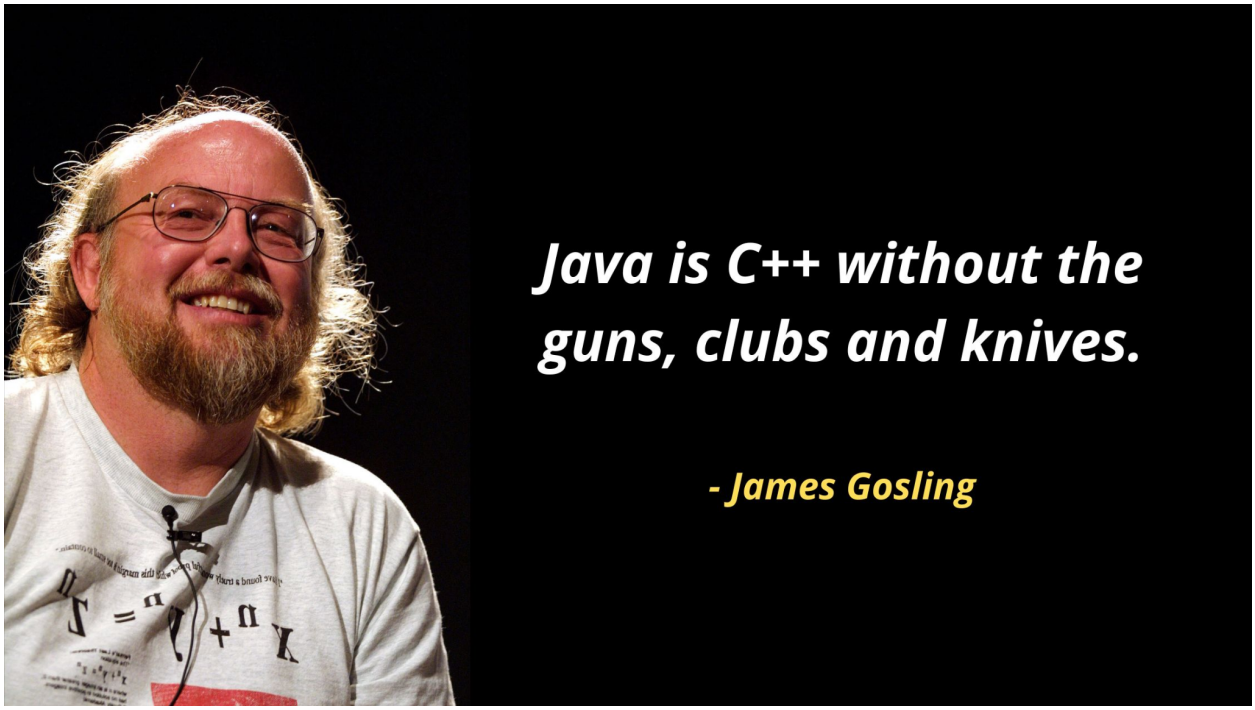
+NOTES

JAVA

A PROGRAMMING LANGUAGE BY JAMES GOSLING



Introduction

Java is a **programming language and a platform**. Java is a high level, robust, object-oriented and secure programming language.

Java was developed by ***Sun Microsystems*** (which is now the subsidiary of Oracle) in the year **1995**. *James Gosling* is known as the father of Java. Before Java, its name was ***Oak***. Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

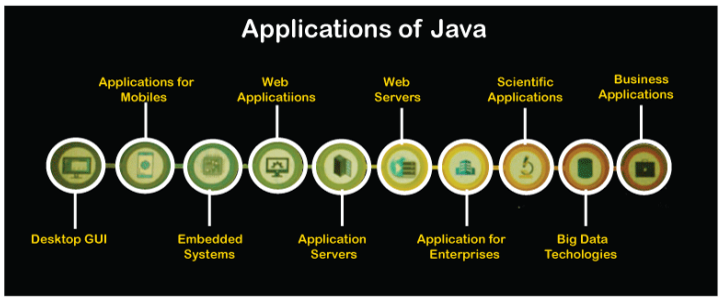
Why Use Java?

- Java **works on different platforms** (Windows, Mac, Linux, Raspberry Pi, etc.)
- It is one of the most **popular programming language** in the world
- It is **easy to learn** and **simple to use**
- It is **open-source and free**
- It is **secure, fast and powerful**
- It has a **huge community support** (tens of millions of developers)
- Java is an **object oriented language** which gives a clear structure to programs and allows code to be reused, lowering development costs

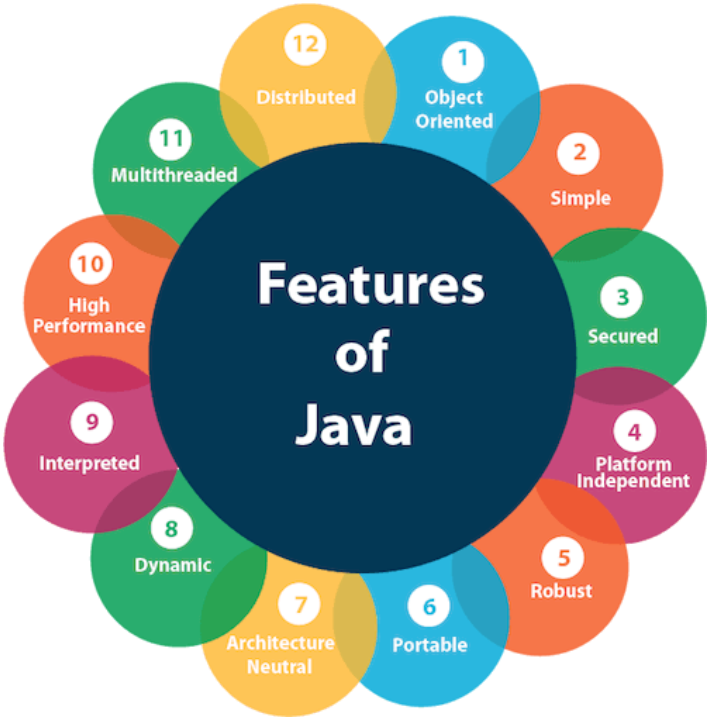
Application

According to Sun, 3 billion devices run Java. There are many devices where Java is currently used. Some of them are as follows:

- ➔ Desktop Applications such as acrobat reader, media player, antivirus, etc.
- ➔ Web Applications such as irctc.co.in, javatpoint.com, etc.
- ➔ Enterprise Applications such as banking applications.
- ➔ Mobile
- ➔ Embedded System
- ➔ Smart Card
- ➔ Robotics
- ➔ Games, etc.



Features of Java



1. Platform Independent: Compiler converts **source code to bytecode** and then the **JVM executes the bytecode** generated by the compiler. This bytecode can run on any platform be it Windows, Linux, or macOS which means if we compile a program on Windows, then we can run it on Linux and vice versa. Each operating system has a different JVM, but the **output produced by all the OS is the same** after the execution of bytecode. That is why we call java a **platform-independent language.**

2. Object-Oriented Programming Language: Organizing the program in the terms of **collection of objects** is a way of object-oriented programming, each of which represents an instance of the class.

The four main **concepts of Object-Oriented programming** are:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

3. Simple: Java is one of the simple languages as it **does not have** complex features like **pointers, operator overloading, multiple inheritances, and Explicit memory allocation**.

4. Robust: Java language is robust which means **reliable.** It is developed in such a way that it puts a lot of effort into checking errors as early as possible, that is why the java compiler is able to detect even those errors that are not easy to detect by another programming language. The main features of java that make it robust are **garbage collection, Exception Handling, and memory allocation.**

5. Secure: In java, we don't have pointers, so we cannot access out-of-bound arrays i.e it shows **ArrayIndexOutOfBound Exception** if we try to do so. That's why several security flaws like **stack corruption or buffer overflow** are impossible to exploit in Java. Also java programs run in an environment that is **independent of the os**(operating system) environment which makes java programs more secure .

6. Distributed: We can create distributed applications using the java programming language. **Remote Method Invocation** and **Enterprise Java Beans** are used for creating distributed applications in java. The java programs can be easily distributed on one or more systems that are connected to each other through an internet connection.

7. Multithreading: Java supports multithreading. It is a Java feature that allows **concurrent execution of two or more parts of a program** for maximum utilization of the CPU.

8. Portable: As we know, java code written on one machine can be **run on another machine.** The platform-independent feature of java in which its platform-independent bytecode can be taken to any platform for execution makes java portable.

9. High Performance: Java architecture is defined in such a way that it **reduces overhead** during the runtime and at some time java uses **Just In Time (JIT) compiler** where the compiler compiles code on-demand basics where it only compiles those methods that are called making applications to execute faster.

10. Dynamic flexibility: Java being completely object-oriented gives us the **flexibility to add classes, new methods to existing classes and even create new classes through**

sub-classes. Java even supports functions written in other languages such as C, C++ which are referred to as native methods.

11. Sandbox Execution: Java programs run in a **separate space** that allows user to execute their applications without affecting the underlying system with help of a bytecode verifier. Bytecode verifier also provides additional security as its role is to check the code for any violation of access.

12. Write Once Run Anywhere: As discussed above java application generates a '.class' file which corresponds to our applications(program) but contains code in binary format. It provides ease t architecture-neutral ease as bytecode is not dependent on any machine architecture. It is the primary reason java is used in the enterprising IT industry globally worldwide.

13. Power of compilation and interpretation: Most languages are designed with purpose either they are compiled language or they are interpreted language. But java integrates arising enormous power as Java compiler compiles the source code to bytecode and JVM executes this bytecode to machine OS-dependent executable code.

History of Java:

The history of Java is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time. The history of Java starts with the Green Team. Java team members (also known as Green Team), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was best suited for internet programming. Later, Java technology was incorporated by Netscape.

The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic". Java was developed by James Gosling, who is known as the father of Java, in 1995. James Gosling and his team members started the project in the early '90s.

Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc. Following are given significant points that describe the history of Java.

1) **James Gosling, Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.

2) Initially it was designed for small, embedded systems in electronic appliances like set-top boxes.

3) Firstly, it was called **"Greentalk"** by James Gosling, and the file extension was .gt.

4) After that, it was called **Oak** and was developed as a part of the Green project.

Why Java was named as “OAK”?

Why Oak? Oak is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc. In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

Why Java programming named as “JAVA”?

Why had they chose the name Java for Java language? The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA", etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell, and fun to say.

According to James Gosling, "Java was one of the top choices along with **Silk**". Since Java was so unique, most of the team members preferred Java than other names.

8) Java is an island in Indonesia where the first coffee was produced (called Java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having a cup of coffee nearby his office.

9) Notice that Java is just a name, not an acronym.

10) Initially developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.

11) In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.

12) JDK 1.0 was released on January 23, 1996. After the first release of Java, there have been many additional features added to the language. Now Java is being used in Windows applications, Web applications, enterprise applications, mobile applications, cards, etc. Each new version adds new features in Java.

Java versions History:

C++ Vs Java:

Comparison Index	C++	Java

Platform-independent	C++ is platform-dependent.	Java is platform-independent.
Mainly used for	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in Windows-based, web-based, enterprise, and mobile applications.
Design Goal	C++ was designed for systems and applications programming. It was an extension of the C programming language.	Java was designed and created as an interpreter for printing systems but later extended as a support network computing. It was designed to be easy to use and accessible to a broader audience.
Goto	C++ supports the goto statement.	Java doesn't support the goto statement.
Multiple inheritance	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by using interfaces in java.
Operator Overloading	C++ supports operator overloading.	Java doesn't support operator overloading.
Pointers	C++ supports pointers. You can write a pointer program in C++.	Java supports pointer internally. However, you can't write the pointer program in java. It means java has restricted pointer support in java.
Compiler and Interpreter	C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code	Java uses both compiler and interpreter. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform-independent.

	so, C++ is platform dependent.	
Call by Value and Call by reference	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.
Structure and Union	C++ supports structures and unions.	Java doesn't support structures and unions.
Thread Support	C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support.	Java has built-in thread support.
Documentation comment	C++ doesn't support documentation comments.	Java supports documentation comment (/** ... */) to create documentation for java source code.
Virtual Keyword	C++ supports virtual keyword so that we can decide whether or not to override a function.	Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default.
unsigned right shift >>>	C++ doesn't support >>> operator.	Java supports unsigned right shift >>> operator that fills zero at the top for the negative numbers. For positive numbers, it works same like >> operator.
Inheritance Tree	C++ always creates a new inheritance tree.	Java always uses a single inheritance tree because all classes are the child of the Object class in Java. The Object class is the root of the inheritance tree in java.

Hardware	C++ is nearer to hardware.	Java is not so interactive with hardware.
Object-oriented	C++ is an object-oriented language. However, in the C language, a single root hierarchy is not possible.	Java is also an object-oriented language. However, everything (except fundamental types) is an object in Java. It is a single root hierarchy as everything gets derived from java.lang.Object.

What is an IDE?

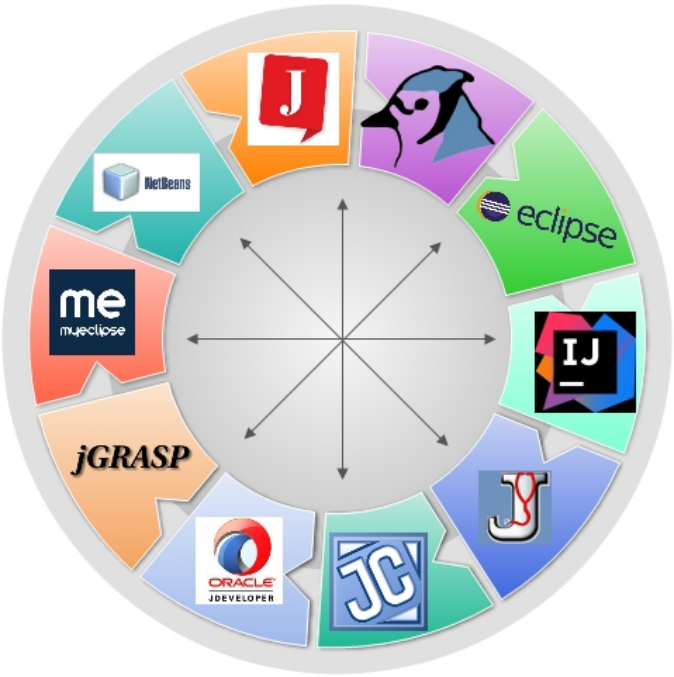
An integrated development environment (IDE) integrates popular developer tools into a single graphical user interface for developing applications (GUI).

An IDE typically consists of the following components:

Source code editor: A text editor can help write software code by offering features such as syntax highlighting with visual prompts, language-specific auto-completion, and bug testing as code is written.

Local build automation: Utilities that automate quick, repeatable tasks such as compiling computer source code into binary code, packaging binary code, and running automated tests to create a local build of the software for use by the developer.

Debugger: A program that can graphically represent the position of a bug in the original code and is used to evaluate other programs.



