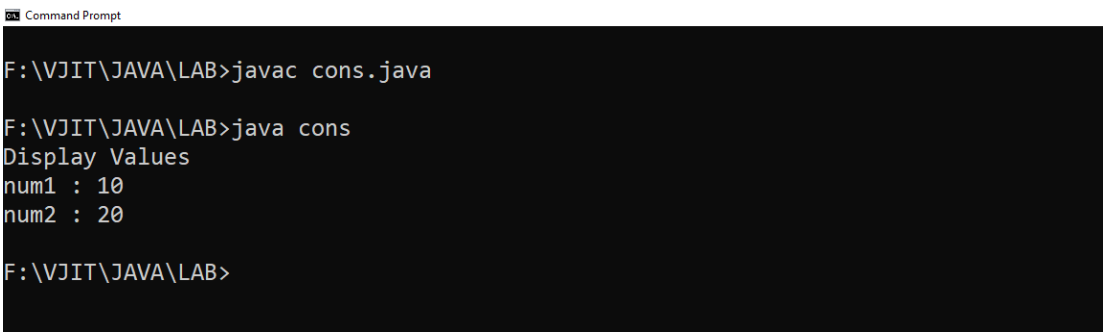


3. Parameterized Constructor

A constructor with one or more arguments is called a parameterized constructor.

Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

```
public class cons
 {
        int num1;
        int num2;
        // Creating parameterized constructor
        cons(int a, int b)
        {
          num1 = a;
          num2 = b;
        }
        public static void main(String[] args)
        {
          // Creating an object by passing the values
          // to initialize the attributes.
          // parameterized constructor will invoke
                cons obj = new cons(10, 20);
           // Printing the objects values
          System.out.println("Display Values");
          System.out.println("num1 : " + obj.num1);
          System.out.println("num2 : " + obj.num2);
        }
  }
```

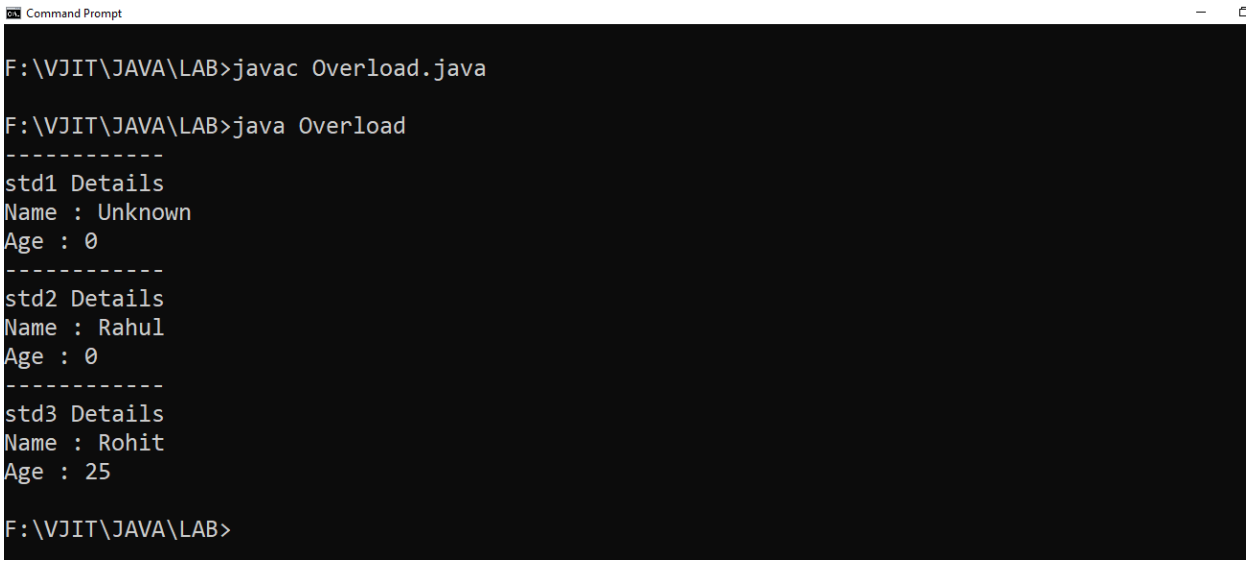


Constructor Overloading in Java

Constructor overloading means multiple constructors are defined with the same signature with different parameter list in a class. When you have multiple constructors with different parameters listed, then it will be known as constructor overloading.

```
// Example of Java Constructor Overloading
// Creating a Student Class
class Student
{
    String name;
    int age;
    // no-args constructor
    Student()
    {
          this.name = "Unknown";
          this.age = 0;
    }
    // parameterized constructor having one parameter
    Student(String name)
    {
             this.name = name;
             this.age = 0;
    }

    // parameterized constructor having both parameters
    Student(String name, int age)
    {
         this.name = name;
         this.age = age;
    }
    public void printDetails()
    {
      System.out.println("Name : " + this.name);
      System.out.println("Age : " + this.age);
    }
  }
```

```
public class Overload
 {
    public static void main(String[] args)
    {
      Student st1 = new Student(); // invokes no-args constructor
      Student st2 = new Student("Rahul"); // invokes parameterized constructor
    Student st3 = new Student("Rohit", 25); // invokes parameterized constructor

      // Printing details
        System.out.println("------------");
      System.out.println("std1 Details");
        st1.printDetails();
        System.out.println("------------");
      System.out.println("std2 Details");
        st2.printDetails();
        System.out.println("------------");
      System.out.println("std3 Details");
        st3.printDetails();
    }
}
```



Java Methods

Java Methods are blocks of code that perform a specific task. A method allows us to reuse code, improving both efficiency and organization. All **methods in Java** must belong to a **class**. Methods are similar to functions and expose the behavior of objects.

Creating a Java Method

Syntax

```
AccessModifier  returnType  nameOfMethod (Parameter List)
{
    // method body
}
```

The syntax shown above includes −

- **AccessModifier** − It defines the access type of the method and it is optional to use.
- **returnType** − Method may return a value.
- **nameOfMethod** − This is the method name. The method signature consists of the method name and the parameter list.
- **Parameter List** − The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- **method body** − The method body defines what the method does with the statements.

