

# **JAVA PROGRAMMING LANGUAGE**

## **WHAT IS JAVA?**

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

## **REASON WE ARE GOING TO JAVA:**

- Simple in Nature.
- Flexible.
- Platform independent.
- Robust.
- Multi-Tasking
- Use Java for various application.

## **Types of Java Applications**

There are mainly 4 types of applications that can be created using Java programming:

### **1) Standalone Application**

Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone applications.

## **2) Web Application**

An application that runs on the server side and creates a dynamic page is called a web application. Currently, **Servlet**, **JSP**, **Struts**, **Spring**, **Hibernate**, **JSF**, etc. technologies are used for creating web applications in Java.

## **3) Enterprise Application**

An application that is distributed in nature, such as banking applications, etc. is called an enterprise application. It has advantages like high-level security, load balancing, and clustering. In Java, **EJB** is used for creating enterprise applications.

## **4) Mobile Application**

An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

## **Java Platforms :**

There are 4 platforms or editions of Java:

### **1) Java SE (Java Standard Edition)**

It is a Java programming platform. It includes Java programming APIs such as java.lang, java.io, java.net, java.util, java.sql, java.math etc. It includes core topics like OOPs, **String**, Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection, etc.

## **2) Java EE (Java Enterprise Edition)**

It is an enterprise platform that is mainly used to develop web and enterprise applications. It is built on top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, **JPA**, etc.

## **3) Java ME (Java Micro Edition)**

It is a micro platform that is dedicated to mobile applications.

## **4) JavaFX**

It is used to develop rich internet applications. It uses a lightweight user interface API.

## **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 has a large demand in the current job market
- It is easy to learn and simple to use
- It is open-source and free

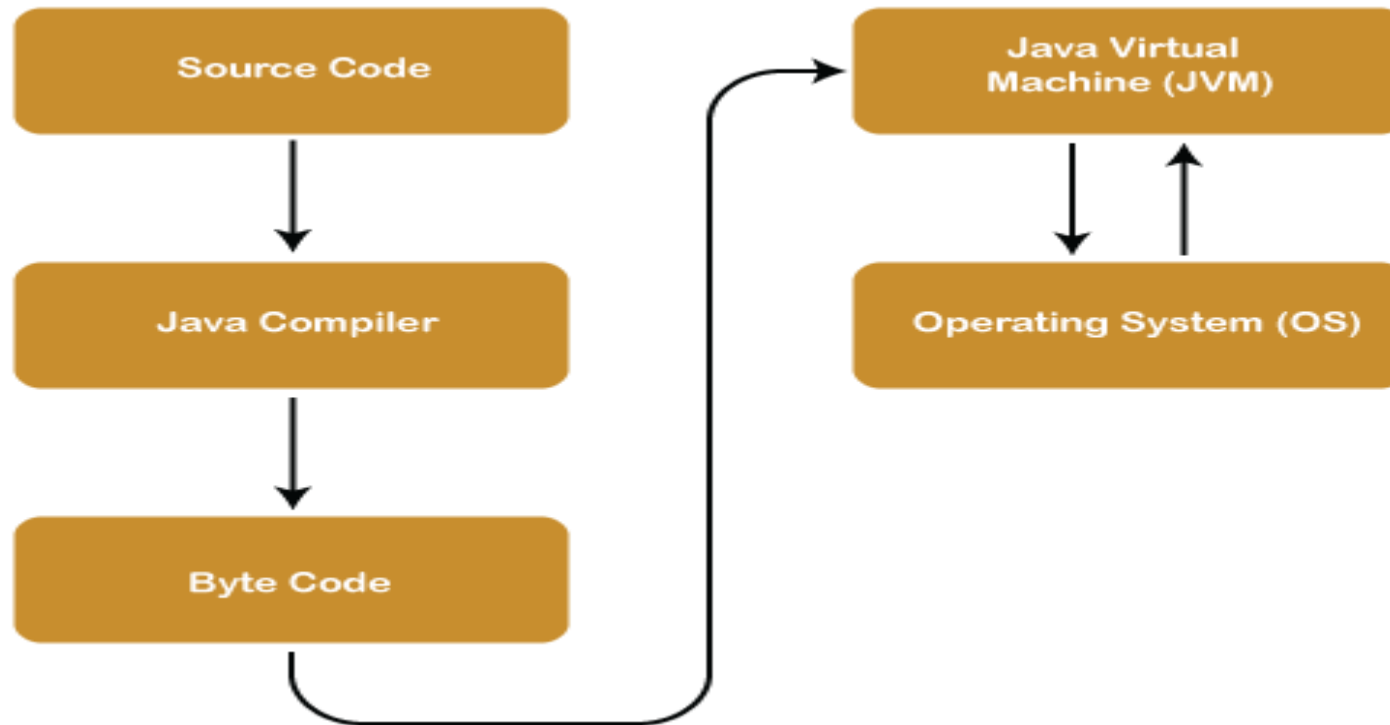
- It is secure, fast and powerful
- It has a huge community support (tens of millions of developers)
- Java is an object oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs
- As Java is close to [C++](#) and [C#](#), it makes it easy for programmers to switch to Java or vice versa