VARIOUS IDE'S FOR JAVA

JAVA ARCHITECTURE

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:

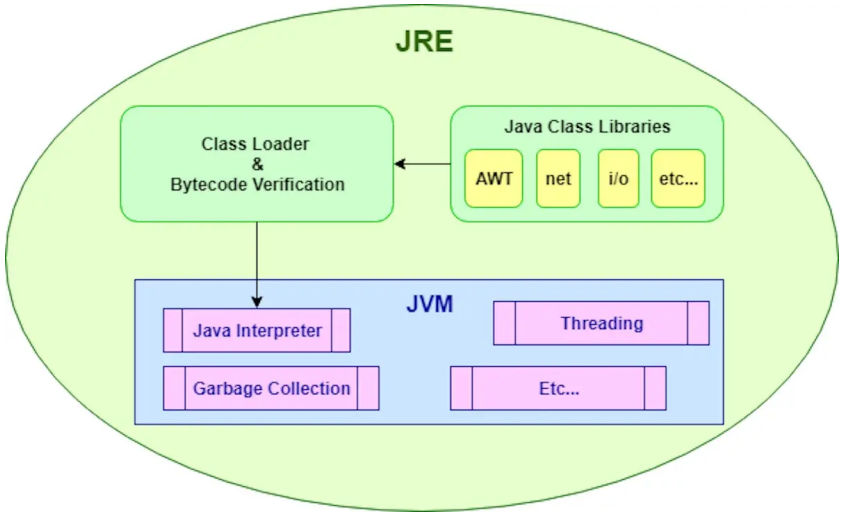
- Loads code

- Verifies code
- Executes code
- Provides runtime environment

JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. 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.

The implementation of JVM is also actively released by other companies besides Sun Micro Systems.



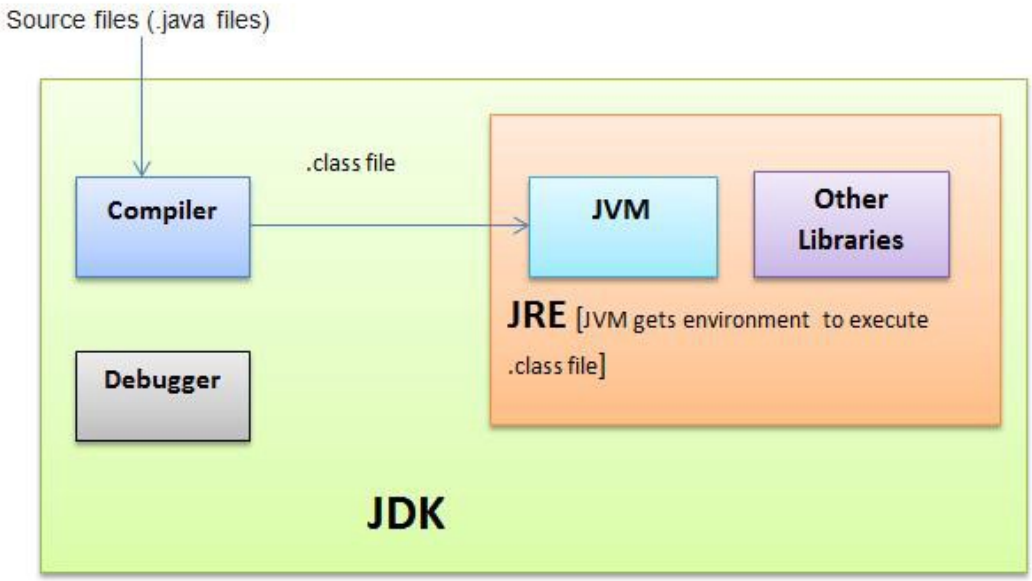
JDK

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.



JVM ARCHITECTURE

It is:

1. A specification where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.
2. An implementation Its implementation is known as JRE (Java Runtime Environment).

3. Runtime Instance Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

What it does

The JVM performs following operation:

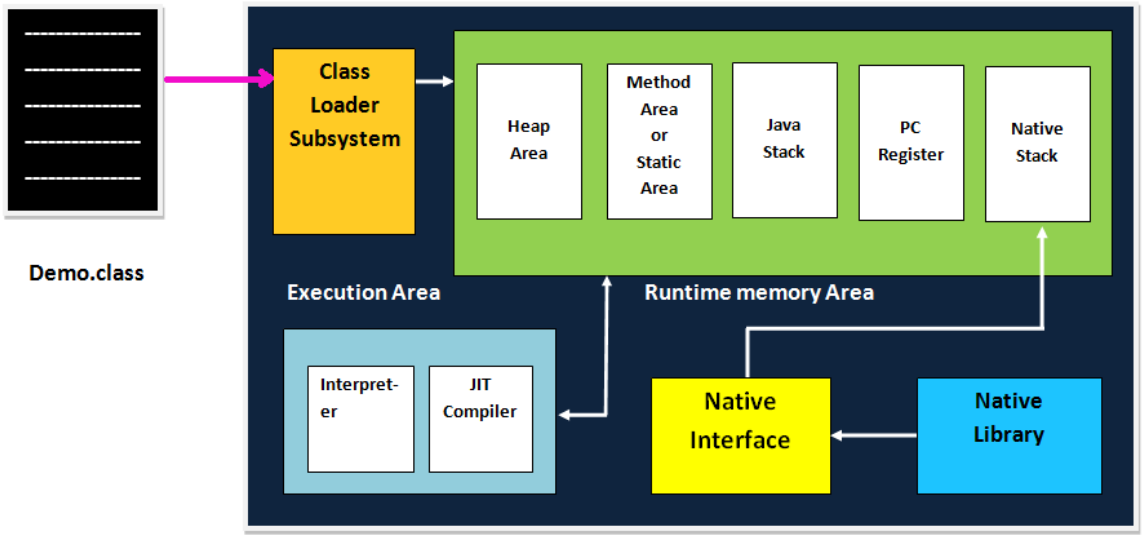
- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

JVM Architecture

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



1) Classloader

Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built-in classloaders in Java.

1. **Bootstrap ClassLoader:** This is the first classloader which is the super class of Extension classloader. It loads the *rt.jar* file which contains all class files of Java Standard Edition like java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes etc.
2. **Extension ClassLoader:** This is the child classloader of Bootstrap and parent classloader of System classloader. It loades the jar files located inside *$JAVA_HOME/jre/lib/ext* directory.

3. **System/Application ClassLoader:** This is the child classloader of Extension classloader. It loads the classfiles from classpath. By default, classpath is set to current directory. You can change the classpath using "-cp" or "-classpath" switch. It is also known as Application classloader.

2) Class(Method) Area

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

3) Heap

It is the runtime data area in which objects are allocated.

4) Stack

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.

Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

5) Program Counter Register

PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

6) Native Method Stack

It contains all the native methods used in the application.

7) Execution Engine

It contains:

1. A virtual processor
2. Interpreter: Read bytecode stream then execute the instructions.
3. Just-In-Time(JIT) compiler: It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

8) Java Native Interface

Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc. Java uses JNI framework to send output to the Console or interact with OS libraries.

TOKENS

The Java compiler breaks the line of code into text (words) is called **Java tokens**. These are the smallest element of the Java program. The Java compiler identified these words as tokens. These tokens are separated by the delimiters. It is useful for compilers to detect errors. Remember that the delimiters are not part of the Java tokens.

token <= identifier | keyword | separator | operator | literal | comment

Types of tokens in JAVA

Types of Tokens:

Java token includes the following:

- ***Keywords***
- ***Identifiers***
- ***Literals***
- ***Operators***
- ***Separators***
- ***Comments***

Keywords:

These are the pre-defined reserved words of any programming language. Each keyword has a special meaning. It is always written in lower case.

There are totally 67 keywords are there in java and some of the keywords are listed below:

01. abstract	02. boolean	03. byte	04. break	05. class
06. case	07. catch	08. char	09. continue	10. default
11. do	12. double	13. else	14. extends	15. final
16. finally	17. float	18. for	19. if	20. implements
21. import	22. instanceof	23. int	24. interface	25. long

Identifiers:

Identifiers are used to name a variable, constant, function, class, and array. It usually defined by the user. It uses letters, underscores, or a dollar sign as the first character. The label is also known as a special kind of identifier that is used in the goto statement. Remember that the identifier name must be different from the reserved keywords. There are some rules to declare identifiers are:

- The first letter of an identifier must be a letter, underscore or a dollar sign. It cannot start with digits but may contain digits.
- The whitespace cannot be included in the identifier.
- Identifiers are case sensitive.

Some valid identifiers are:

1. PhoneNumber
2. PRICE

Some invalid identifiers are:

1.3cyber

2.cy ber

Literals:

In programming literal is a notation that represents a fixed value (constant) in the source code. It can be categorized as an integer literal, string literal, Boolean literal, etc. It is defined by the programmer. Once it has been defined cannot be changed. Java provides five types of literals are as follows:

- Integer
- Floating Point
- Character
- String
- Boolean

Literal	Type
23	int
9.86	double
false, true	boolean
'K', '7', '-'	char
"javatpoint"	String
null	any reference type

Operators:

In programming, operators are the special symbol that tells the compiler to perform a special operation. Java provides different types of operators that can be classified according to the functionality they provide. There are eight types of operators in Java, are as follows:

- Arithmetic Operators
- Assignment Operators
- Relational Operators
- Unary Operators
- Logical Operators
- Ternary Operators
- Bitwise Operators
- Shift Operators

Operator	Symbols
Arithmetic	+ , - , / , * , %
Unary	++ , - - , !
Assignment	= , += , -= , *= , /= , %= , ^=
Relational	==, != , < , >, <= , >=

Logical	&& , ||
Ternary	(Condition) ? (Statement1) : (Statement2);
Bitwise	& , | , ^ , ~
Shift	<< , >> , >>>

Separators:

The separators in Java is also known as **punctuators**. There are nine separators in Java, are as follows:

Square Brackets [ ]: It is used to define array elements. A pair of square brackets represents the single-dimensional array, two pairs of square brackets represent the two-dimensional array.

Parentheses (): It is used to call the functions and parsing the parameters.

Curly Braces {}: The curly braces denote the starting and ending of a code block.

Comma (,): It is used to separate two values, statements, and parameters.

Assignment Operator (=): It is used to assign a variable and constant.

Semicolon (;): It is the symbol that can be found at end of the statements. It separates the two statements.

Period (.): It separates the package name form the sub-packages and class. It also separates a variable or method from a reference variable.

Comments:

Comments allow us to specify information about the program inside our Java code. Java compiler recognizes these comments as tokens but excludes it form further processing. The Java compiler treats comments as whitespaces. Java provides the following two types of comments:

- **Line Oriented:** It begins with a pair of forwarding slashes (**//**).
- **Block-Oriented:** It begins with /* and continues until it founds ***/**.

DATATYPES:

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

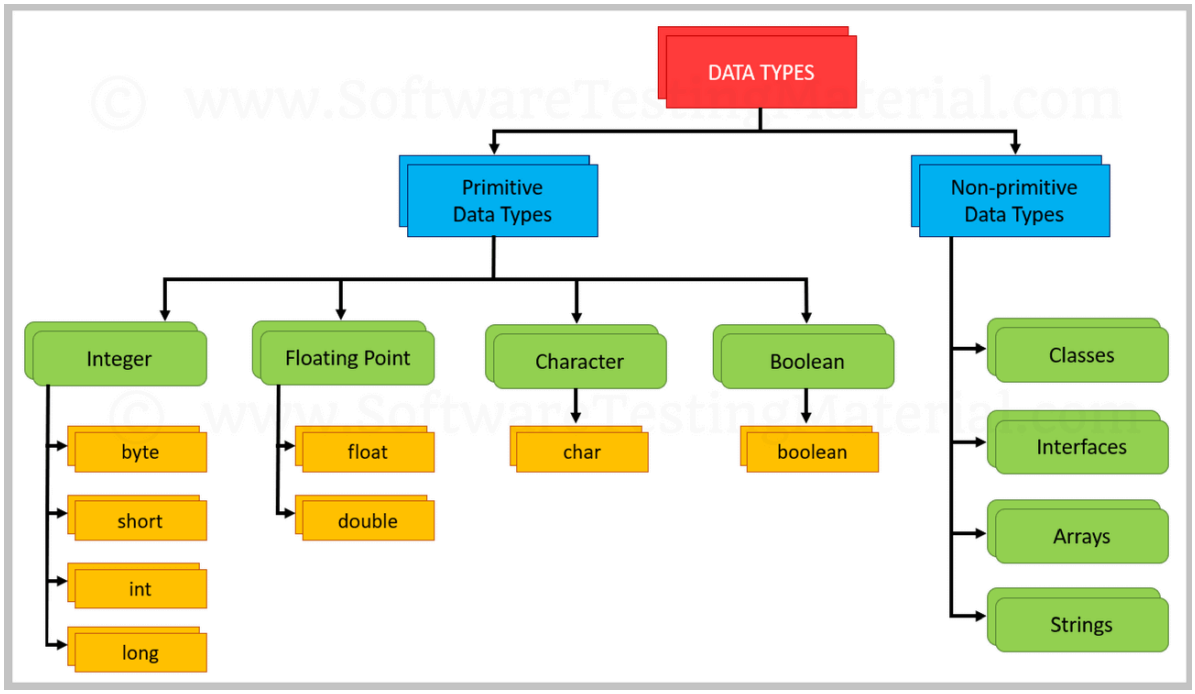
1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

Java primitive data types:

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

There are 8 types of primitive data types:

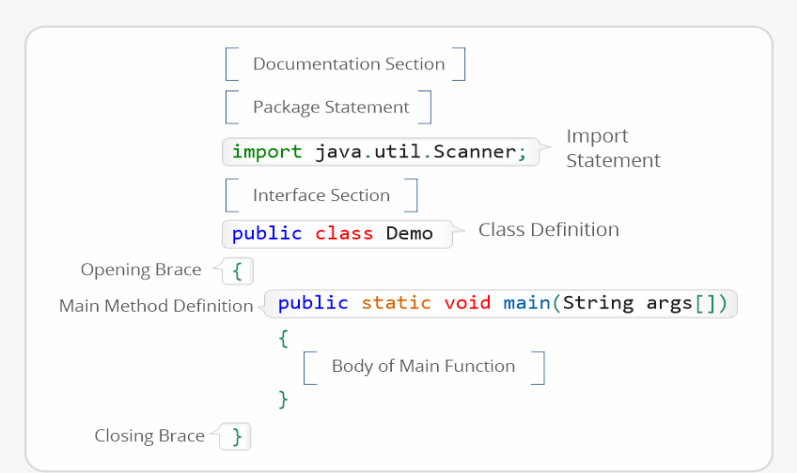
- boolean data type
- byte data type
- char data type
- short data type
- int data type
- long data type
- float data type
- double data type