Example to Create a Java Method

```
/** the snippet returns the minimum between two numbers */
public static int minFunction(int n1, int n2)
 {
   int min;
   if (n1 > n2)
      min = n2;
   else
      min = n1;
   return min;
}
```

Calling a Java Method

For using a method, it should be called. There are two ways in which a method is called i.e., method returns a value or returning nothing (no return value).

Example: Defining and Calling a Java Method

```
public class ExampleMinNumber
 {
    public static void main(String[] args)
    {
    int a = 11;
    int b = 6;
     ExampleMinNumber obj=new ExampleMinNumber();
         int c=obj.min(a,b);
    System.out.println("Minimum Value = " + c);
  }
  /** returns the minimum of two numbers */
  public int min(int n1, int n2)
  {
    int min;
    if (n1 > n2)
      min = n2;
    else
      min = n1;
    return min;
  }
}
```

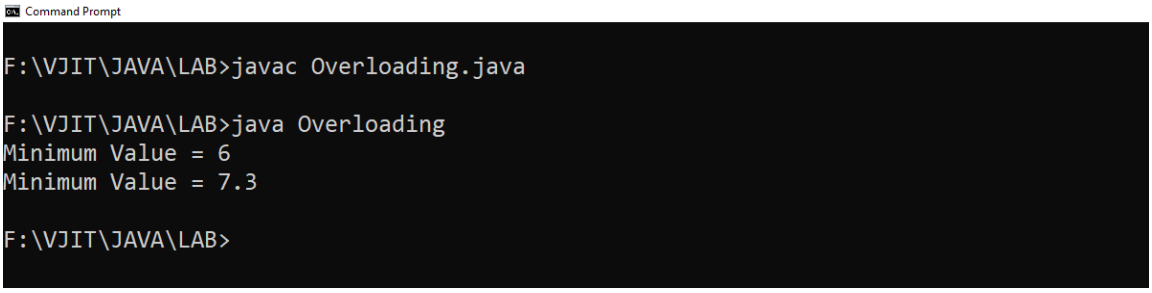
Output

```
Command Prompt

F:\VJIT\JAVA\LAB>javac ExampleMinNumber.java

F:\VJIT\JAVA\LAB>java ExampleMinNumber
Minimum Value = 6

F:\VJIT\JAVA\LAB>
```

Java Methods Overloading

When a class has two or more methods by the same name but different parameters, it is known as method overloading. It is different from overriding. In overriding, a method has the same method name, type, number of parameters, etc.

```
public class Overloading
{
  public static void main(String[] args)
  {
        int a = 11;
        int b = 6;
        double c = 7.3;
        double d = 9.4;
        int result1 = min(a, b);
        // same function name with different parameters
        double result2 = min(c, d);
        System.out.println("Minimum Value = " + result1);
        System.out.println("Minimum Value = " + result2);
  }
  // for integer
  public static int min(int n1, int n2)
  {
        int min;
        if (n1 > n2)
          min = n2;
        else
          min = n1;
        return min;
  }
    // for double
  public static double min(double n1, double n2)
  {
        double min;
        if (n1 > n2)
          min = n2;
        else
          min = n1;
        return min;
  }
}
```



Parameter Passing Techniques

Parameter passing in Java refers to the mechanism of transferring data between methods. Java supports two types of parameters passing techniques

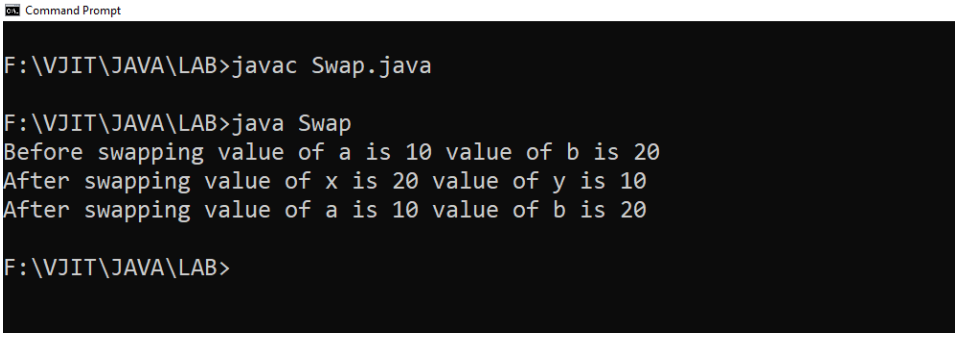
1. Pass-by-value (or) Call-by-value
2. Pass-by-reference (or) Call-by-reference.

1. Pass-by-value (or) Call-by-value

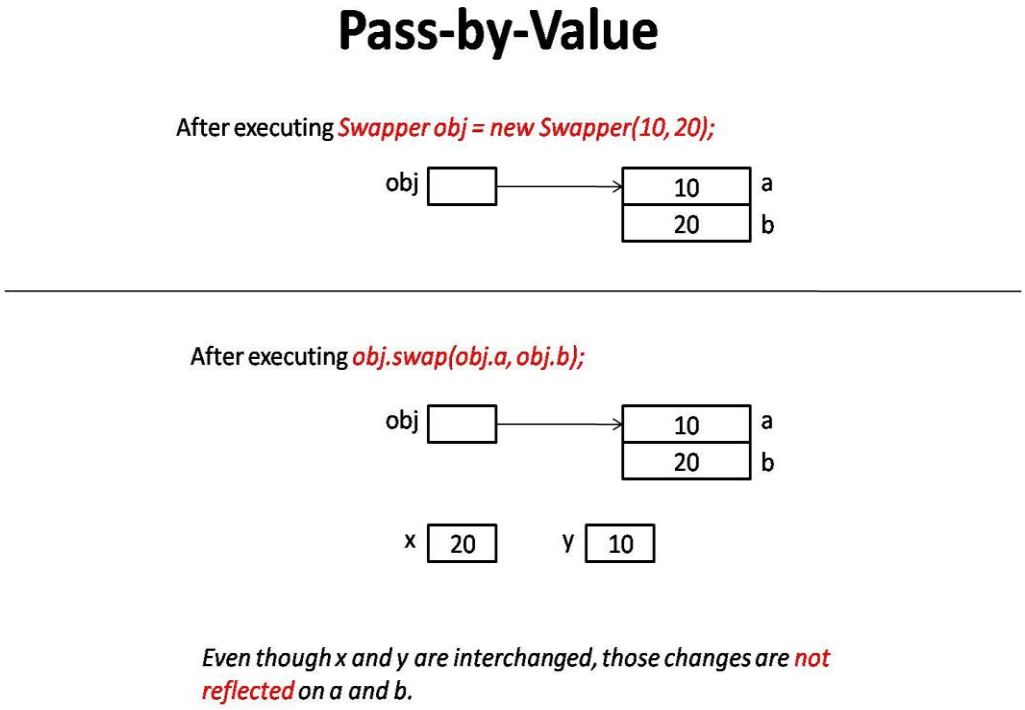
In *pass-by-value* technique, the actual parameters in the method call are *copied* to the dummy parameters in the method definition. So, whatever changes are performed on the dummy parameters, they are not reflected on the actual parameters as the changes you make are done to the copies and to the originals.

```
class Swapper
{
    int a;
    int b;
    Swapper(int x, int y)
    {
            a = x;
            b = y;
    }
    void swap(int x, int y)
    {
            int temp;
            temp = x;
            x = y;
            y = temp;
            System.out.println("After swapping value of x is "+x+" value of y is "+y);
    }
}
```

```
class Swap
{
    public static void main(String[] args)
    {
            Swapper obj = new Swapper(10, 20);
System.out.println("Before swapping value of a is "+obj.a+" value of b is "+obj.b);
            obj.swap(obj.a, obj.b);
System.out.println("After swapping value of a is "+obj.a+" value of b is "+obj.b);
    }
}
```