## **Java Architecture:**

**Java Architecture** is a collection of components, i.e., **JVM**, **JRE**, and **JDK**. It integrates the process of interpretation and compilation. It defines all the processes involved in creating a Java program. **Java Architecture** explains each and every step of how a program is compiled and executed.

**Java Architecture** can be explained by using the following steps:

- There is a process of compilation and interpretation in Java.
- Java compiler converts the Java code into byte code.
- After that, the JVM converts the byte code into machine code.
- The machine code is then executed by the machine.



### **Components of Java Architecture:**

The Java architecture includes the three main components:

- Java Virtual Machine (JVM)

- Java Runtime Environment (JRE)
- Java Development Kit (JDK)

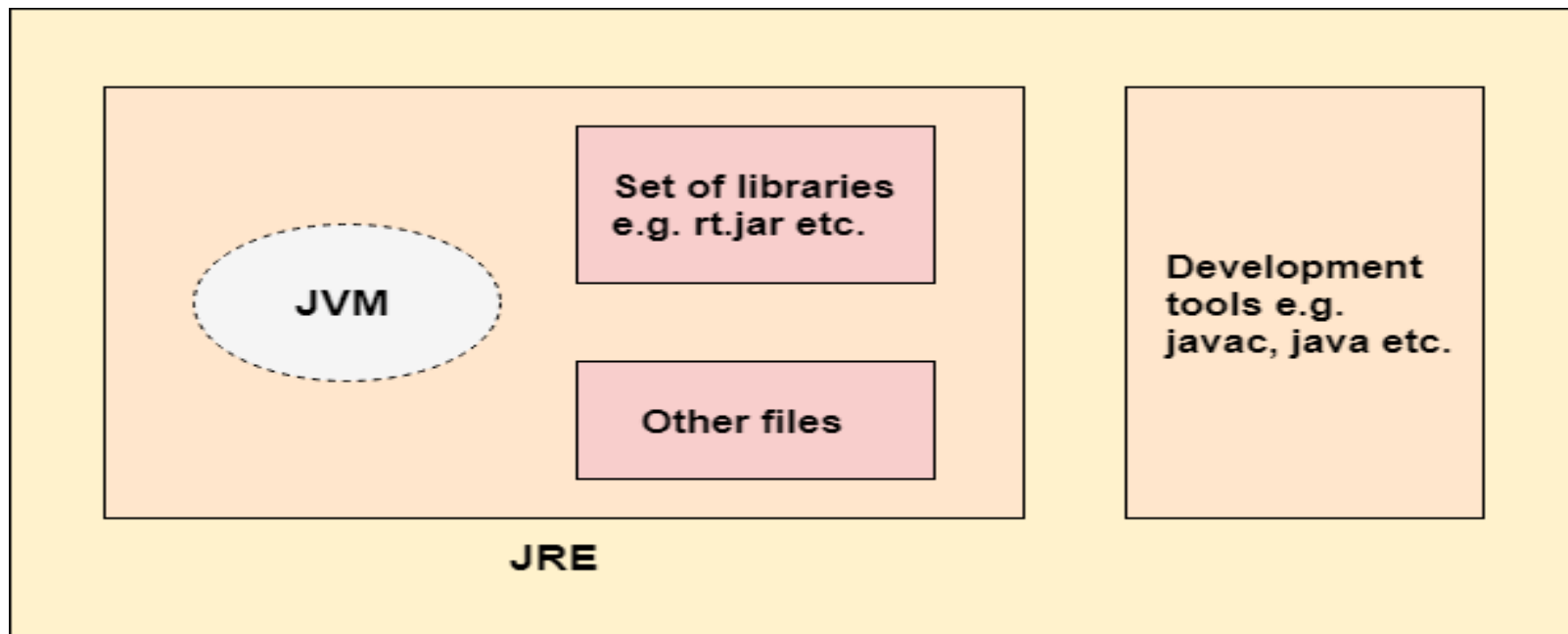
### **JDK(JAVA DEVELOPMENT KIT):**

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.



**JDK**

### **JVM(JAVA VIRTUAL MACHINE):**

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:

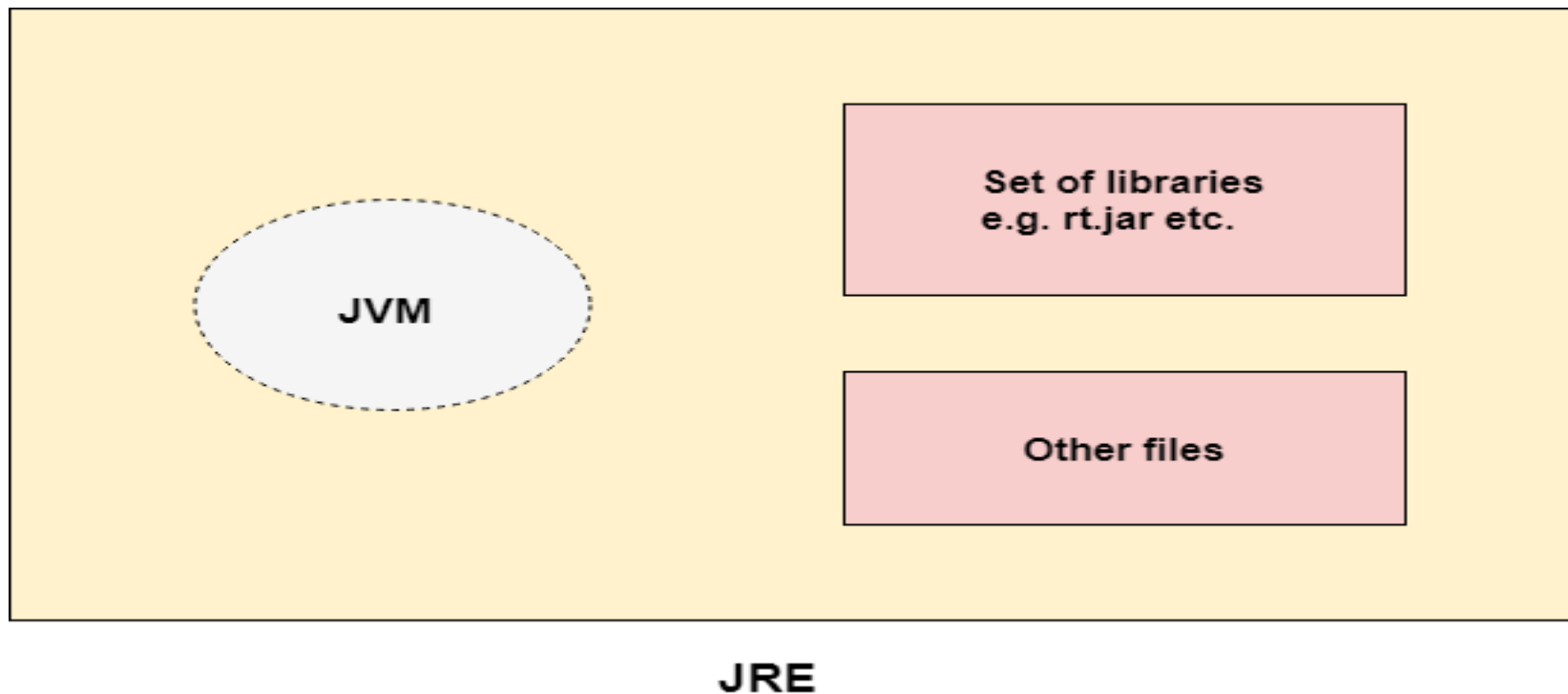
- Loads code
- Verifies code
- Executes code
- Provides runtime environment

### **JRE(JAVA RUNTIME ENVIRONMENT):**

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.