Data Type	Default Value	Default size	Range	Examples
boolean	false	1 bit	true,false	true,false
char	'\u0000'	2 byte	0 to 2^16-1	‘a’,’b’,’\n’,’\101’
byte	0	1 byte	-128 to 127	(none)
short	0	2 byte	-32,768 to 32,767	(none)
int	0	4 byte	-2147483648 to 2147483647	-2,-1,0,1,2
long	0L	8 byte	0 to 2^64-1	-2L,-1L,0L,1L,2L
float	0.0f	4 byte	Upto 7 decimal digits	2.356f,506.12789f

double	0.0d	8 byte	Upto 16 decimal digits	12.123456789d

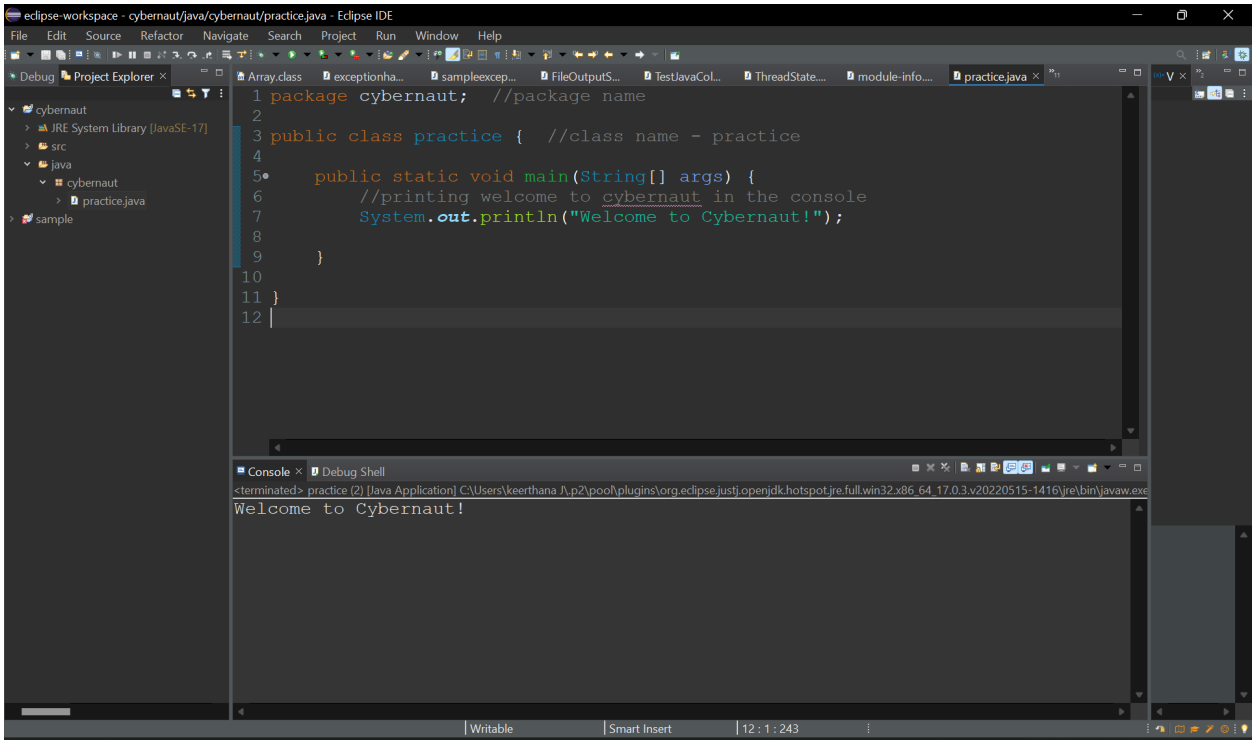
STRUCTURE OF A JAVA PROGRAM:



Program to print "Welcome to Cybernaut!"

```
package cybernaut;  //package name

public class practice {  //class name - practice

        public static void main(String[] args) {

                //printing welcome to cybernaut in the console

                System.out.println("Welcome to Cybernaut!");
```

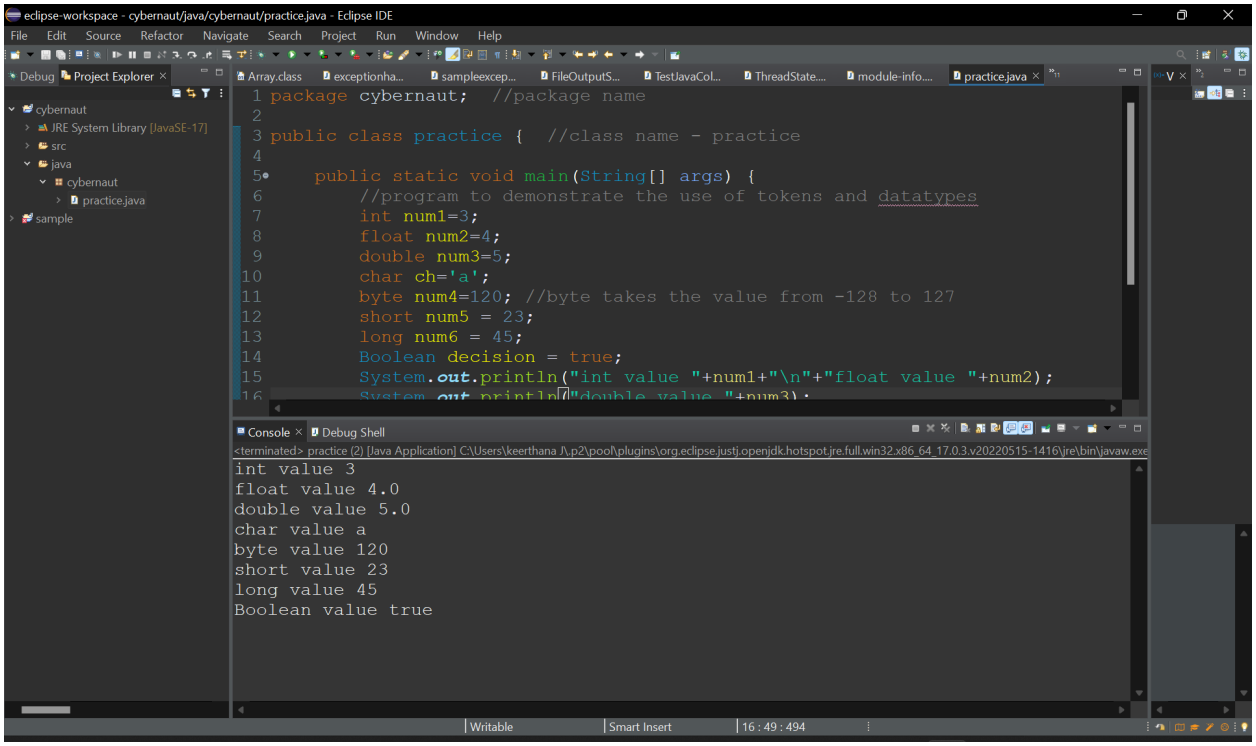
```
    }
}
```



Program to demonstrate the use of Tokens and Datatypes:

```
package cybernaut;  //package name

public class practice {  //class name - practice

        public static void main(String[] args) {

                //program to demonstrate the use of tokens and datatypes

                int num1=3;
```

```
        float num2=4;

        double num3=5;

        char ch='a';

        byte num4=120; //byte takes the value from -128 to 127

        short num5 = 23;

        long num6 = 45;

        Boolean decision = true;

        System.out.println("int value "+num1+"\n"+"float value "+num2);

        System.out.println("double value "+num3);

        System.out.println("char value "+ch);

        System.out.println("byte value "+num4);

        System.out.println("short value "+num5);

        System.out.println("long value "+num6);

        System.out.println("Boolean value "+decision);

    }

}
```



OPERATORS IN JAVA

Operator in Java is a symbol that is used to perform operations. For example: +, -, *, / etc.

Java Operator Precedence

Operator Type	Category	Precedence

Unary	postfix	*expr*++ *expr*--
	prefix	++*expr* --*expr* +*expr* -*expr* ~ !
Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>
Relational	comparison	< > <= >= instanceof
	equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||

Ternary	ternary	? :
Assignment	assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Java Unary Operator

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

- incrementing/decrementing a value by one
- negating an expression
- inverting the value of a boolean

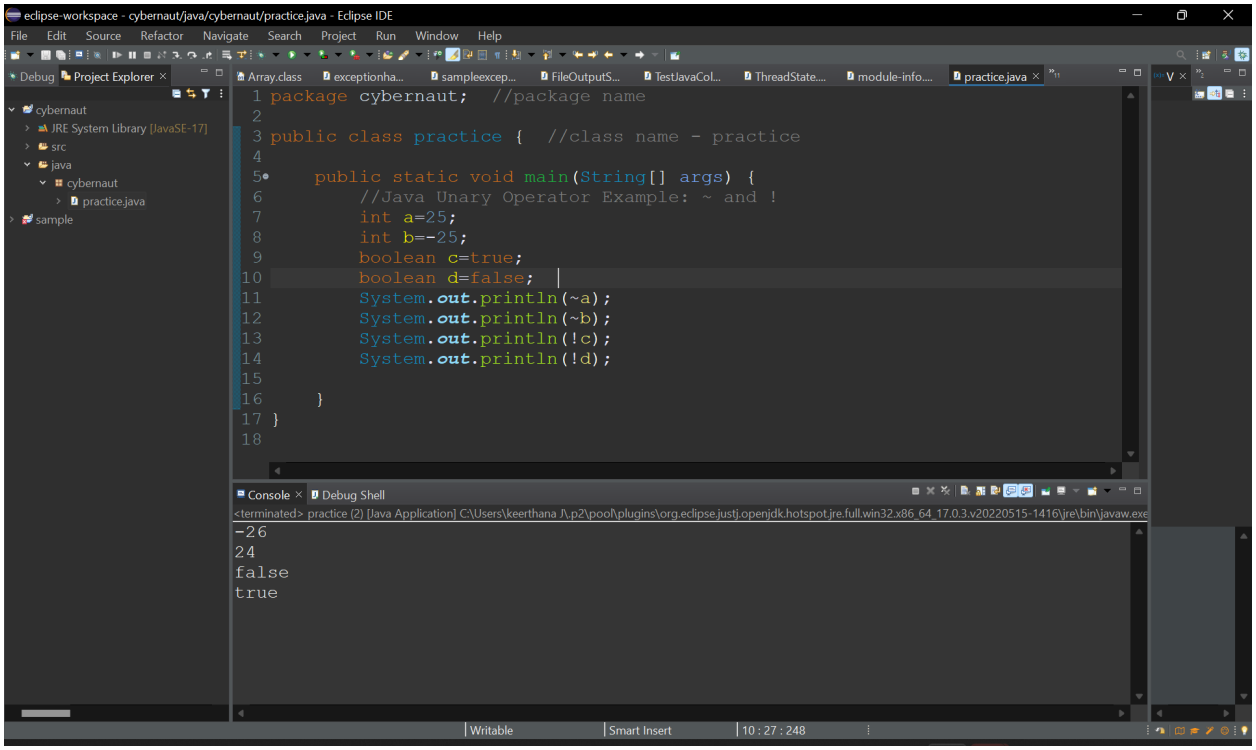
Java Unary Operator Example: ++ and --

```
package cybernaut;  //package name

public class practice {  //class name - practice

        public static void main(String[] args) {

                //Java Unary Operator Example: ++ and --

                int x=25;

                System.out.println(x++); //25

                System.out.println(++x); //27

                System.out.println(x--); //27

                System.out.println(--x); //25

        }
```

```
}
```

Java Unary Operator Example: ~ and !

```
package cybernaut;  //package name

public class practice {  //class name - practice

	public static void main(String[] args) {

		//Java Unary Operator Example: ~ and !

		int a=25;

		int b=-25;

		boolean c=true;

		boolean d=false;

		System.out.println(~a);

		System.out.println(~b);

		System.out.println(!c);

		System.out.println(!d);

	}

}
```

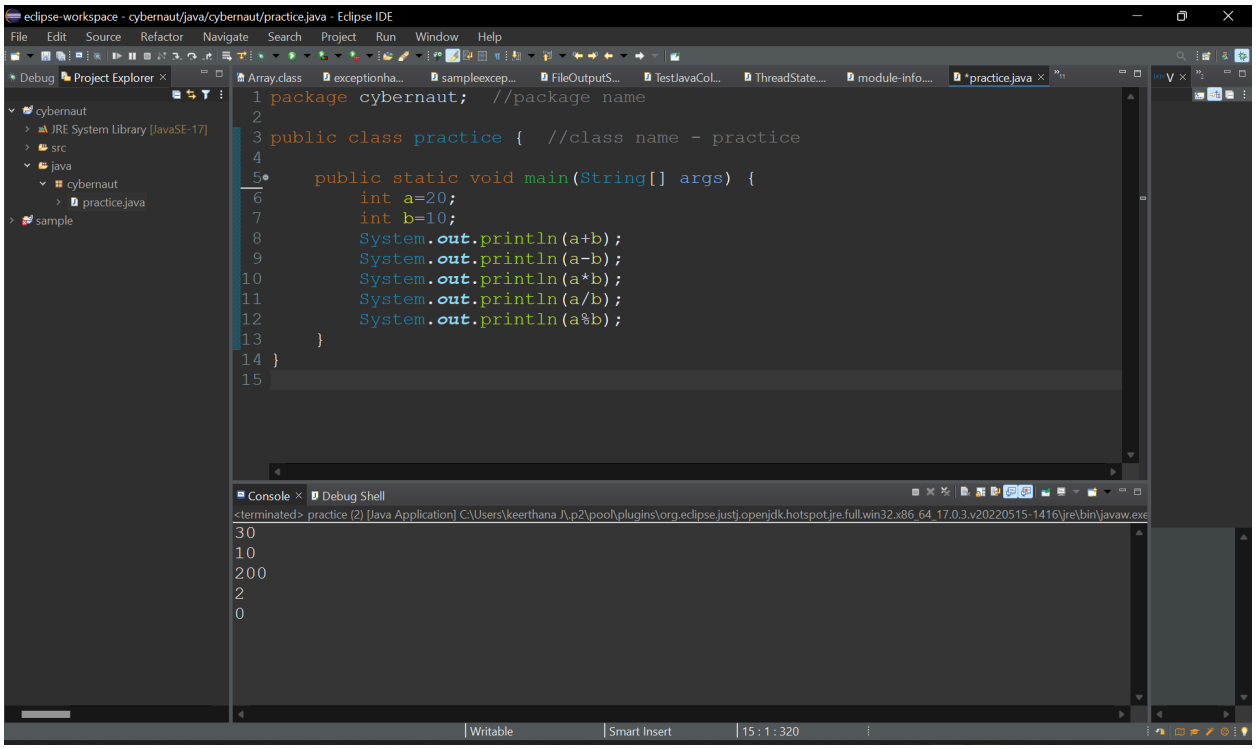


Java Arithmetic Operators

Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

```
package cybernaut;  //package name

public class practice {  //class name - practice

        public static void main(String[] args) {

                int a=20;

                int b=10;

                System.out.println(a+b);

                System.out.println(a-b);

                System.out.println(a*b);
```

```
            System.out.println(a/b);

            System.out.println(a%b);

        }

}
```



Java Relational Operator

Java Relational Operators are a bunch of binary operators used to check for relations between two operands, including equality, greater than, less than, etc. They return a boolean result after the comparison and are extensively used in looping statements as well as conditional if-else statements and so on. The general format of representing relational operator is:

Syntax:

variable1 relation_operator variable2

Let us look at each one of the relational operators in Java:

1) Equal to operator (==)

This operator is used to check whether the two given operands are equal or not. The operator returns true if the operand at the left-hand side is equal to the right-hand side, else false.

Syntax:

var1 == var2

Illustration:

var1 = "GeeksforGeeks"

var2 = 20

var1 == var2 results in false

Example:

```
import java.io.*;
class Equal {
    public static void main(String[] args)
    {
        int var1 = 5, var2 = 10, var3 = 5;
        System.out.println("Var1 = " + var1);
```

```
        System.out.println("Var2 = " + var2);
        System.out.println("Var3 = " + var3);
        System.out.println("var1 == var2: " + (var1 == var2));
        System.out.println("var1 == var3: " + (var1 == var3));
    }
}
```

Output :

```
Var1 = 5
Var2 = 10
Var3 = 5
var1 == var2: false
var1 == var3: true
```

2) Not equal to Operator(!=)

This operator is used to check whether the two given operands are equal or not. It functions opposite to that of the equal-to-operator. It returns true if the operand at the left-hand side is not equal to the right-hand side, else false.