Although values of *x* and *y* are interchanged, those changes are not reflected on *a* and *b*. Memory representation of variables is shown in below figure:



2. **Pass-by-reference (or) Call-by-reference**

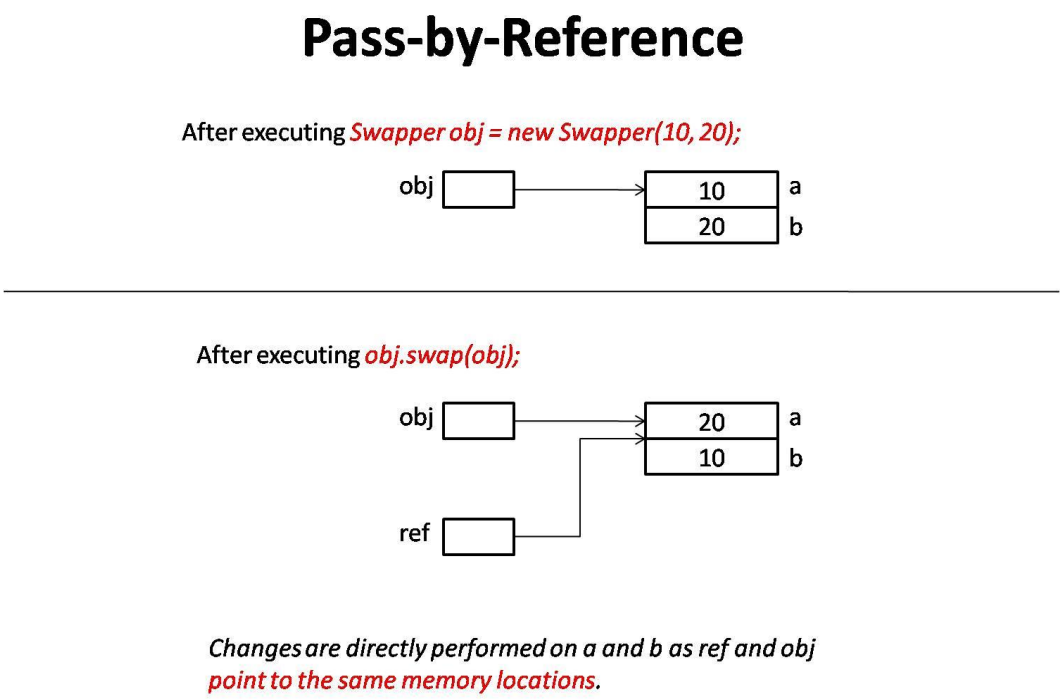
In *pass-by-reference* technique, reference (address) of the actual parameters are passed to the dummy parameters in the method definition. So, whatever changes are performed on the dummy parameters, they are reflected on the actual parameters too as both references point to same memory locations containing the original values.

```
class Swapper
{
	int a;
	int b;
	Swapper(int x, int y)
	{
		a = x;
		b = y;
	}
	void swap(Swapper ref)
	{
		int temp;
		temp = ref.a;
		ref.a = ref.b;
		ref.b = temp;
	}
}
class Swap
{
	public static void main(String[] args)
	{
		Swapper obj = new Swapper(10, 20);
System.out.println("Before swapping value of a is "+obj.a+" value of b is "+obj.b);
		obj.swap(obj);
System.out.println("After swapping value of a is "+obj.a+" value of b is "+obj.b);
	}
}
```

Command Prompt

```
F:\VJIT\JAVA\LAB>javac Swap.java

F:\VJIT\JAVA\LAB>java Swap
Before swapping value of a is 10 value of b is 20
After swapping value of a is 20 value of b is 10

F:\VJIT\JAVA\LAB>
```

The changes performed inside the method *swap* are reflected on *a* and *b* as we have passed the reference *obj* into *ref* which also points to the same memory locations as *obj*. Memory representation of variables is shown in below figure:



Note: In Java, parameters of *primitive types* are passed by value which is same as *pass-by-value* and parameters of *reference types* are also passed by value (the reference is copied) which is same as *pass-by-reference*. So, in Java all parameters are passed by value only.

Access Control in Java

Access control in Java is a fundamental concept that determines what parts of a program can be accessed and manipulated by which parts of the program. Java provides access control mechanisms to help developers create secure and maintainable software.

Access Control in Java refers to the mechanism used to restrict or allow access to certain parts of a Java program, such as classes, methods, and variables. Java's access control mechanism promotes code encapsulation, and information hiding, and reduces the likelihood of errors and security vulnerabilities in the program. It is implemented by using access control modifiers, which are keywords placed before the declaration of the class member.