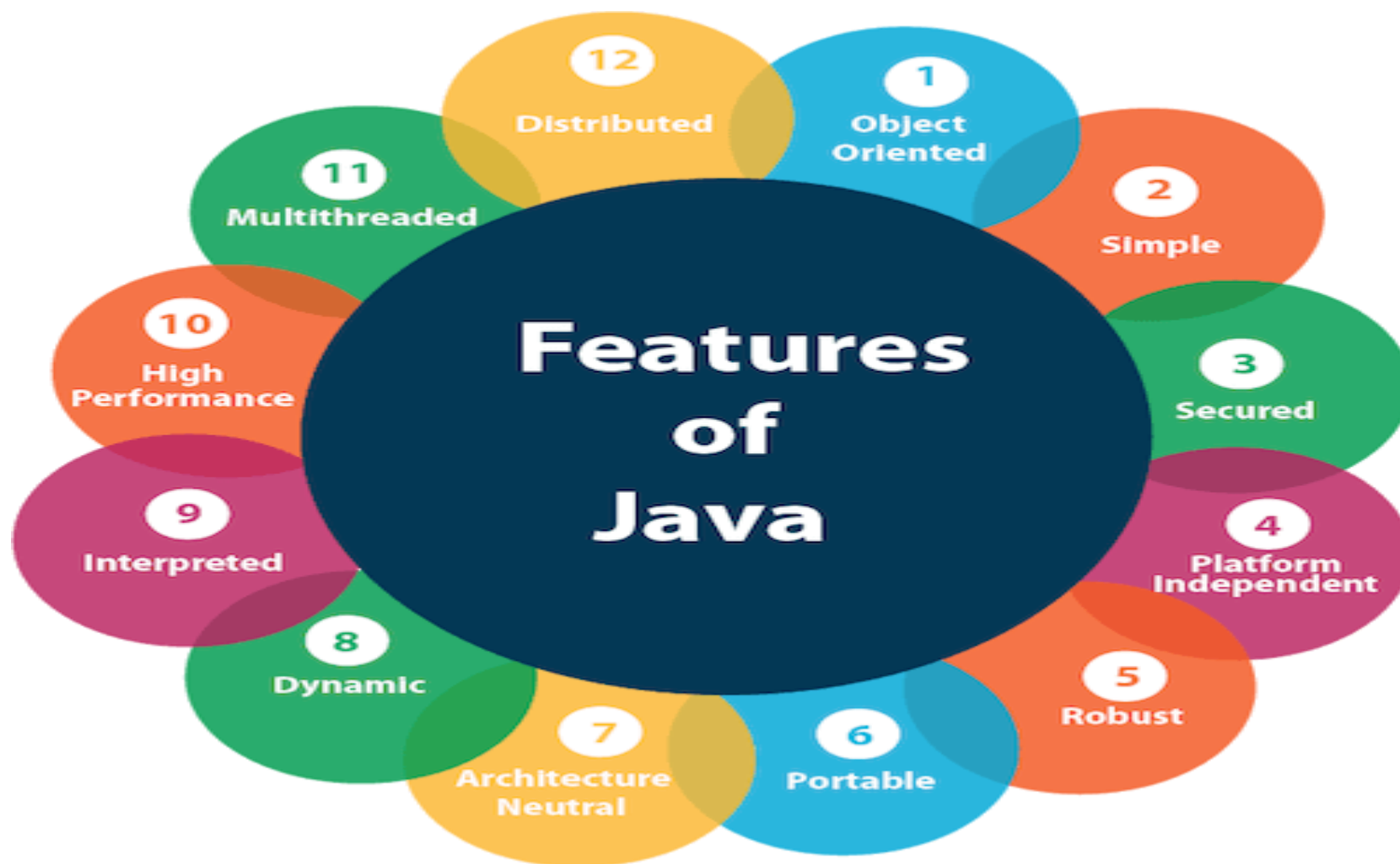


## FEATURES OF JAVA:

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

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

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



## Java Comments

The **Java** comments are the statements in a program that are not executed by the compiler and interpreter.