Syntax:

```
var1 != var2
```

Illustration:

```
var1 = "GeeksforGeeks"
var2 = 20
var1 != var2 results in true
```

Example:

```
import java.io.*;
class NotEqual {
```

```
    public static void main(String[] args)
    {
        int var1 = 5, var2 = 10, var3 = 5;
        System.out.println("Var1 = " + var1);
        System.out.println("Var2 = " + var2);
        System.out.println("Var3 = " + var3);
        System.out.println("var1 == var2: " + (var1 != var2));
        System.out.println("var1 == var3: " + (var1 != var3));
    }
}
```

Output :

```
Var1 = 5
Var2 = 10
Var3 = 5
var1 == var2: true
var1 == var3: false
```

3) Greater than operator(>)

This checks whether the first operand is greater than the second operand or not. The operator returns true when the operand at the left-hand side is greater than the right-hand side.

Syntax:

```
var1 > var2
```

Illustration:

```
var1 = 30
var2 = 20
```

var1 > var2 results in true

Example:

```
import java.io.*;
class Greater {
    public static void main(String[] args)
    {
        int var1 = 30, var2 = 20, var3 = 5;
        System.out.println("Var1 = " + var1);
        System.out.println("Var2 = " + var2);
        System.out.println("Var3 = " + var3);
        System.out.println("var1 > var2: " + (var1 > var2));
        System.out.println("var3 > var1: “  + (var3 >= var1));
    }
}
```

Output :

```
Var1 = 30
Var2 = 20
Var3 = 5
var1 > var2: true
var3 > var1: false
```

4) Less than Operator(<)

This checks whether the first operand is less than the second operand or not. The operator returns true when the operand at the left-hand side is less than the right-hand side. It functions opposite to that of the greater-than operator.

Syntax:

var1 < var2

Illustration:

var1 = 10

var2 = 20

var1 < var2 results in true

Example:

```
class Lesser {
   public static void main(String[] args)
   {
      int var1 = 10, var2 = 20, var3 = 5;
      System.out.println("Var1 = " + var1);
      System.out.println("Var2 = " + var2);
      System.out.println("Var3 = " + var3);
      // Comparing var1 and var2 and
      // printing corresponding boolean value
      System.out.println("var1 < var2: " + (var1 < var2));
      // Comparing var2 and var3 and
      // printing corresponding boolean value
      System.out.println("var2 < var3: " + (var2 < var3));
   }
}
```

Output :

```
Var1 = 10
Var2 = 20
Var3 = 5
var1 < var2: true
var2 < var3: false
```

5) Greater than or equal to (>=)

This checks whether the first operand is greater than or equal to the second operand or not. The operator returns true when the operand at the left-hand side is greater than or equal to the right-hand side.

Syntax:

var1 >= var2

Illustration:

var1 = 20

var2 = 20

var3 = 10

var1 >= var2 results in true

var2 >= var3 results in true

Example:

```
import java.io.*;
class GreaterEqual {
    public static void main(String[] args)
    {
        int var1 = 20, var2 = 20, var3 = 10;
        System.out.println("Var1 = " + var1);
        System.out.println("Var2 = " + var2);
```

```
        System.out.println("Var3 = " + var3);

        // Comparing var1 and var2 and
        // printing corresponding boolean value
        System.out.println("var1 >= var2: " + (var1 >= var2));

        // Comparing var2 and var3 and
        // printing corresponding boolean value
        System.out.println("var2 >= var3: " + (var3 >= var1));
    }
}
```

Output :

```
Var1 = 20
Var2 = 20
Var3 = 10
var1 >= var2: true
var2 >= var3: false
```

6) Less than or equal to (<=)

This checks whether the first operand is less than or equal to the second operand or not. The operator returns true when the operand at the left-hand side is less than or equal to the right-hand side.

Syntax:

```
var1 <= var2
```

Illustration:

var1 = 10

var2 = 10

var3 = 9

var1 <= var2 results in true

var2 <= var3 results in false

Example:

```
import java.io.*;
class LesserEqual {
    public static void main(String[] args)
    {
        int var1 = 10, var2 = 10, var3 = 9;
        System.out.println("Var1 = " + var1);
        System.out.println("Var2 = " + var2);
        System.out.println("Var3 = " + var3);

        // Comparing var1 and var2 and
        // printing corresponding boolean value
        System.out.println("var1 <= var2: " + (var1 <= var2));

        // Comparing var2 and var3 and
        // printing corresponding boolean value
        System.out.println("var2 <= var3: " + (var2 <= var3));
    }
}
```

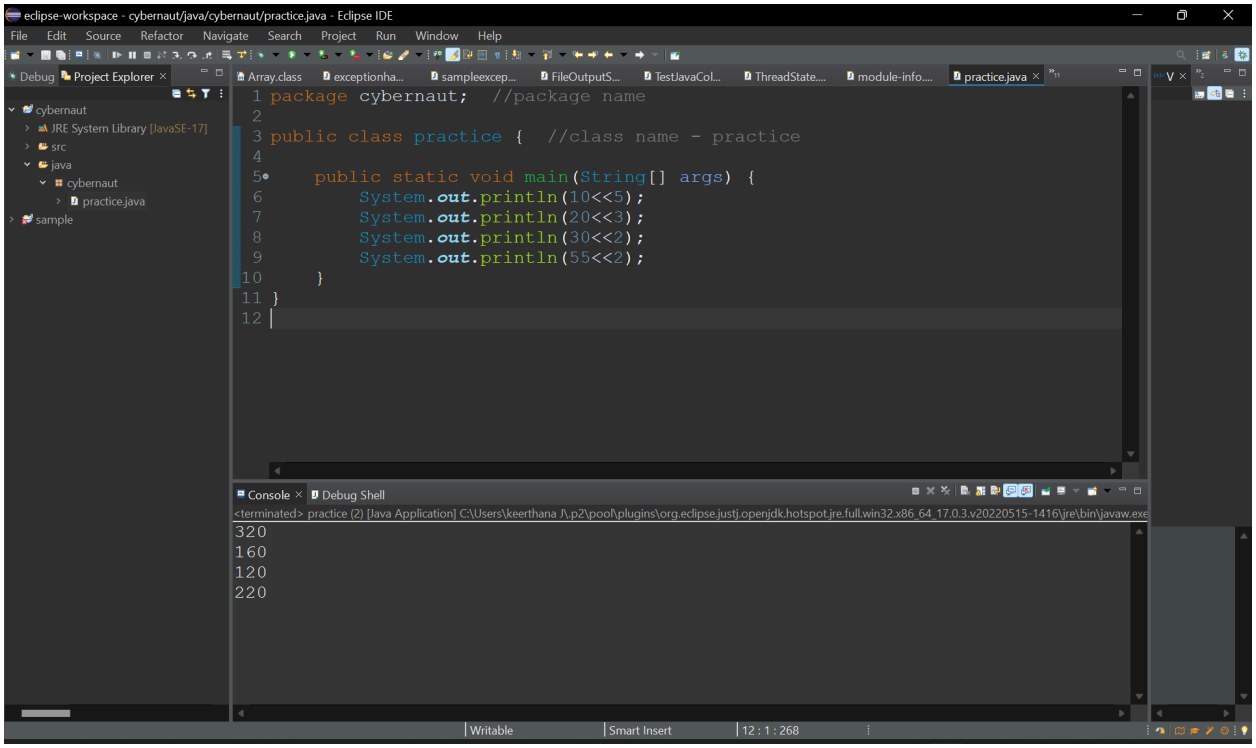
Output :

```
Var1 = 10
Var2 = 10
Var3 = 9
var1 <= var2: true
var2 <= var3: false
```

Java Left Shift Operator

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

```
package cybernaut;  //package name

public class practice {  //class name - practice

	public static void main(String[] args) {

		System.out.println(10<<5);

		System.out.println(20<<3);

		System.out.println(30<<2);

		System.out.println(55<<2);

	}

}
```

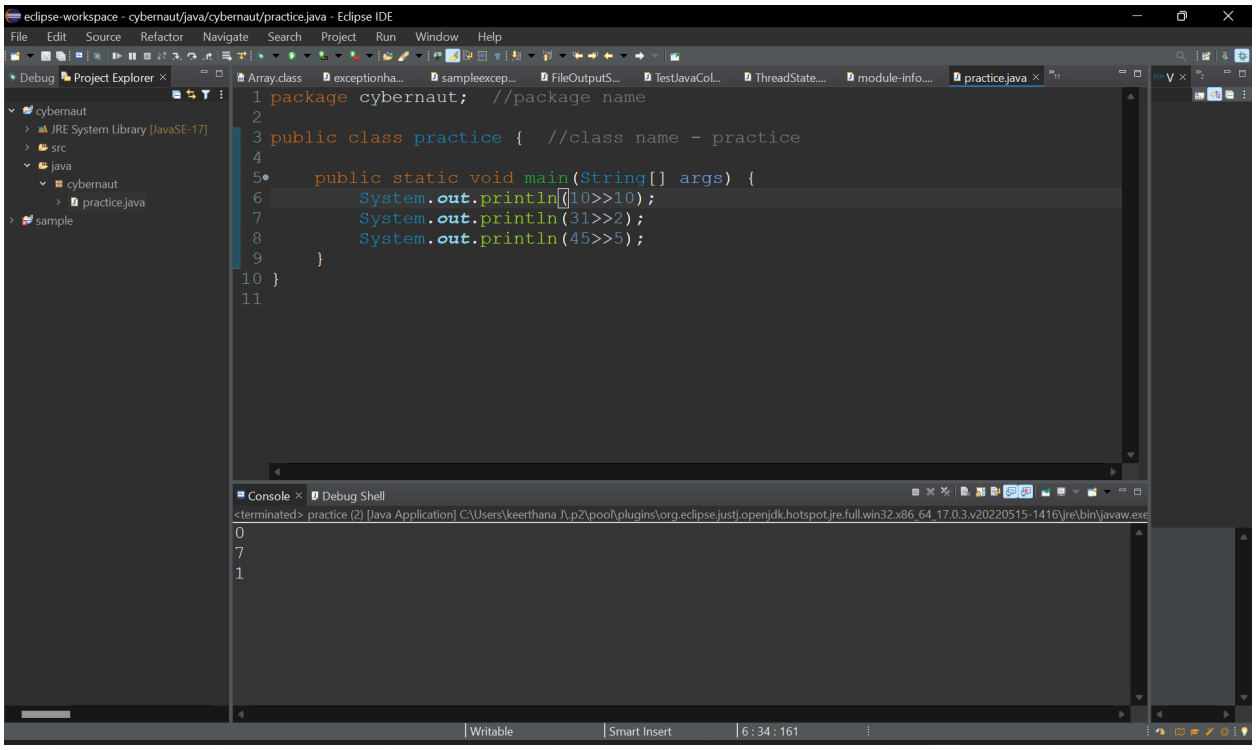


Java Right Shift Operator

The Java right shift operator >> is used to move the value of the left operand to right by the number of bits specified by the right operand.

```
package cybernaut;  //package name

public class practice {  //class name - practice

        public static void main(String[] args) {

                System.out.println(10>>10);

                System.out.println(31>>2);

                System.out.println(45>>5);

        }
```

}



Java AND Operator Example: Logical && and Bitwise &

The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true.

The bitwise & operator always checks both conditions whether first condition is true or false.

package cybernaut; //package name

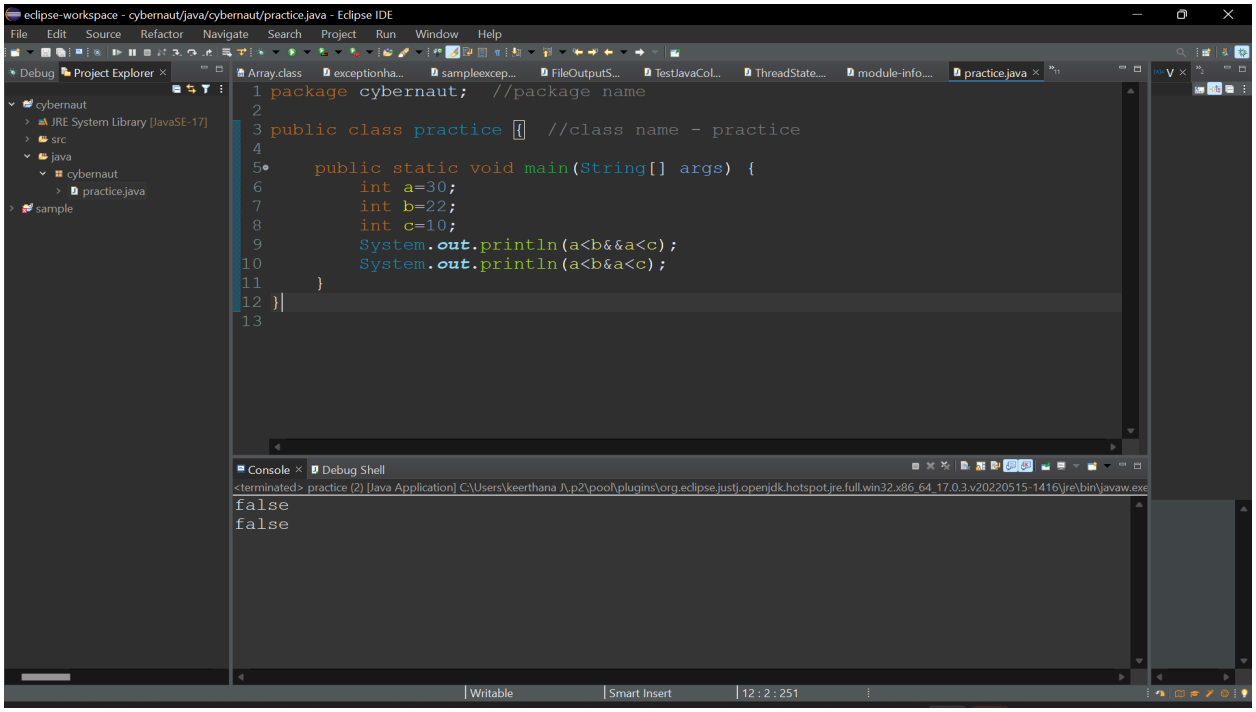
public class practice { //class name - practice

public static void main(String[] args) {

int a=30;

int b=22;

```
        int c=10;

        System.out.println(a<b&&a<c);

        System.out.println(a<b&a<c);

    }

}
```



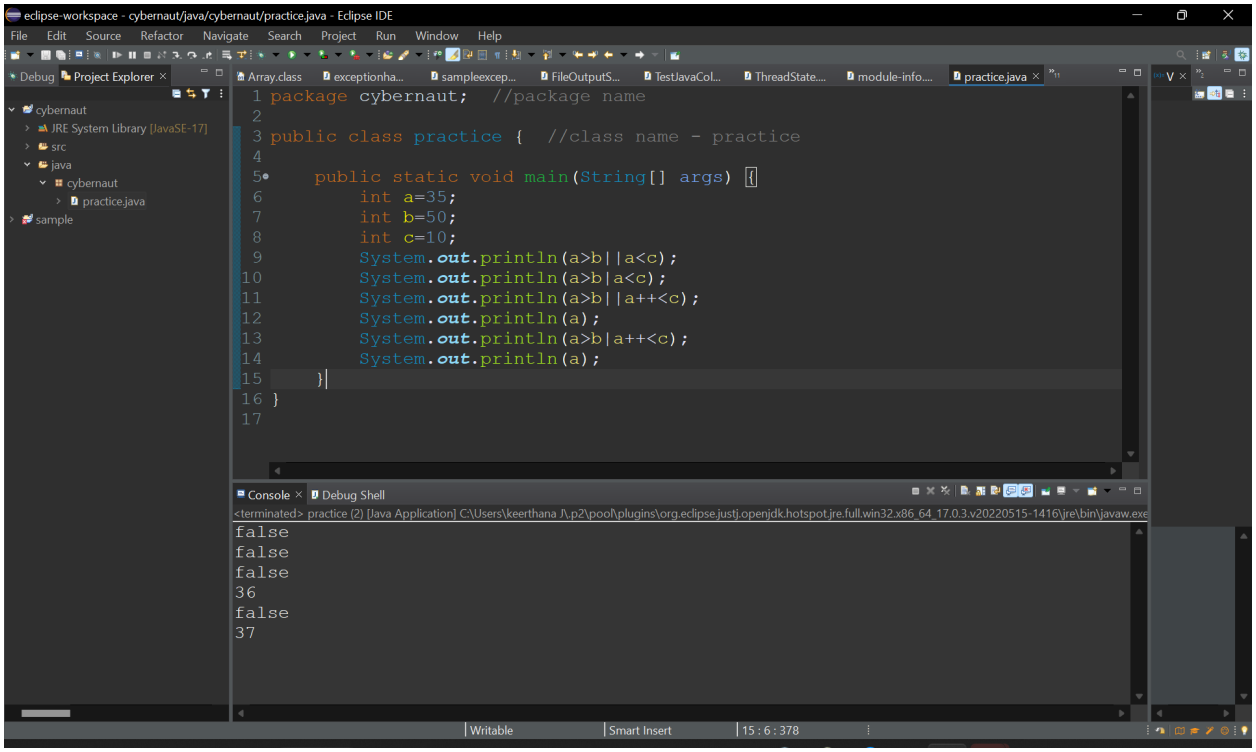
Java OR Operator Example: Logical || and Bitwise |

The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false.