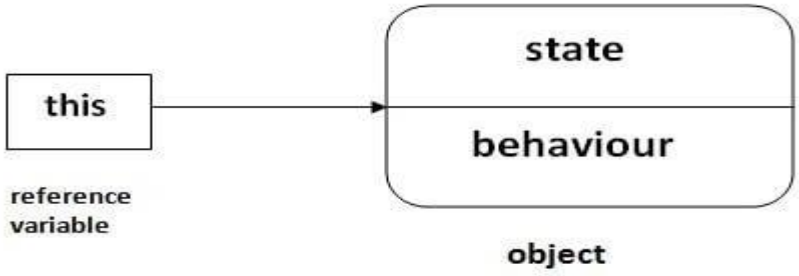
There are **four** access control modifiers in Java

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

Access Modifier	within class	within package	outside package by subclass only	outside package
Public	Y	Y	Y	Y
Private	Y	N	N	N
Protected	Y	Y	Y	N
Default	Y	Y	N	N

this Keyword

In Java, 'this' is a reference variable that refers to the current class object. It can be used to call current class methods and fields, to pass an instance of the current class as a parameter, and to differentiate between the local and instance variables. Using "this" reference can improve code readability and reduce naming conflicts.

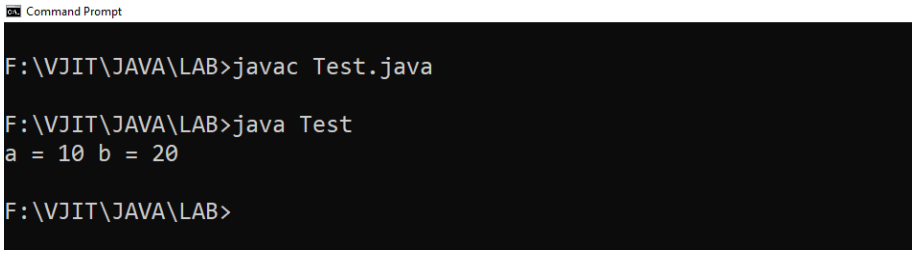


Usage of **'this'** keyword

1. Using the 'this' keyword to refer to current class instance variables.
2. Using this() to invoke the current class constructor
3. Using 'this' keyword to return the current class instance
4. Using 'this' keyword as the method parameter
5. Using 'this' keyword to invoke the current class method
6. Using 'this' keyword as an argument in the constructor call

1. Using 'this' keyword to refer to current class instance variables

```
// using 'this' keyword to refer current class instance variables
class Test
{
  int a;
  int b;
  // Parameterized constructor
  Test(int a, int b)
  {
        this.a = a;
        this.b = b;
  }

  void display()
  {
        // Displaying value of variables a and b
        System.out.println("a = " + a + " b = " + b);
  }

  public static void main(String[] args)
  {
        Test object = new Test(10, 20);
        object.display();
  }
}
```



2. Using this() to invoke current class constructor

```
// Java code for using this() to
// invoke current class constructor
class Test
{
    int a;
    int b;
    // Default constructor
    Test()
    {
            this(10, 20);
            System.out.println("Inside default constructor \n");
    }
    // Parameterized constructor
    Test(int a, int b)
    {
            this.a = a;
            this.b = b;
            System.out.println("Inside parameterized constructor");
    }
    public static void main(String[] args)
    {
            Test object = new Test();
    }
}
```

3. Using 'this' keyword to return the current class instance

```
// Java code for using 'this' keyword
// to return the current class instance
class Test
{
    int a;
    int b;
    // Default constructor
    Test()
    {
            a = 10;
            b = 20;
    }
    // Method that returns current class instance
    Test get()
    {
    return this;
    }
    // Displaying value of variables a and b
    void display()
    {
            System.out.println("a = " + a + " b = " + b);
    }
    public static void main(String[] args)
    {
            Test object = new Test();
            object.get().display();
    }
}
```

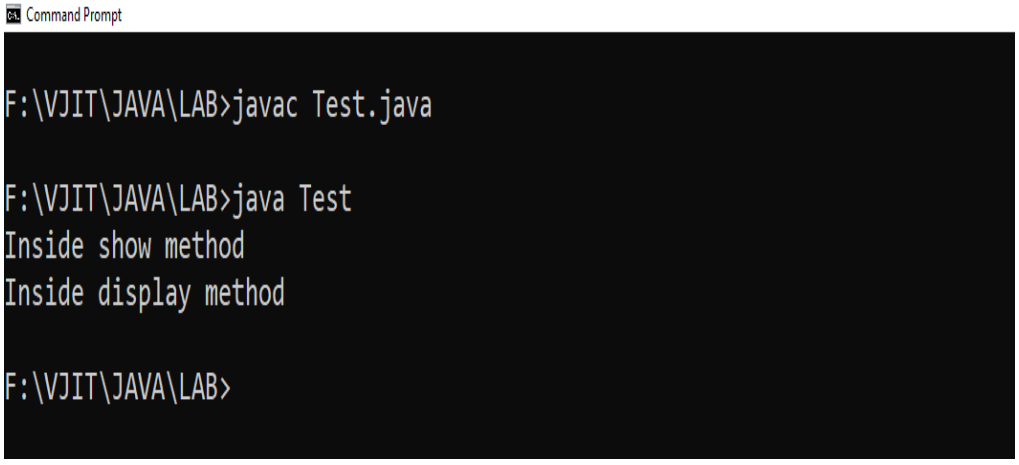
```
Command Prompt

F:\VJIT\JAVA\LAB>javac Test.java

F:\VJIT\JAVA\LAB>java Test
a = 10 b = 20

F:\VJIT\JAVA\LAB>
```

4. Using 'this' keyword as a method parameter

```
// Java code for using 'this'
// keyword as method parameter
class Test
{
   int a;
   int b;

   // Default constructor
   Test()
   {
         a = 10;
         b = 20;
   }
   // Method that receives 'this' keyword as parameter
   void display(Test obj)
   {
         System.out.println("a = " + obj.a+ " b = " + obj.b);
   }

   // Method that returns current class instance
   void get()
   {
     display(this);
   }
   // main function
   public static void main(String[] args)
   {
         Test object = new Test();
         object.get();
   }
}
```

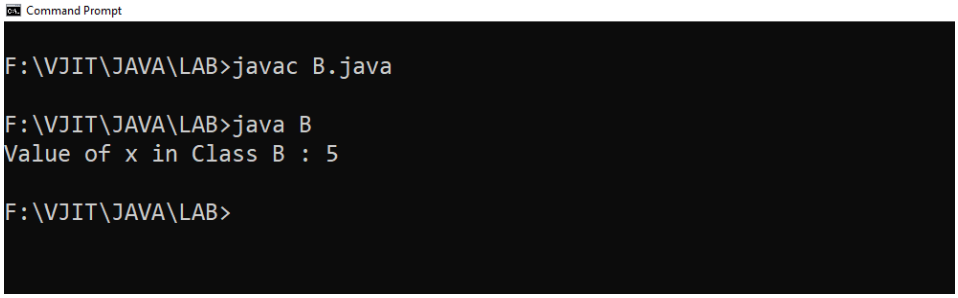
5. Using 'this' keyword to invoke the current class method

```
// Java code for using this to invoke current class method
class Test
{
  void display()
  {
        // calling function show()
        this.show();
        System.out.println("Inside display method");
  }
  void show()
  {
        System.out.println("Inside show method");
  }
  public static void main(String args[])
  {
        Test t1 = new Test();
        t1.display();
  }
}
```



6. Using 'this' keyword as an argument in the constructor call

```
// Java code for using this as an argument in constructor call
// Class with object of Class B as its data member
class A
{
        B obj;
        // Parameterized constructor with object of B as a parameter
        A(B obj)
        {
                this.obj = obj;
                // calling display method of class B
                obj.display();
        }
}
class B
{
        int x = 5;
        // Default Constructor that create an object of A
        // with passing this as an argument in the constructor
        B()
        {
          A obj = new A(this);
        }
        // method to show value of x
        void display()
        {
                System.out.println("Value of x in Class B : " + x);
        }
        public static void main(String[] args)
        {
                B obj = new B();
        }
}
```



static Keyword

The **static keyword** in Java is used for memory management. The static keyword belongs to the class rather than an instance of the class.

The static can be used as:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class

Characteristics of Static Keyword

1. **Belongs to the class**: When a member is declared as **static**, it is associated with the class rather than with instances of the class.
2. **Accessed without creating an instance**: Since **static** members are associated with the class itself, they can be accessed using the class name, without needing to create an instance of the class. For example, **ClassName.staticMethod()**.
3. **Shared among all instances**: Since there is only one copy of a **static** member per class, it is shared among all instances of the class. This can be useful for maintaining common data or behavior across all instances.
4. **Can access other static members directly**: **static** members can directly access other **static** members of the same class, but they cannot directly access non-static (instance) members. This is because **static** members exist independently of any particular instance.
5. **Initialization**: **static** variables are initialized only once, at the start of the execution, before the initialization of any instance variables. They are initialized in the order they are declared.
6. **Used for utility methods and constants**: **static** methods are commonly used for utility methods that perform a task related to the class, but do not require any instance-specific data. **static** variables are often used for constants that are shared among all instances of the class.

- **Static variables**

When a variable is declared as static, then a single copy of the variable is created and shared among all objects at the class level. Static variables are, essentially, global variables. All instances of the class share the same static variable.

Important points for static variables:

- We can create static variables at the class level only.
- static block and static variables are executed in the order they are present in a program.

- **Static methods**

When a method is declared with the *static* keyword, it is known as the static method. The most common example of a static method is the *main( )* method. As discussed above, any static member can be accessed before any objects of its class are created, and without reference to any object. Methods declared as static have several restrictions:

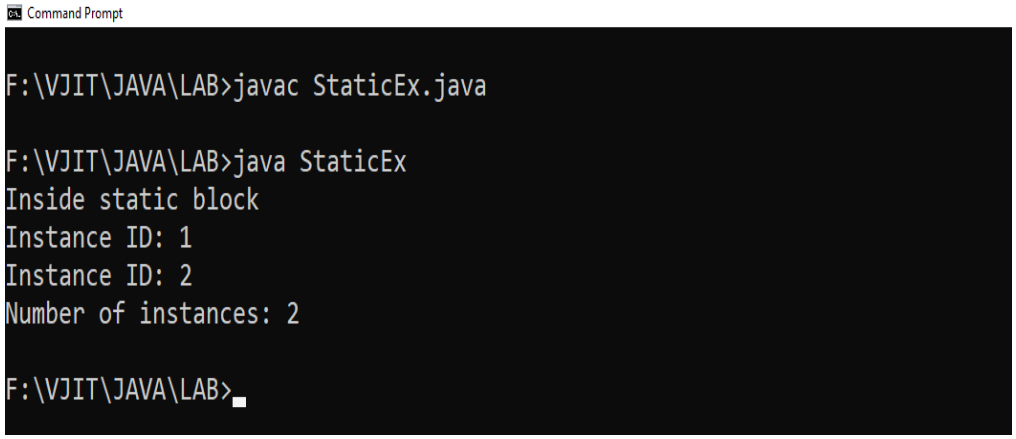
- They can only directly call other static methods.
- They can only directly access static data.
- They cannot refer to this or super in any way.

- **Static blocks**

If you need to do the computation in order to initialize your **static variables**, you can declare a static block that gets executed exactly once, when the class is first loaded.

```
public class StaticEx
{
   public static int count = 0;
   public int id;
 // static block
   static
   {
      System.out.println("Inside static block");
    }
   public StaticEx()
  {
      count++;
      id = count;
   }
```

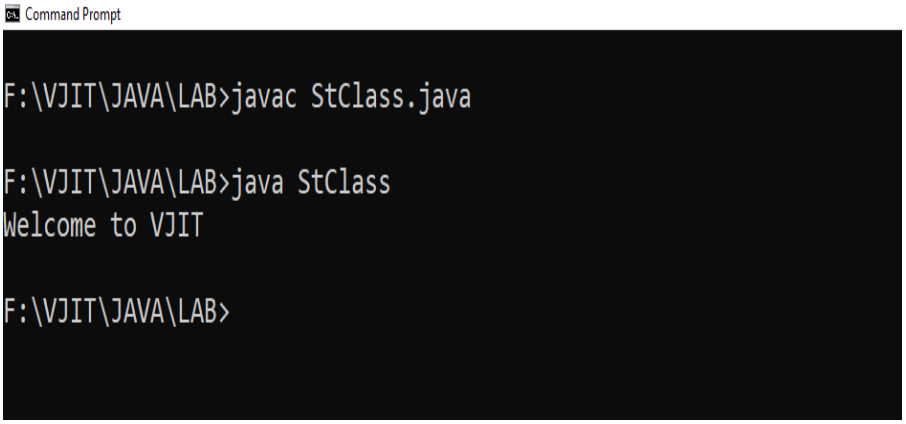
```
    public static void printCount()
    {
        System.out.println("Number of instances: " + count);
    }

    public void printId()
    {
        System.out.println("Instance ID: " + id);
    }

    public static void main(String[] args)
    {
        StaticEx s1 = new StaticEx();
        StaticEx s2 = new StaticEx();
        s1.printId();
        s2.printId();
        StaticEx.printCount();
    }
}
```