The bitwise | operator always checks both conditions whether first condition is true or false.

package cybernaut; //package name

```
public class practice {  //class name - practice
	public static void main(String[] args) {
		int a=35;
		int b=50;
		int c=10;
		System.out.println(a>b||a<c);
		System.out.println(a>b|a<c);
		System.out.println(a>b||a++<c);
		System.out.println(a);
		System.out.println(a>b|a++<c);
		System.out.println(a);
	}
}
```

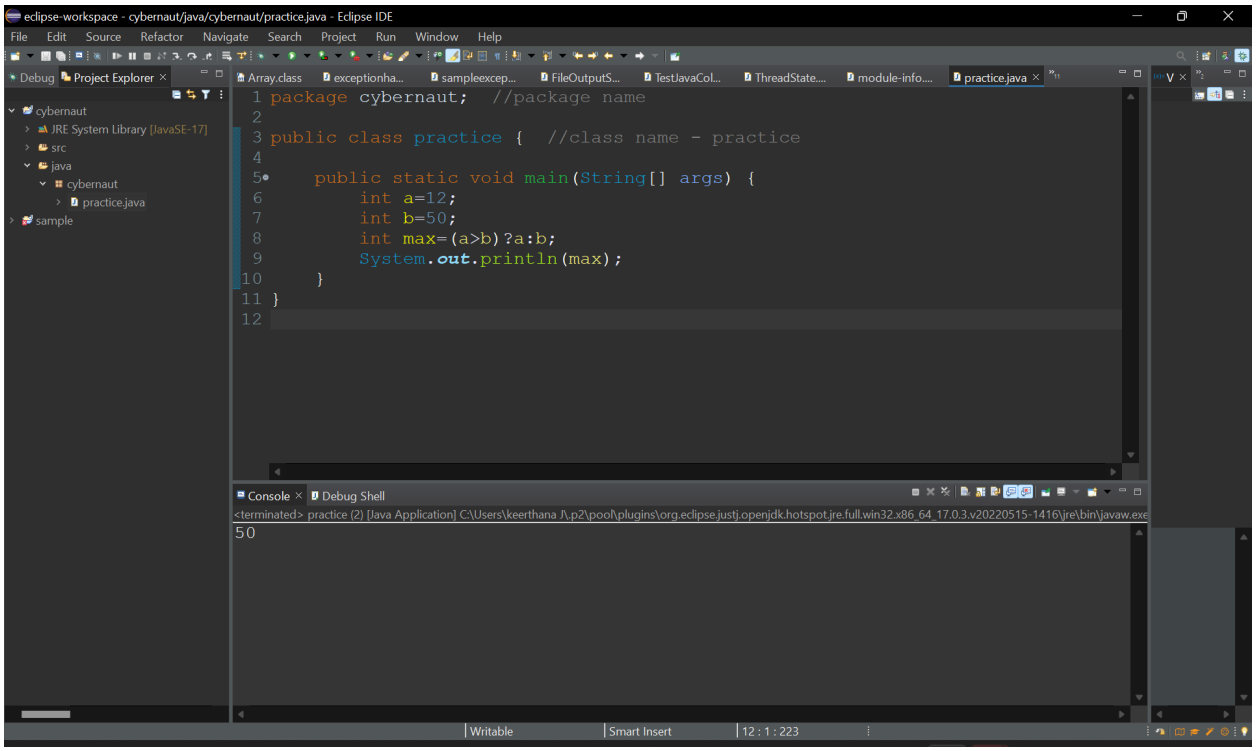


Java Ternary Operator

Java Ternary operator is used as one line replacement for if-then-else statement and used a lot in Java programming. It is the only conditional operator which takes three operands.

```
package cybernaut;  //package name

public class practice {  //class name - practice

        public static void main(String[] args) {

                int a=12;

                int b=50;

                int max=(a>b)?a:b;

                System.out.println(max);
```

```
    }
}
```

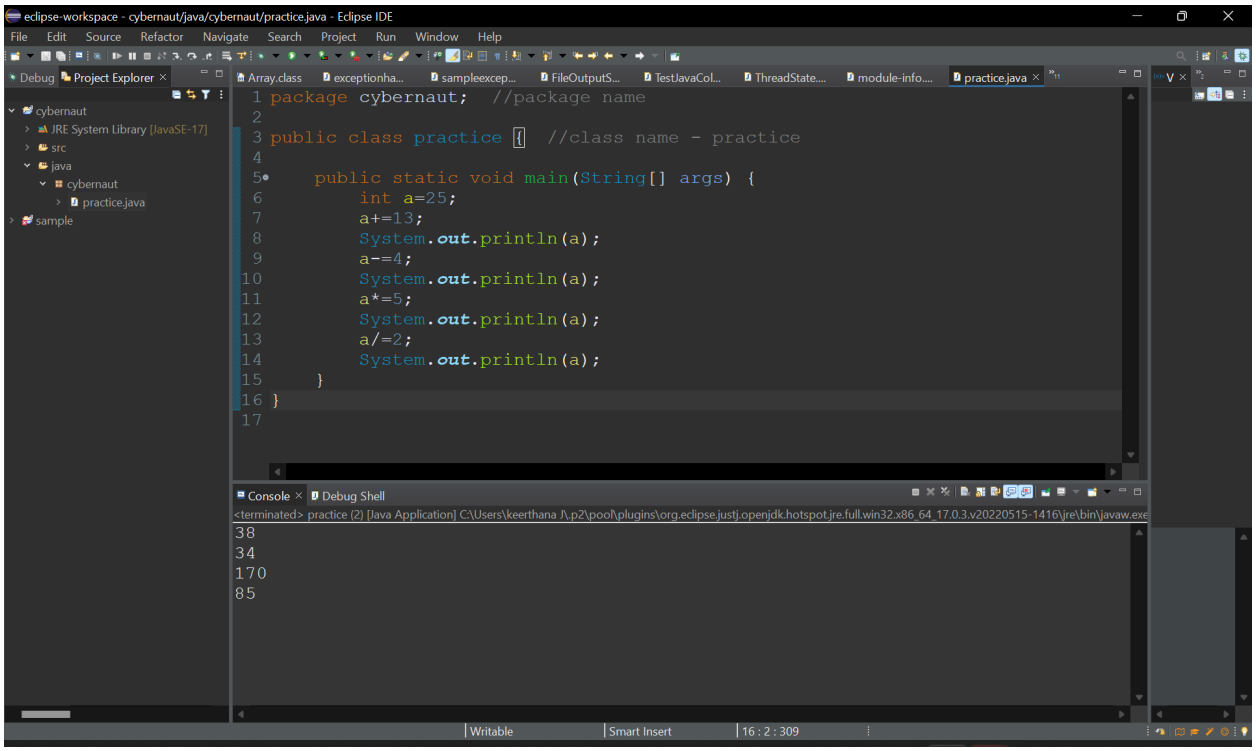


Java Assignment Operator

Java assignment operator is one of the most common operators. It is used to assign the value on its right to the operand on its left.

```
package cybernaut;  //package name


public class practice {  //class name - practice
        public static void main(String[] args) {
                int a=25;
```

```
        a+=13;

        System.out.println(a);

        a-=4;

        System.out.println(a);

        a*=5;

        System.out.println(a);

        a/=2;

        System.out.println(a);

    }

}
```



Java Control Statements | Control Flow in Java

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

Java provides three types of control flow statements.

1. **Decision Making statements**
    - if statements
    - switch statement
2. **Loop statements**
    - do while loop
    - while loop
    - for loop
    - for-each loop
3. **Jump statements**
    - break statement
    - continue statement

Decision-Making statements:

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression

and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

1) If Statement:

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

1. Simple if statement
2. if-else statement
3. if-else-if ladder
4. Nested if-statement

Let's understand the if-statements one by one.

1) Simple if statement:

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

Syntax of if statement is given below.

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

Example:

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

Output :

```
x + y is greater than 20
```

2) if-else statement

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

Syntax:

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

Example 1:

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

Output :

```
x + y is greater than 20
```

3) if-else-if ladder:

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

Syntax :

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

```
else {
statement 2; //executes when all the conditions are false
}
```

Example :

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

Output :

```
Delhi
```

4. Nested if-statement

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

Syntax :

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

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

Program :

```
public class Student {
public static void main(String[] args) {
String address = "Delhi, India";
if(address.endsWith("India")) {
if(address.contains("Meerut")) {
System.out.println("Your city is Meerut");
}else if(address.contains("Noida")) {
System.out.println("Your city is Noida");
}else {
System.out.println(address.split(",")[0]);
}
}else {
System.out.println("You are not living in India");
}
}
}
```

Output:

```
Delhi
```

Switch Statement:

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

Points to be noted about switch statement:

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

Syntax :

```
switch (expression){
   case value1:
    statement1;
    break;
   .
```

```
    .
    .
    case valueN:
statementN;
        break;
    default:
        default statement;
}
```

Example :

Program:

```
public class Student implements Cloneable {
public static void main(String[] args) {
int num = 2;
switch (num){
case 0:
System.out.println("number is 0");
break;
case 1:
System.out.println("number is 1");
break;
default:
System.out.println(num);
}
}
}
```

Output :

```
2
```

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used.

Java Arrays

Normally, an array is a collection of similar type of elements which has contiguous memory location.

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

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

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

array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multi dimentional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.

Advantages

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

Disadvantages

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

There are two types of array.

- Single Dimensional Array

- Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

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

Instantiation of an Array in Java

arrayRefVar=**new** datatype[size];

Example:

```
//Java Program to illustrate how to declare, instantiate, initialize

//and traverse the Java array.

class Testarray{

public static void main(String args[]){

int a[]=new int[5];//declaration and instantiation

a[0]=10;//initialization

a[1]=20;
```

```
a[2]=70;

a[3]=40;

a[4]=50;

//traversing array

for(int i=0;i<a.length;i++)//length is the property of array

System.out.println(a[i]);

}}
```

Output:

```
10

20

70

40

50
```

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

1. `int a[]={33,3,4,5};//declaration, instantiation and initialization`

Let's see the simple example to print this array.

```
//Java Program to illustrate the use of declaration, instantiation

//and initialization of Java array in a single line
```

```
class Testarray1{

public static void main(String args[]){

int a[]={33,3,4,5};//declaration, instantiation and initialization

//printing array

for(int i=0;i<a.length;i++)//length is the property of array

System.out.println(a[i]);

}}
```

Output:

33

3

4

5

For-each Loop for Java Array

We can also print the Java array using **for-each loop**. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

```
for(data_type variable:array){
```

```
//body of the loop
}
```

Example:

Let us see the example of print the elements of Java array using the for-each loop.

```
//Java Program to print the array elements using for-each loop
class Testarray1{
public static void main(String args[]){
int arr[]={33,3,4,5};
//printing array using for-each loop
for(int i:arr)
System.out.println(i);
}}
```

Output:

33

3

4

5

Passing Array to a Method in Java

We can pass the java array to method so that we can reuse the same logic on any array.

Let's see the simple example to get the minimum number of an array using a method.

```
//Java Program to demonstrate the way of passing an array

//to method.

class Testarray2{

//creating a method which receives an array as a parameter

static void min(int arr[]){

int min=arr[0];

for(int i=1;i<arr.length;i++)

 if(min>arr[i])

  min=arr[i];


System.out.println(min);

}


public static void main(String args[]){

int a[]={33,3,4,5};//declaring and initializing an array

min(a);//passing array to method

}}
```

Output:

3

Anonymous Array in Java

Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

```
//Java Program to demonstrate the way of passing an anonymous array
//to method.
public class TestAnonymousArray{
//creating a method which receives an array as a parameter
static void printArray(int arr[]){
for(int i=0;i<arr.length;i++)
System.out.println(arr[i]);
}

public static void main(String args[]){
printArray(new int[]{10,22,44,66});//passing anonymous array to method
}}
```

Output:

10

22

44

66

Returning Array from the Method

We can also return an array from the method in Java.

```
//Java Program to return an array from the method

class TestReturnArray{

//creating method which returns an array

static int[] get(){

return new int[]{10,30,50,90,60};

}


public static void main(String args[]){

//calling method which returns an array

int arr[]=get();

//printing the values of an array

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

System.out.println(arr[i]);
```

```
}}
```

Output:

```
10

30

50

90

60
```

ArrayIndexOutOfBoundsException

The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array in negative, equal to the array size or greater than the array size while traversing the array.

```
//Java Program to demonstrate the case of

//ArrayIndexOutOfBoundsException in a Java Array.

public class TestArrayException{

public static void main(String args[]){

int arr[]={50,60,70,80};

for(int i=0;i<=arr.length;i++){

System.out.println(arr[i]);
```

```
}

}}
```

Output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4

        at TestArrayException.main(TestArrayException.java:5)

50

60

70

80
```

Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java

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

4. dataType []arrayRefVar[];

Example to instantiate Multidimensional Array in Java

```
int[][] arr=new int[3][3];//3 row and 3 column
```

Example to initialize Multidimensional Array in Java

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

Example of Multidimensional Java Array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
//Java Program to illustrate the use of multidimensional array
class Testarray3{
public static void main(String args[]){
//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
//printing 2D array
```

```
for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
   System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}
}}
```

Output:

```
1 2 3
2 4 5
4 4 5
```

Addition of 2 Matrices in Java

Let's see a simple example that adds two matrices.

```
//Java Program to demonstrate the addition of two matrices in Java
class Testarray5{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,3,4},{3,4,5}};
```

```
int b[][]={{1,3,4},{3,4,5}};


//creating another matrix to store the sum of two matrices

int c[][]=new int[2][3];


//adding and printing addition of 2 matrices

for(int i=0;i<2;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

}


}}
```