- **Static Classes**

A class can be made **static** only if it is a nested class. We cannot declare a top-level class with a static modifier but can declare nested classes as static. Such types of classes are called Nested static classes. Nested static class doesn't need a reference of Outer class. In this case, a static class cannot access non-static members of the Outer class.

```
// A java program to demonstrate use of static keyword with Classes
import java.io.*;
public class StClass
{
   private static String str = "Welcome to VJIT";

        // Static class
        static class MyNestedClass
         {
                  // non-static method
                 public void disp()
                 {
                  System.out.println(str);
                 }
        }
     public static void main(String args[])
   {
      StClass.MyNestedClass obj = new StClass.MyNestedClass();
      obj.disp();
   }
}
```



nested and inner classes

Nested Classes in Java

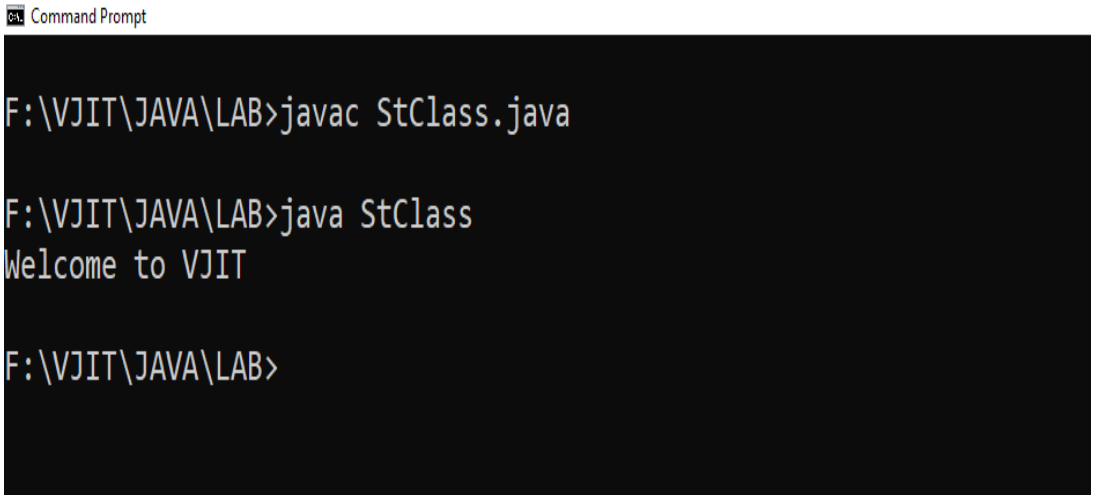
In Java, it is possible to define a class within another class, such classes are known as *nested* classes.

- The scope of a nested class is bounded by the scope of its enclosing class.
- A nested class has access to the members, including private members, of the class in which it is nested. But the enclosing class does not have access to the member of the nested class.
- A nested class is also a member of its enclosing class.
- As a member of its enclosing class, a nested class can be declared *private*, *public*, *protected*, or *package-private* (default).
- Nested classes are divided into two categories:
    1. **static nested class:** Nested classes that are declared *static* are called static nested classes.
    2. **inner class:** An inner class is a non-static nested class.

Syntax:

```
class OuterClass
{
    ...
    class NestedClass
    {
        ...
    }
}
```

```
// A java program to demonstrate use of static keyword with Classes
import java.io.*;
public class StClass
{
   private static String str = "Welcome to VJIT";

        // Static class
        static class MyNestedClass
         {
                  // non-static method
                 public void disp()
                 {
                  System.out.println(str);
                 }
        }
     public static void main(String args[])
   {
      StClass.MyNestedClass obj = new StClass.MyNestedClass();
      obj.disp();
   }
}
```

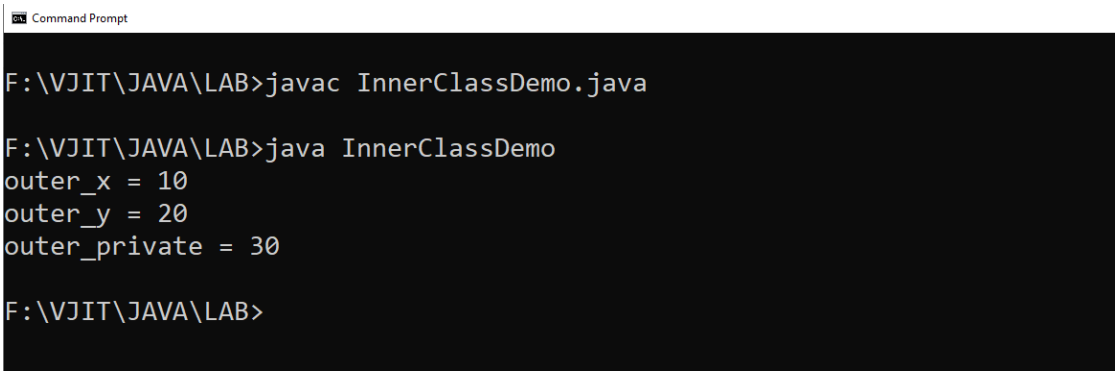


Inner classes

To instantiate an inner class, you must first instantiate the outer class. **Syntax:**

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

```
// accessing inner class & outer class

class OuterClass
{
    // static member
    static int outer_x = 10;
    // instance(non-static) member
    int outer_y = 20;
    // private member
    private int outer_private = 30;
    // inner class
    class InnerClass
    {
            void display()
            {
                    // can access static member of outer class
                    System.out.println("outer_x = " + outer_x);
                    // can also access non-static member of outer class
                    System.out.println("outer_y = " + outer_y);
                    // can also access a private member of the outer class
                    System.out.println("outer_private = "+ outer_private);
            }
    }
}
public class InnerClassDemo
{
    public static void main(String[] args)
    {
            // accessing an inner class
            OuterClass outerObject = new OuterClass();
            OuterClass.InnerClass innerObject = outerObject.new InnerClass();
            innerObject.display();
    }
}
```

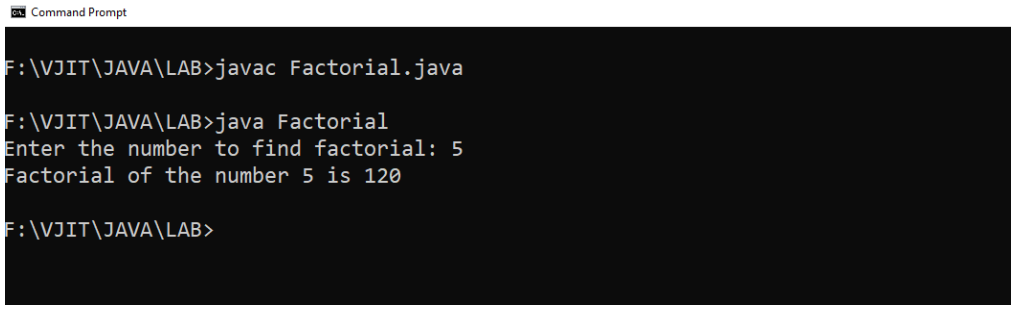


Recursion in Java

Recursion in Java is a process in which a method calls itself continuously. A method that calls itself is called a **recursive** method. A few Java recursion examples are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

Program to find Factorial of a Number

```
import java.util.Scanner;
public class Factorial
{
    public static int fact(int n)
    {
            if (n != 0) // ending condition
                    return n * fact(n - 1); // recursive call
            else
                    return 1;
    }
    public static void main(String[] args)
    {
      Scanner sc = new Scanner(System.in);
      // Prompt user for the number to find factorial
      System.out.print("Enter the number to find factorial: ");
      int num = sc.nextInt();
       int fact=fact(num);
      System.out.println("Factorial of the number "+num+" is "+fact);
    }
}
```



Garbage Collection in Java

Garbage Collection in Java is an automatic memory management process that helps Java programs run efficiently. Java programs compile to bytecode that can be run on a Java Virtual Machine (JVM). When Java programs run on the JVM, objects in the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

The garbage collector automatically finds and removes objects that are no longer needed, freeing up memory in the heap. It runs in the background as a daemon thread, helping to manage memory efficiently without requiring the programmer's constant attention.

Ways to Request JVM to Run Garbage Collection

- Once an object is eligible for garbage collection, it may not be destroyed immediately. The garbage collector runs at the JVM's discretion, and you cannot predict when it will occur.
- We can also request JVM to run Garbage Collector. There are two ways to do it
    - Using **System.gc()**: This static method requests the JVM to perform garbage collection.
    - Using **Runtime.getRuntime().gc()**: This method also requests garbage collection through the Runtime class.

Strings

A **String** in Java is a sequence of characters that can be used to store and manipulate text data and it is basically an array of characters that are stored in a sequence of memory locations.

All the strings in Java are immutable in nature, i.e. once the string is created we can't change it.

```
1. String s="VJIT";
2. char[] ch={'H','y','d','e','r','a','b','a','d'};
   String s=new String(ch);
```

In Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

Create a String object?

There are two ways to create String object:

1. By String literal
2. By new keyword

1) String Literal

Java String literal is created by using double quotes.

String s="welcome";

2) By new keyword

*String s=**new** String("Welcome");*

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

S.No.	Method	Description
1.	int length()	It returns string length
2.	char charAt(int index)	It returns char value for the particular index
3.	String substring(int beginIndex)	It returns substring for given begin index.
4.	String substring(int beginIndex, int endIndex)	It returns substring for given begin index and end index.
5.	boolean equals(Object another)	It checks the equality of string with the given object.
6.	boolean isEmpty()	It checks if string is empty.
7.	String concat(String str)	It concatenates the specified string.
8.	String replace(char old, char new)	It replaces all occurrences of the specified char value.
9.	String toLowerCase()	It returns a string in lowercase.
10.	String toUpperCase()	It returns a string in uppercase.
11.	String trim()	It removes beginning and ending spaces of this string.

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

       // create a string
          String s = "Hello! World";
          System.out.println("String: " + s);

       // get the length of string
          int len = s.length();
          System.out.println("Length: " + len);

      //display char at position
         String myStr = "VJIT";
          char result = myStr.charAt(2);
          System.out.println(result);

      // create first string
          String first = "Java ";
          System.out.println("First String: " + first);
          // create second string
          String second = "Programming";
          System.out.println("Second String: " + second);
          // join two strings
          String joined = first.concat(second);
          System.out.println("Joined String: " + joined);

         //substring
         String st = "Hello, World!";
          System.out.println(st.substring(7, 12));