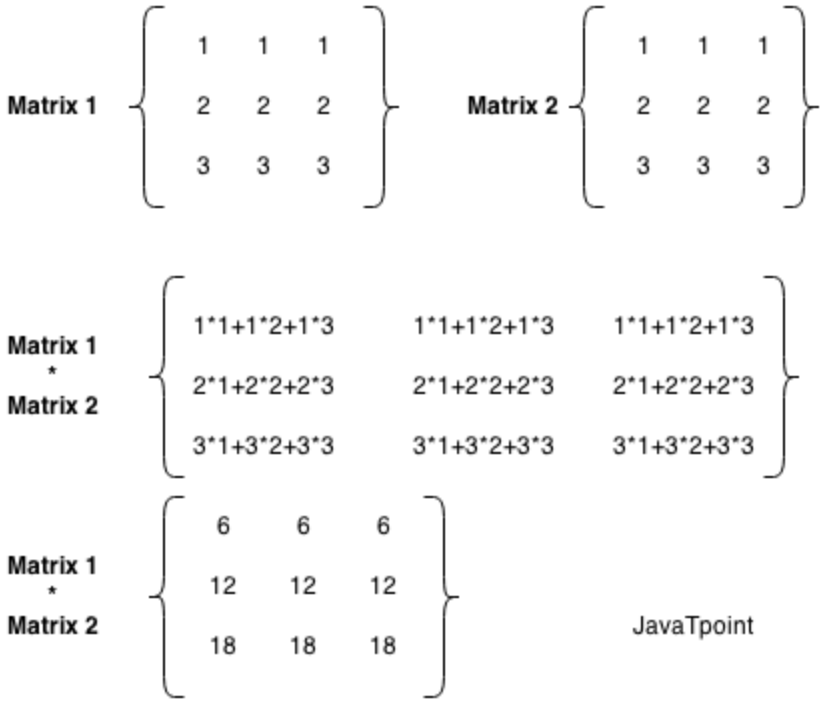
Output:

2 6 8

6 8 10

Multiplication of 2 Matrices in Java

In the case of matrix multiplication, a one-row element of the first matrix is multiplied by all the columns of the second matrix which can be understood by the image given below.




```
//Java Program to multiply two matrices
public class MatrixMultiplicationExample{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,1,1},{2,2,2},{3,3,3}};
int b[][]={{1,1,1},{2,2,2},{3,3,3}};

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

//multiplying and printing multiplication of 2 matrices
```

```
for(int i=0;i<3;i++){
for(int j=0;j<3;j++){
c[i][j]=0;
for(int k=0;k<3;k++)
{
c[i][j]+=a[i][k]*b[k][j];
}//end of k loop
System.out.print(c[i][j]+" ");  //printing matrix element
}//end of j loop
System.out.println();//new line
}
}}
```

Output:

6 6 6

12 12 12

18 18 18

Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

```
//Java Program to illustrate the jagged array
class TestJaggedArray{
```

```
public static void main(String[] args){
    //declaring a 2D array with odd columns
    int arr[][] = new int[3][];
    arr[0] = new int[3];
    arr[1] = new int[4];
    arr[2] = new int[2];
    //initializing a jagged array
    int count = 0;
    for (int i=0; i<arr.length; i++)
        for(int j=0; j<arr[i].length; j++)
            arr[i][j] = count++;

    //printing the data of a jagged array
    for (int i=0; i<arr.length; i++){
        for (int j=0; j<arr[i].length; j++){
            System.out.print(arr[i][j]+" ");
        }
        System.out.println();//new line
    }
}
```

}

Output:

0 1 2

3 4 5 6

7 8

For-each Loop for Java Array

We can also print the Java array using **for-each loop**. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

```
for(data_type variable:array){
//body of the loop
}
```

Let us see the example of print themente eles of Java array using the for-each loop.

```
//Java Program to print the array elements using for-each loop
class Testarray1{
public static void main(String args[]){
int arr[]={33,3,4,5};
//printing array using for-each loop
for(int i:arr)
System.out.println(i);
```

8. }}

Output:

33

3

4

5

WHILE LOOP

The Java *while loop* is used to iterate a part of the program repeatedly until the specified Boolean condition is true. As soon as the Boolean condition becomes false, the loop automatically stops.

The while loop is considered as a repeating if statement. If the number of iteration is not fixed, it is recommended to use the while loop.

Syntax:

```
while (condition){
//code to be executed
Increment / decrement statement
}
```

Parts of while loop:

1. Condition: It is an expression which is tested. If the condition is true, the loop body is executed and control goes to update expression. When the condition becomes false, we exit the while loop.

Example:

i <=100

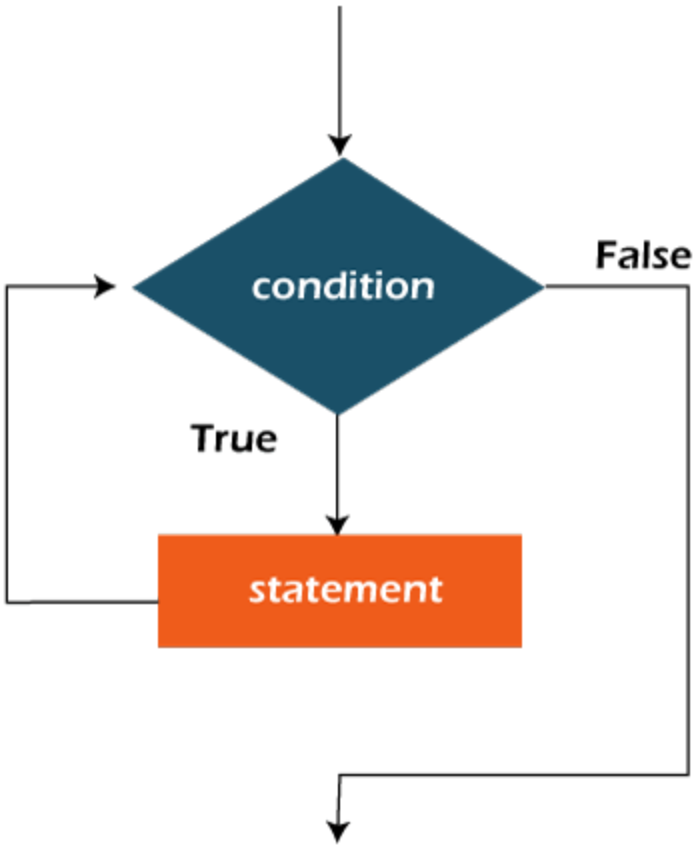
2. Update expression: Every time the loop body is executed, this expression increments or decrements loop variable.

Example:

I++;

Flow chart of while loop:

Here, the important thing about while loop is that, sometimes it may not even execute. If the condition to be tested results into false, the loop body is skipped and first statement after the while loop will be executed.



Example:

```
public class WhileExample {
public static void main(String[] args) {
   int i=1;
   while(i<=10){
      System.out.println(i);
   i++;
   }
}
}
```

Output:

1

2

3

4

5

6

7

8

9

10

FOR LOOP

The Java *for loop* is used to iterate a part of the program several times. If the number of iteration is **fixed**, it is recommended to use for loop.

There are three types of for loops in Java.

- Simple for Loop
- For-each or Enhanced for Loop
- Labeled for Loop

Parts of for loop:

Initialization: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.

Condition: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.

Increment/Decrement: It increments or decrements the variable value. It is an optional condition.

Statement: The statement of the loop is executed each time until the second condition is false.

Syntax:

```
for(initialization; condition; increment/decrement){
//statement or code to be executed
```

```
        }
```

Flowchart:

Example:

```
//Java Program to demonstrate the example of for loop
//which prints table of 1
public class ForExample {
public static void main(String[] args) {
   //Code of Java for loop
   for(int i=1;i<=10;i++){
      System.out.println(i);
   }
}
}
```

Output:

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

Java Nested for Loop

If we have a for loop inside the another loop, it is known as nested for loop. The inner loop executes completely whenever outer loop executes.

Example:

```
public class NestedForExample {
public static void main(String[] args) {
//loop of i
for(int i=1;i<=3;i++){
//loop of j
for(int j=1;j<=3;j++){
    System.out.println(i+" "+j);
}//end of i
}//end of j
}
}
```

Output:

11

12

13

21

22

23

31

32

33

Example :

```
public class PyramidExample {
public static void main(String[] args) {
for(int i=1;i<=5;i++){
for(int j=1;j<=i;j++){
System.out.print("* ");
}
System.out.println();//new line
}
}
}
```

Output:

```
*
* *
* * *
* * * *
* * * * *
```

For-each Loop

The for-each loop is used to traverse array or collection in Java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation.

It works on the basis of elements and not the index. It returns element one by one in the defined variable.

Syntax:

```
for(data_type variable : array_name){
//code to be executed
}
```

Example:

```
//Java For-each loop example which prints the
//elements of the array
public class ForEachExample {
public static void main(String[] args) {
   //Declaring an array
   int arr[]={12,23,44,56,78};
   //Printing array using for-each loop
   for(int i:arr){
      System.out.println(i);
   }
}
}
```

Output:

12

23

44

56

78

Labeled For Loop

We can have a name of each Java for loop. To do so, we use label before the for loop. It is useful while using the nested for loop as we can break/continue specific for loop.

Syntax:

```
labelname:
 for(initialization; condition; increment/decrement){
 //code to be executed
 }
```

Example:

```
//A Java program to demonstrate the use of labeled for loop
public class LabeledForExample {
public static void main(String[] args) {
  //Using Label for outer and for loop
  aa:
```

```
    for(int i=1;i<=3;i++){
      bb:
        for(int j=1;j<=3;j++){
          if(i==2&&j==2){
            break aa;
          }
          System.out.println(i+" "+j);
        }
    }
}
      }
```

Output:

11

12

13

21

If you use break bb;, it will break inner loop only which is the default behaviour of any loop.

LOOP MATRIX

- It is a matrix with stream j as column and loop i as row, $[a_{ij}]$. If a loop i includes a stream j, the element in the loop matrix $a_{ij} = 1$, otherwise $a_{ij} = 0$. In the case of above, the loop matrix could be given as in Eqn.(1): (1)

•It's one of the most useful data structures in the programming world. You can use a two-dimensional array to make finite state machine (FSM) solve state-based problems, you can use a 2D array to

create board games like Chess, Sudoku, and Tic-Tac-To and you can even use a two-dimensional array to create 2D arcade games e.g. Tetris, Super Mario Bros and so on. Whatever you see on your screen is nothing but a 2D array that is populated using tiles.

EXAMPLE

```
public class TwoDimensionalArrayDemo{
public static void main(String args[]) {
int[][] board = new int[4][4];
for (int row = 0; row < board.length; row++) {
for (int col = 0; col < board[row].length; col++) {
board[row][col] = row * col; } }

for (int row = 0; row < board.length; row++) {
for (int col = 0; col < board[row].length; col++) {
board[row][col] = row * col;
System.out.print(board[row][col] + "\t"); }
System.out.println(); } } }
```

OUTPUT

```
0 0 0 0
0 1 2 3
0 2 4 6
0 3 6 9
```

Do-while Loop

The Java *dow*hile loop is called an exit control loop. Therefore, unlike while loop and for loop, th-*while loop* is used to iterate a part of the program repeatedly, until the specified condition is true. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use a do-while loop.

Java do-we do-while check the condition at the end of loop body. The Java *do-while loop* is executed at least once because condition is checked after loop body.

Syntax:

```
do{
//code to be executed / loop body
//update statement
}while (condition);
```

The different parts of do-while loop:

1. Condition: It is an expression which is tested. If the condition is true, the loop body is executed and control goes to update expression. As soon as the condition becomes false, loop breaks automatically.

Example:

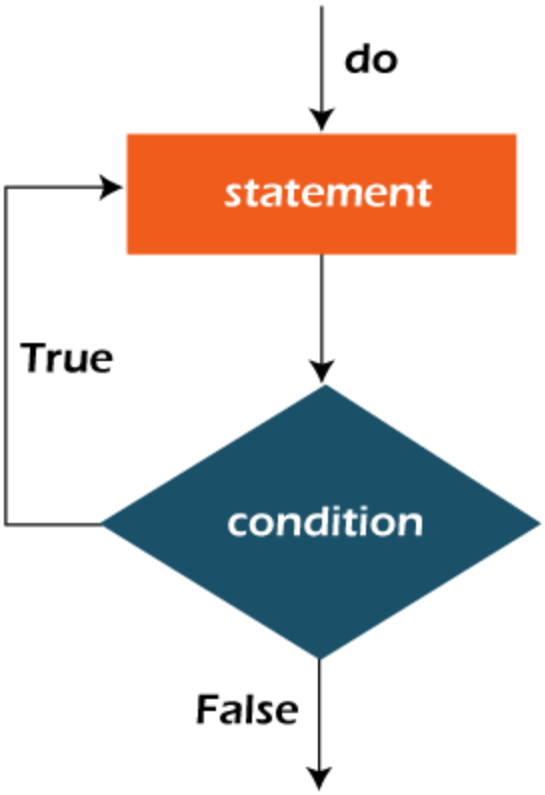
i <=100

2. Update expression: Every time the loop body is executed, the this expression increments or decrements loop variable.

Example:

```
I++;
```

Flowchart of do-while loop:



Example:

```
public class DoWhileExample {
public static void main(String[] args) {
   int i=1;
   do{
      System.out.println(i);
```

```
    i++;
    }while(i<=10);
}
}
```

Output:

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

Example:

```
public class DoWhileExample {
public static void main(String[] args) {
    int i=1;
    do{
```

```
        System.out.println(i);
    i++;
    }while(i<=10);
}
}
```

Syntax:

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

Example:

```
public class DoWhileExample2 {
public static void main(String[] args) {
do{
        System.out.println("infinitive do while loop");
    }while(true);
}
}
```

Output:

```
Infinitive do-while Loop
```

Infinitive do-while Loop

Infinitive do-while Loop

Ctrl+c

We need to enter ctrl+c to get out of the loop

OBJECT ORIENTED PROGRAMMING

Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

1. **Object**

   An object in Java is the physical as well as a logical entity. It is an entity that has state and behaviour.

2. **Class**

   A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

3. **Inheritance**

   When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

4. **Polymorphism**
   If one task is performed in different ways, it is known as polymorphism.

5. **Abstraction**

   Abstraction is a process of hiding the implementation details and showing only functionality to the user.

6. **Encapsulation**

   Encapsulation in Java refers to integrating data (variables) and code (methods) into a single unit. In encapsulation, a class's variables are hidden from other classes and can only be accessed by the methods of the class in which they are found.

OBJECTS AND CLASS IN JAVA

Class

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

- Fields
- Methods
- Constructors
- Blocks
- Nested class and interface

Syntax:

```
class <class_name>{
    field;
    method;
```

```
}
```

Object

In object-oriented programming technique, we design a program using objects and classes.

An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only. An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc.

An object has three characteristics:

- **State**(Instance variables):represents the data (value) of an object.
- **Behavior(methods):** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity(unique id):** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization
- By factory method etc.

CONSTRUCTORS IN JAVA

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory. It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default. It is called constructor because it constructs the values at the time of object creation.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Rules for creating Java Constructor

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)

A constructor is called "Default Constructor" when it doesn't have any parameter.