         // create 3 strings
          String s1 = "java programming";
          String s2 = "java programming";
          String s3 = "python programming";
```

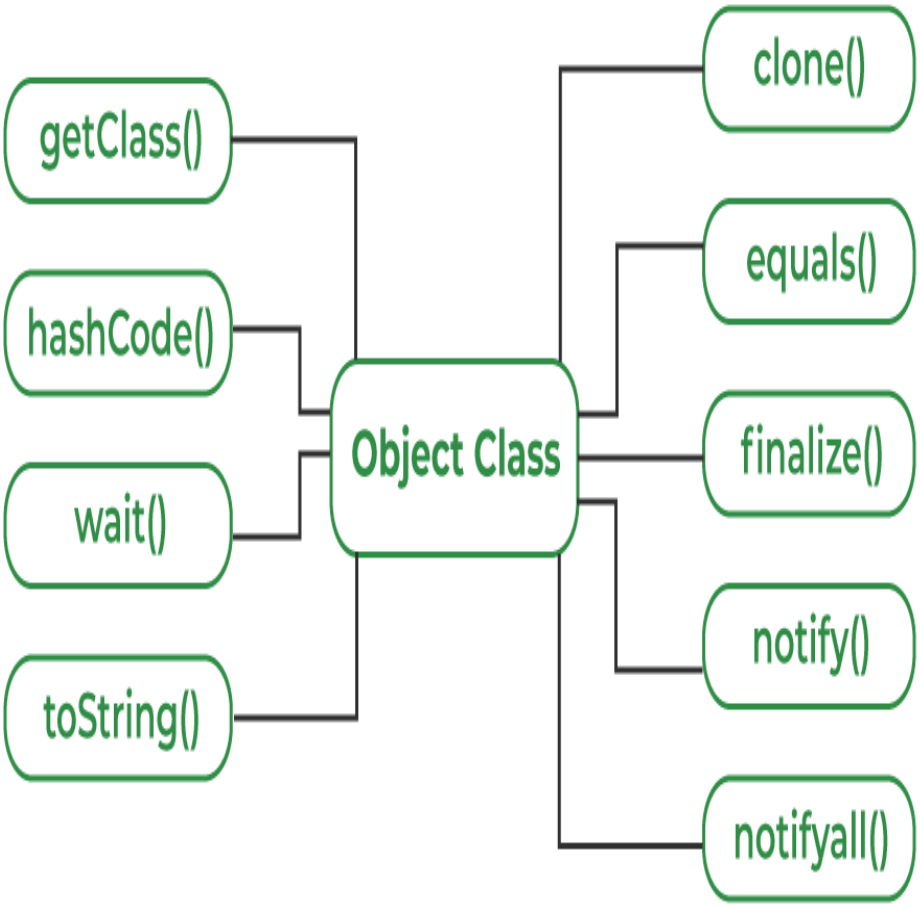
```
        // compare first and second strings
          boolean result1 = s1.equals(s2);
          System.out.println("Strings first and second are equal: " + result1);
          // compare first and third strings
          boolean result2 = s1.equals(s3);
          System.out.println("Strings first and third are not equal: " + result2);

       //Replace
         String st1 = "Hello";
          System.out.println(st1.replace('l', 'p'));
         String txt = "Hello World";

         //convert upper or lower case
          System.out.println(txt.toUpperCase());
          System.out.println(txt.toLowerCase());

       //trim
          String str = "      Java World!      ";
          System.out.println(str);
          System.out.println(str.trim());
         }
}
```

```
Command Prompt

F:\VJIT\JAVA\LAB>javac Strings.java

F:\VJIT\JAVA\LAB>java Strings
String: Hello! World
Length: 12
I
First String: Java
Second String: Programming
Joined String: Java Programming
World
Strings first and second are equal: true
Strings first and third are not equal: false
Heppo
HELLO WORLD
hello world
      Java World!
Java World!

F:\VJIT\JAVA\LAB>
```

Object class

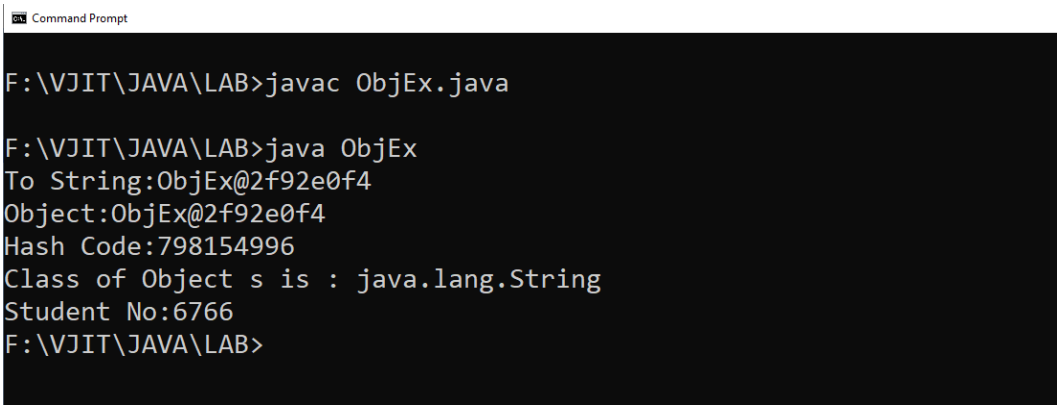
Object class is the base class for all classes in Java. The Object class resides within the **java.lang** package. It serves as a foundation for all classes, directly or indirectly. If a class doesn't extend any other class, it's a direct child of Object; if it extends another class, it's indirectly derived. Consequently, all Java classes inherit the methods of the Object class.

Object Class Methods



S.No.	Method Name	Description
1.	**String toString()**	This method returns a string representation of the object.
2.	**int hashCode()**	This method returns a hash code value for the object.
3.	**boolean equals(Object obj)**	This method indicates whether some other object is "equal to" this one.
4.	**Class<?> getClass()**	This method returns the runtime class of this Object.
5.	**protected Object clone()**	This method creates and returns a copy of this object.
6.	**protected void finalize()**	This method is called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
7.	**void wait()**	This method causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
8.	**void notify()**	This method wakes up a single thread that is waiting on this object's monitor.
9.	**void notifyAll()**	This method wakes up all threads that are waiting on this object's monitor.

```
import java.util.*;

public class ObjEx implements Cloneable
{
   int stno;

   ObjEx(int stno)
   {
      this.stno=stno;
   }

    public static void main(String args[])
   {
      ObjEx obj = new ObjEx(6766);

      // Below two statements are equivalent
      System.out.println("To String:"+obj.toString());
      System.out.println("Object:"+obj);
            //hashcode
       System.out.println("Hash Code:"+obj.hashCode());
            //getclass
       Object s = new String("Hi");
       Class c = s.getClass();
      //for the String
      System.out.println("Class of Object s is : " + c.getName());
           //clone
         try
         {
           ObjEx obj2 = (ObjEx)obj.clone();
           System.out.print("Student No:"+obj2.stno);
         }
         catch (Exception e)
         {
       System.out.println(e);
      }
   }
}
```


Command Prompt
F:\VJIT\JAVA\LAB>javac ObjEx.java
F:\VJIT\JAVA\LAB>java ObjEx
To String:ObjEx@2f92e0f4
Object:ObjEx@2f92e0f4
Hash Code:798154996
Class of Object s is : java.lang.String
Student No:6766
F:\VJIT\JAVA\LAB>