2. Parameterized constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

Examples

```
//Java Program to demonstrate the use of the parameterized constructor.
class Student4{
   int id;
   String name;
   //creating a parameterized constructor
   Student4(int i,String n){
   id = i;
   name = n;
   }
```

```
  //method to display the values
  void display(){System.out.println(id+" "+name);}

  public static void main(String args[]){
  //creating objects and passing values
  Student4 s1 = new Student4(111,"Karan");
  Student4 s2 = new Student4(222,"Aryan");
  //calling method to display the values of object
  s1.display();
  s2.display();
 }
}
```

Output:

111 Karan

222 Aryan

111 Karan

222 Aryan

111 Karan

222 Aryan

111 Karan

222 Arya

METHODS

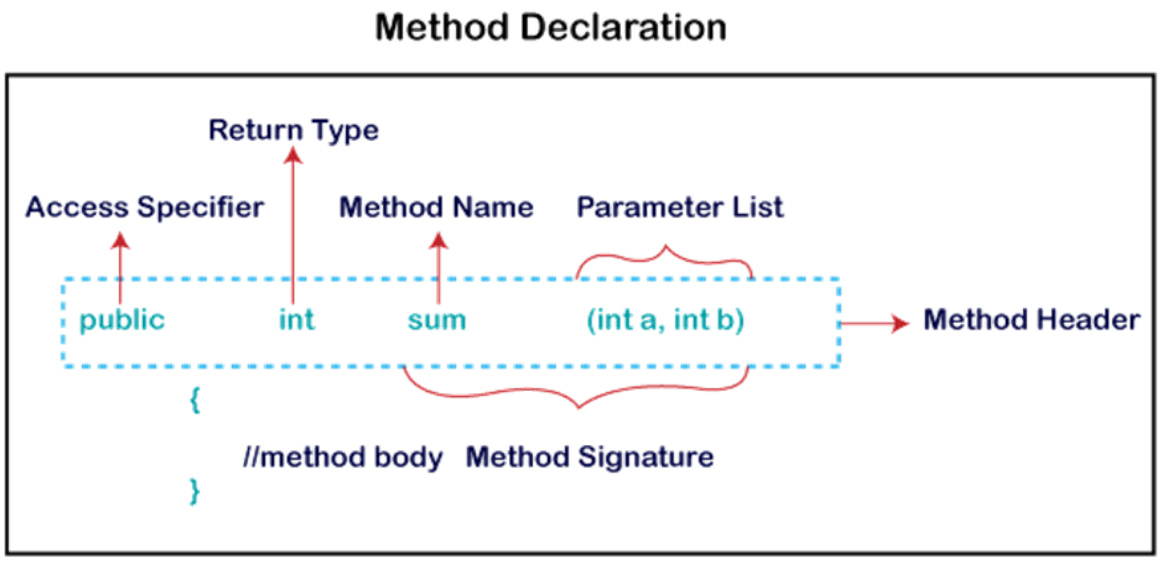
Introduction

A method is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the reusability of code. We write a method once and use it many times. We are not required to write code again and again. A method is executed only when we call or invoke it.

The most important method in Java is the main() method.

Method Declaration

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method header**, as we have shown in the following figure.



Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list**.

Access Specifier: Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier:

- **Public:** The method is accessible by all classes when we use public specifier in our application.
- **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
- **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

Return Type: Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

Method Name: It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction().** A method is invoked by its name.

Parameter List: It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

Method Body: It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Naming Conventions

While defining a method, remember that the method name must be a verb and start with a lowercase letter. If the method name has more than two words, the first name must be a verb followed by an adjective or noun. In the multi-word method name, the first letter of each word must be in uppercase except the first word.

For example,

Single-word method name: sum(), area()

Multi-word method name: areaOfCircle(), stringComparision()

Types Of Methods

There are two types of methods in Java:

1. **Predefined Method**:

   In Java, predefined methods are the method that is already defined in the Java class libraries. It is also known as the standard library method or built-in method. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are length(), equals(), compareTo(), sqrt(), etc. Each and every predefined method is defined inside a class. For example, **print()** method is defined in the **java.io.PrintStream** class.

2. **User-defined Method**:

   The method written by the user or programmer is known as a user-defined method. These methods are modified according to the requirement.

   For Example,

```
public static void findEvenOdd(int num)
{
//method body
        if(num%2==0)
                System.out.println(num+" is even");
        else
                System.out.println(num+" is odd");
}
```

Call by Value

There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

Example 1:

```
class Operation{
 int data=50;
 void change(int data){
 data=data+100;//changes will be in the local variable only
 }
 public static void main(String args[]){
   Operation op=new Operation();
   System.out.println("before change "+op.data);
   op.change(500);
   System.out.println("after change "+op.data);
 }
}
```

OUTPUT:

```
Output:before change 50
       after change 50
```

Example 2:

```
class Operation2{
 int data=50;
 void change(Operation2 op){
     op.data=op.data+100;//changes will be in the instance variable
 }
 public static void main(String args[]){
   Operation2 op=new Operation2();
   System.out.println("before change "+op.data);
   op.change(op);//passing object
```

```
    System.out.println("after change "+op.data);
  }
 }
```

OUTPUT:

```
Output:before change 50
       after change 150
```

Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.

The constructor name must be same as the class name.	The method name may or may not be same as the class name.

RETURN STATEMENT IN JAVA

In Java programming, the return statement is used for returning a value when the execution of the block is completed. The return statement inside a loop will cause the loop to break and further statements will be ignored by the compiler.

Returning a Value from a Method

In Java, every method is declared with a return type such as int, float, double, string, etc.

These return types required a return statement at the end of the method. A return keyword is used for returning the resulted value.

The void return type doesn't require any return statement. If we try to return a value from a void method, the compiler shows an error.

Following are the important points must remember while returning a value:

- The return type of the method and type of data returned at the end of the method should be of the same type. For example, if a method is declared with the float return type, the value returned should be of float type only.
- The variable that stores the returned value after the method is called should be a similar data type otherwise, the data might get lost.
- If a method is declared with parameters, the sequence of the parameter must be the same while declaration and method call.

Syntax:

The syntax of a return statement is the return keyword is followed by the value to be returned.

```
return returnvalue;
```

Example 1:

```
public class SampleReturn1
{
   /* Method with an integer return type and no arguments */
   public int CompareNum()
   {
```

```
        int x = 3;
        int y = 8;
        System.out.println("x = " + x + "\ny = " + y);
        if(x>y)
            return x;
        else
            return y;
    }

    /* Driver Code */
    public static void main(String ar[])
    {
        SampleReturn1 obj = new SampleReturn1();
        int result = obj.CompareNum();
        System.out.println("The greater number among x and y is: " + result);
    }
}
```

OUTPUT:

```
x = 3
y = 8
The greater number among x and y is: 8
```

Example 2:

```
public class SampleReturn2
{
```

```
    /* Method with an integer return type and arguments */
    public int CompareNum(int x, int y)
    {
        System.out.println("x = " + x + "\ny = " + y);
        if(x>y)
            return x;
        else
            return y;
    }

    /* Driver Code */
    public static void main(String ar[])
    {
        SampleReturn2 obj = new SampleReturn2();
        int result = obj.CompareNum(15,24);
        System.out.println("The greater number among x and y is: " + result);
    }
}
```

OUTPUT:

```
x = 15
y = 24
The greater number among x and y is: 24
```

Returning a Class or Interface

A method can have the class name as its return type. Therefore it must return the object of the exact class or its subclass.

An interface name can also be used as a return type but the returned object must implement methods of that interface.

The following Java program shows the implementation of a class name as a return type.

```
class SumReturn
{
   private int a;
   public SumReturn(int i)
   {
      a = i;
   }
   /*The addition method returns a SumReturn object with adding 100 into it. */
   public SumReturn addition()
   {
      SumReturn result = new SumReturn(a + 100);
        return result;
   }
   public void display()
   {
      System.out.println("Additon result: " + a);
   }
}
public class SampleReturn3
```

```
{
    /* Driver Code */
    public static void main(String[] args)
    {
        SumReturn obj1 = new SumReturn(50);
        SumReturn obj2;
        /* addition method returns a reference of SumReult class */
        obj2 = obj1.addition();
        obj2.display();
    }
}
```

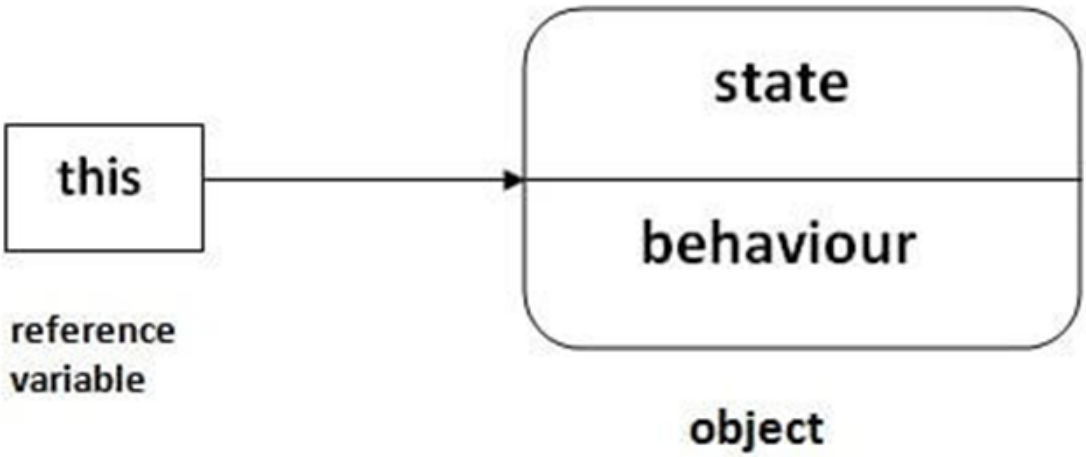
OUTPUT:

```
Additon result: 150
```

THIS KEYWORD

Introduction

In Java, this is a **reference variable** that refers to the current object.



Usage Types

This keyword can be used in six different ways. They are:

1. **this can be used to refer current class instance variable.**

   Example of how a program can be written with this keyword

Without this keyword	With this keyword

class Student{
int rollno;
String name;
float fee;
Student(**int** rollno,String name,**float** fee){
rollno=rollno;
name=name;
	class Student{
	int rollno;
	String name;
	float fee;
	Student(**int** rollno,String name,**float** fee){
	this.rollno=rollno;
	this.name=name;

```
    fee=fee;

    }

    Void display(){

    System.out.println(rollno+"
    "+name+" "+fee);}

    }

    class TestThis1{

    public static void main(String
    args[]){

    Student                    s1=new
    Student(111,"ankit",5000f);

    Student                    s2=new
    Student(112,"sumit",6000f);

    s1.display();

    s2.display();

}}
```

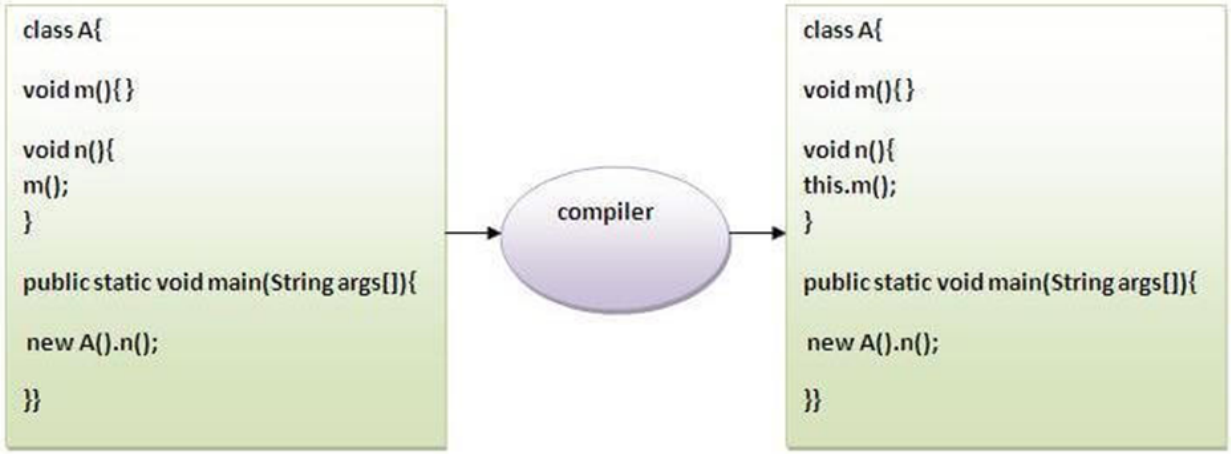
```
this.fee=fee;

}

void
display(){System.out.println(rollno+"
"+name+" "+fee);}

}


class TestThis2{

public static void main(String args[]){

Student                    s1=new
Student(111,"ankit",5000f);

Student                    s2=new
Student(112,"sumit",6000f);

s1.display();

s2.display();

}}
```

Output for both

```
111 ankit 5000.0
112 sumit 6000.0
```

2. **this can be used to invoke current class method (implicitly)**

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method.



Example:

```
class A{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
}
}
class TestThis4{
public static void main(String args[]){
A a=new A();
```

```
a.n();
}}
```

Output:

```
hello n
hello m
```

.

3. **this() can be used to invoke the current class constructor.**

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining.

Example:

```
class Student{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee){
this(rollno,name,course);//reusing constructor
```

```
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
}
class TestThis7{
public static void main(String args[]){
Student s1=new Student(111,"ankit","java");
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
s2.display();
}}
```

Output:

4. **this can be passed as an argument in the method call.**

   The this keyword can also be passed as an argument in the method. In event handling (or) in a situati The this keyword can also be passed as an argument in the method. on where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

   Example:

```
class S2{
```

```
void m(S2 obj){
System.out.println("method is invoked");
 }
void p(){
m(this);
}
public static void main(String args[]){
S2 s1 = new S2();
s1.p();
  }
}
```

Output:

```
method is invoked
```

5. **this can be passed as an argument in the constructor call.**
6. **this can be used to return the current class instance from the method.**

INHERITANCE

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

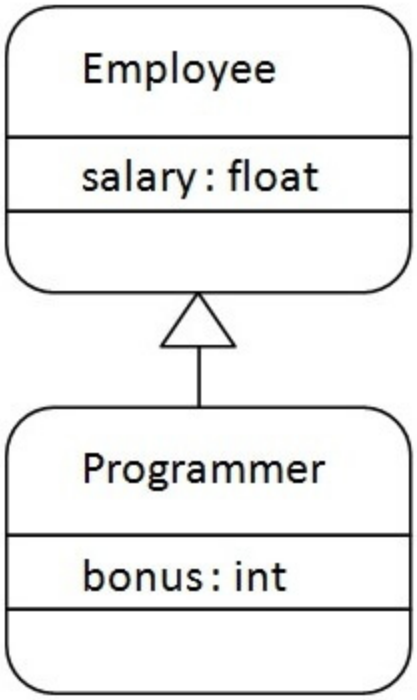
The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
```

```
  //methods and fields
}
```

The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality. In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is Programmer IS-A Employee. It means that Programmer is a type of Employee.

SAMPLE PROGRAM:

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

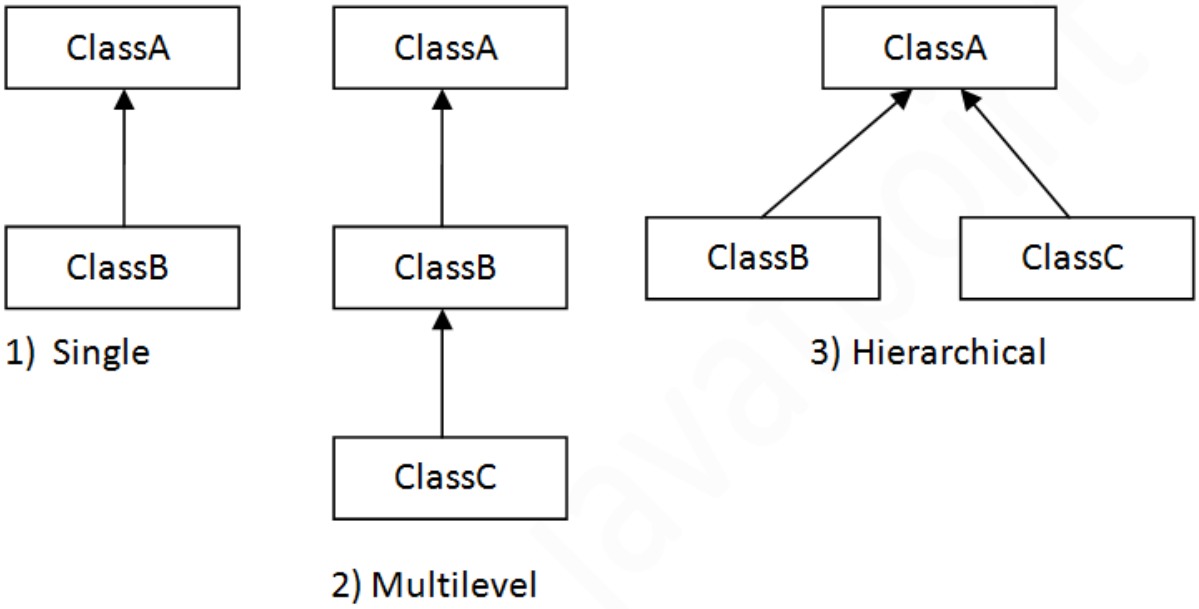
```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Types of inheritance in java

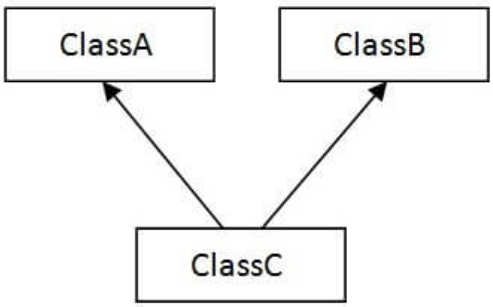
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only.

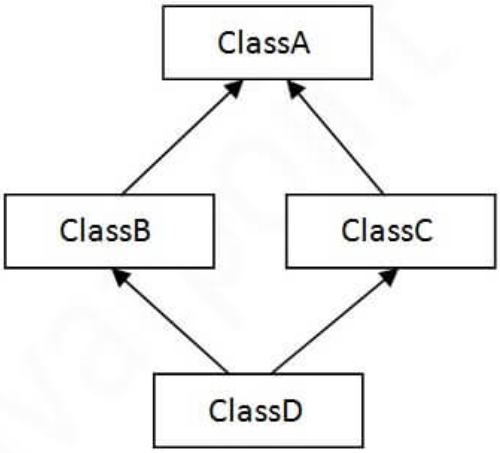


Multiple inheritance is not supported in Java through class. To reduce the complexity and simplify the language, multiple inheritance is not supported in java. Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



4) Multiple



5) Hybrid

.

ACCESS MODIFIERS IN JAVA

There are two types of modifiers in Java: access modifiers and non-access modifiers.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **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.

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. **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.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

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

1) Private

The private access modifier is accessible only within the class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

Program:

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

Program:

```
class A{
private A(){}//private constructor
void msg(){System.out.println("Hello java");}
}
public class Simple{
 public static void main(String args[]){
   A obj=new A();//Compile Time Error
 }
}
```

Note : A class cannot be private or protected except nested class.

2) Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

Programs:

```
//save by A.java
package pack;
class A{
  void msg(){System.out.println("Hello");}
```

```
}
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

---

3) Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifer.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

Programs:

```
//save by A.java
```

```
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

Output:

```
Output:Hello
```

4) Public

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

Programs:

```
//save by A.java
package pack;
public class A{
public void msg(){System.out.println("Hello");}
```

```
}
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Output:

```
Output:Hello
```

Java Access Modifiers with Method Overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

Program:

```
class A{
protected void msg(){System.out.println("Hello java");}
}


public class Simple extends A{
void msg(){System.out.println("Hello java");}//C.T.Error
 public static void main(String args[]){
   Simple obj=new Simple();
   obj.msg();
```

```
    }
}
```

The default modifier is more restrictive than protected. That is why, there is a compile-time error.

Method Overloading in Java

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

In Java, Method Overloading is not possible by changing the return type of the method only.

1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

Output :

```
22
33
```

2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

Output :

```
22
24.9
```

STATIC KEYWORD

The static keyword in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class.

The static can be:

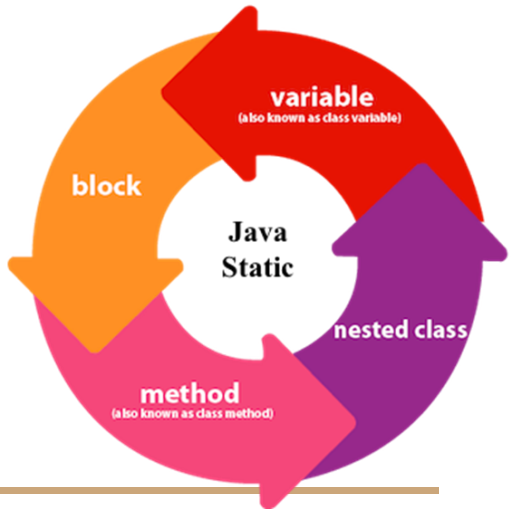
1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class

1) Java static variable

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Java static property is shared to all objects.



Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

Example 1 :

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.

Program:

```
//Java Program to demonstrate the use of static variable
class Student{
  int rollno;//instance variable
  String name;
  static String college ="ITS";//static variable
  //constructor
  Student(int r, String n){
  rollno = r;
  name = n;
  }
  //method to display the values
  void display (){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
public class TestStaticVariable1{
 public static void main(String args[]){
```

```
    Student s1 = new Student(111,"Karan");
    Student s2 = new Student(222,"Aryan");
    //we can change the college of all objects by the single line of code
    //Student.college="BBDIT";
    s1.display();
    s2.display();
    }
   }
```

Output:

```
111 Karan ITS
222 Aryan ITS
```

Program of the counter without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable. If it is incremented, it won't reflect other objects. So each object will have the value 1 in the count variable.

Example 2:

```
    //Java Program to demonstrate the use of an instance variable
    //which get memory each time when we create an object of the class.
    class Counter{
    int count=0;//will get memory each time when the instance is created
```

```
Counter(){
count++;//incrementing value
System.out.println(count);
}

public static void main(String args[]){
//Creating objects
Counter c1=new Counter();
Counter c2=new Counter();
Counter c3=new Counter();
}
}
```

OUTPUT:

Program of counter by static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
//Java Program to illustrate the use of static variable which
//is shared with all objects.
```

```
class Counter2{
static int count=0;//will get memory only once and retain its value

Counter2(){
count++;//incrementing the value of static variable
System.out.println(count);
}

public static void main(String args[]){
//creating objects
Counter2 c1=new Counter2();
Counter2 c2=new Counter2();
Counter2 c3=new Counter2();
}
}
```

OUTPUT:

```
1
2
3
```

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Example of static method

```
//Java Program to demonstrate the use of a static method.
class Student{
    int rollno;
    String name;
    static String college = "ITS";
    //static method to change the value of static variable
    static void change(){
    college = "BBDIT";
    }
    //constructor to initialize the variable
    Student(int r, String n){
    rollno = r;
    name = n;
    }
    //method to display values
    void display(){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to create and display the values of object
public class TestStaticMethod{
```

```
    public static void main(String args[]){
    Student.change();//calling change method
    //creating objects
    Student s1 = new Student(111,"Karan");
    Student s2 = new Student(222,"Aryan");
    Student s3 = new Student(333,"Sonoo");
    //calling display method
    s1.display();
    s2.display();
    s3.display();
    }
}
```

OUTPUT :

```
Output:111 Karan BBDIT
       222 Aryan BBDIT
       333 Sonoo BBDIT
```

Another example of a static method that performs a normal calculation

```
class Calculate{
  static int cube(int x){
  return x*x*x;
  }
```

```
  public static void main(String args[]){
  int result=Calculate.cube(5);
  System.out.println(result);
  }
}
```

Output :

```
Output:125
```

Restrictions for the static method

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

Program:

```
class A{
 int a=40;//non static

 public static void main(String args[]){
  System.out.println(a);
 } }
```

Output :

```
Output:Compile Time Error
```

3) Java static block

- Is used to initialize the static data member.
- It is executed before the main method at the time of classloading.

Example of static block

```
class A2{
  static{System.out.println("static block is invoked");}
  public static void main(String args[]){
   System.out.println("Hello main");
  }
}
```

Output :

```
Output:static block is invoked
       Hello main
```

FINAL KEYWORD

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method

3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

Program:

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class
```

OUTPUT :

```
Output:Compile Time Error
```

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

Program:

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda honda= new Honda();
  honda.run();
  }
}
```

Output :

```
Output:Compile Time Error
```

3)Java final class

If you make any class as final, you cannot extend it.

Example of final class

Program :

```
final class Bike{}
class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}
  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

Output :

```
Output:Compile Time Error
```

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

```
class A{
  static final int data;//static blank final variable
  static{ data=50;}
```

```
    public static void main(String args[]){
      System.out.println(A.data);
   }
}
```

Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

Program :

```
//Java Program to demonstrate why we need method overriding
//Here, we are calling the method of parent class with child
//class object.
//Creating a parent class
class Vehicle{
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike extends Vehicle{
  public static void main(String args[]){
  //creating an instance of child class
  Bike obj = new Bike();
  //calling the method with child class instance
  obj.run();
  }
}
```

Output :

```
Vehicle is running
```

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of

the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

Program :

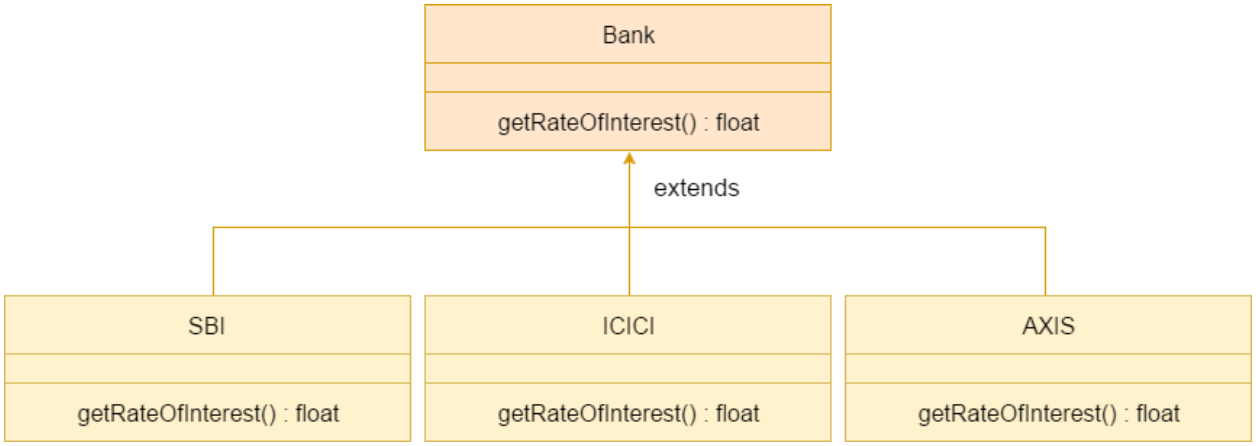
```
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Output :

```
Bike is running safely
```

A real example of Java Method Overriding

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.



Program :

```
//Java Program to demonstrate the real scenario of Java Method Overriding

//where three classes are overriding the method of a parent class.
//Creating a parent class.
class Bank{
int getRateOfInterest(){return 0;}
}
//Creating child classes.
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}
class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}
//Test class to create objects and call the methods
class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
```

```
        AXIS a=new AXIS();
        System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
        System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
        System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
        }
        }
```

Output :

```
Output:
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

Difference between method overloading and method overriding in java

There are many differences between method overloading and method overriding in java. A list of differences between method overloading and method overriding are given below:

Sno	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.

2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

SUPER KEYWORD

The super keyword in Java is a reference variable which is used to refer immediate parent class object. Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage:

1. **super can be used to refer immediate parent class instance variable.**

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

   Example:

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
```

```
}

class TestSuper1{

public static void main(String args[]){

Dog d=new Dog();

d.printColor();

}}
```

OUTPUT:

```
black
white
```

2. **super can be used to invoke immediate parent class method**

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

   Example:

```
class Animal{

void eat(){System.out.println("eating...");}

}

class Dog extends Animal{
```

```
void eat(){System.out.println("eating bread...");}

void bark(){System.out.println("barking...");}

void work(){

super.eat();

bark();

}

}

class TestSuper2{

public static void main(String args[]){

Dog d=new Dog();

d.work();

}}
```

OUTPUT:

```
eating...
barking...
```

3. **super() can be used to invoke immediate parent class constructor.**

   The super keyword can also be used to invoke the parent class constructor. super() is added in each class constructor automatically by compiler if there is no super() or this(). super() should also be the first line if used inside a constructor.

```
class Animal{

Animal(){System.out.println("animal is created");}

}

class Dog extends Animal{

Dog(){

super();

System.out.println("dog is created");

}

}

class TestSuper3{

public static void main(String args[]){

Dog d=new Dog();

}}
```

   OUTPUT:

```
animal is created
dog is created
```

Java Command Line Arguments

The java command-line argument is an argument i.e. passed at the time of running the java program.

The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behavior of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

Example 1:

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

compile by > javac CommandLineExample.java

run by > java CommandLineExample sonoo

Output :

```
Output: Your first argument is: sonoo
```

Example 2 :

```
class A{
public static void main(String args[]){
    for(int i=0;i<args.length;i++)
            System.out.println(args[i]);
}
}
```

compile by > javac A.java

run by > java A sonoo jaiswal 1 3 abc

Output :

```
Output: sonoo
        jaiswal
        1
        3
        abc
```

VARIABLE ARGUMENT (Varargs):

The varrags allows the method to accept zero or muliple arguments. Before varargs either we use overloaded method or take an array as the method parameter but it was not considered good because it leads to the maintenance problem. If we don't know how many argument we will have to pass in the method, varargs is the better approach.

Advantage of Varargs:

We don't have to provide overloaded methods so less code.

Syntax of varargs:

The varargs uses ellipsis i.e. three dots after the data type. Syntax is as follows:

```
return_type method_name(data_type... variableName){}
```

Rules for varargs:

While using the varargs, you must follow some rules otherwise program code won't compile. The rules are as follows:

- There can be only one variable argument in the method.
- Variable argument (varargs) must be the last argument.

Examples of varargs that fails to compile:

- `void method(String... a, int... b){}//Compile time error`
- `void method(int... a, String b){}//Compile time error`

Example:

```
class VarargsExample{
 static void display(String... values){
  System.out.println("display method invoked ");
  for(String s:values){
   System.out.println(s);
    }
}
```

```
 public static void main(String args[]){
 display();//zero argument
 display("hello");//one argument
 display("my","name","is","varargs");//four arguments
 }
}
```

Output :

```
Output:display method invoked
       display method invoked
       hello
       display method invoked
       my
       name
       is
       varargs
```