### Types of Java Comments

There are three types of comments in Java.

1. Single Line Comment
2. Multi Line Comment
3. Documentation Comment

#### 1) Java Single Line Comment

The single-line comment is used to comment only one line of the code. It is the widely used and easiest way of commenting the statements.

Single line comments starts with two forward slashes (**//**). Any text in front of **//** is not executed by Java.

#### 2) Java Multi Line Comment

The multi-line comment is used to comment multiple lines of code. It can be used to explain a complex code snippet or to comment multiple lines of code at a time (as it will be difficult to use single-line comments there).

Multi-line comments are placed between **/\*** and **\*/**. Any text between **/\*** and **\*/** is not executed by Java.

### 3) Java Documentation Comment

Documentation comments are usually used to write large programs for a project or software application as it helps to create documentation API. These APIs are needed for reference, i.e., which classes, methods, arguments, etc., are used in the code.

To create documentation API, we need to use the **javadoc tool**. The documentation comments are placed between `/**` and `*/`.

#### **FIRST PROGRAM:**

To create a simple Java program, you need to create a class that contains the main method. Let's understand the requirement first.

#### **The requirement for Java Hello World Example**

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

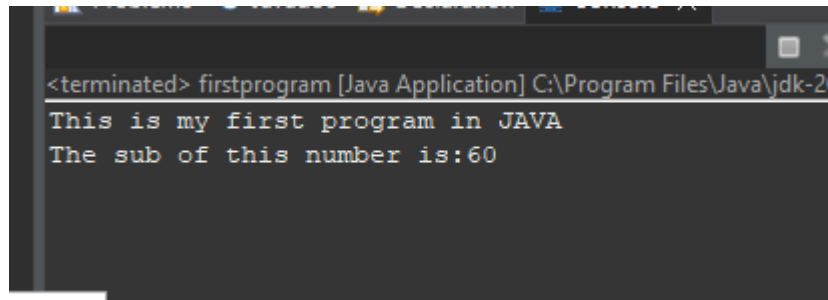
- Install the JDK if you don't have installed it, [download the JDK](#) and install it.
- Set path of the jdk/bin director.
- Create the Java program
- Compile and run the Java program

### **PROGRAM 1:**

```
package firstprogram;
```

```
public class firstprogram {  
    public static void main(String[] args) {  
        System.out.println("This is my first program in JAVA");  
        int a=75,b=15;  
        int c=a-b;  
        System.out.println("The sub of this number is:" + c);  
    }  
}
```

### **OUTPUT:**



```
<terminated> firstprogram [Java Application] C:\Program Files\Java\jdk-2
This is my first program in JAVA
The sub of this number is:60
```

## STATEMENT:

- First we have to create package, then create a class inside the package that you have created.
- Here I declare a package name as firstprogram and class name as firstprogram.
- The program should always start with main function (public static void main(String[] args).
- System.out.println is used to print the value in the screen that we are giving.
- Int for declaring a number.

## Data type:

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.

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

## **Boolean Data Type**

The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.

The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

### **Example:**

Boolean one = **false**



## Byte Data Type

The byte data type is an example of primitive data type. It is an 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

### Example:

**byte** a = 10, **byte** b = -20

## Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

### Example:

**short** s = 10000, **short** r = -5000

## Int Data Type

The int data type is a 32-bit signed two's complement integer. Its value-range lies between - 2,147,483,648 ( $-2^{31}$ ) to 2,147,483,647 ( $2^{31} - 1$ ) (inclusive). Its minimum value is - 2,147,483,648 and maximum value is 2,147,483,647. Its default value is 0.

The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

### **Example:**

```
int a = 100000, int b = -200000
```

### **Long Data Type**

The long data type is a 64-bit two's complement integer. Its value-range lies between - 9,223,372,036,854,775,808 ( $-2^{63}$ ) to 9,223,372,036,854,775,807 ( $2^{63} - 1$ ) (inclusive). Its minimum value is - 9,223,372,036,854,775,808 and maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

### **Example:**

```
long a = 100000L, long b = -200000L
```

## **Float Data Type**

The float data type is a single-precision 32-bit IEEE 754 floating point. Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

### **Example:**

```
float f1 = 234.5f
```

## **Double Data Type**

The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

### **Example:**

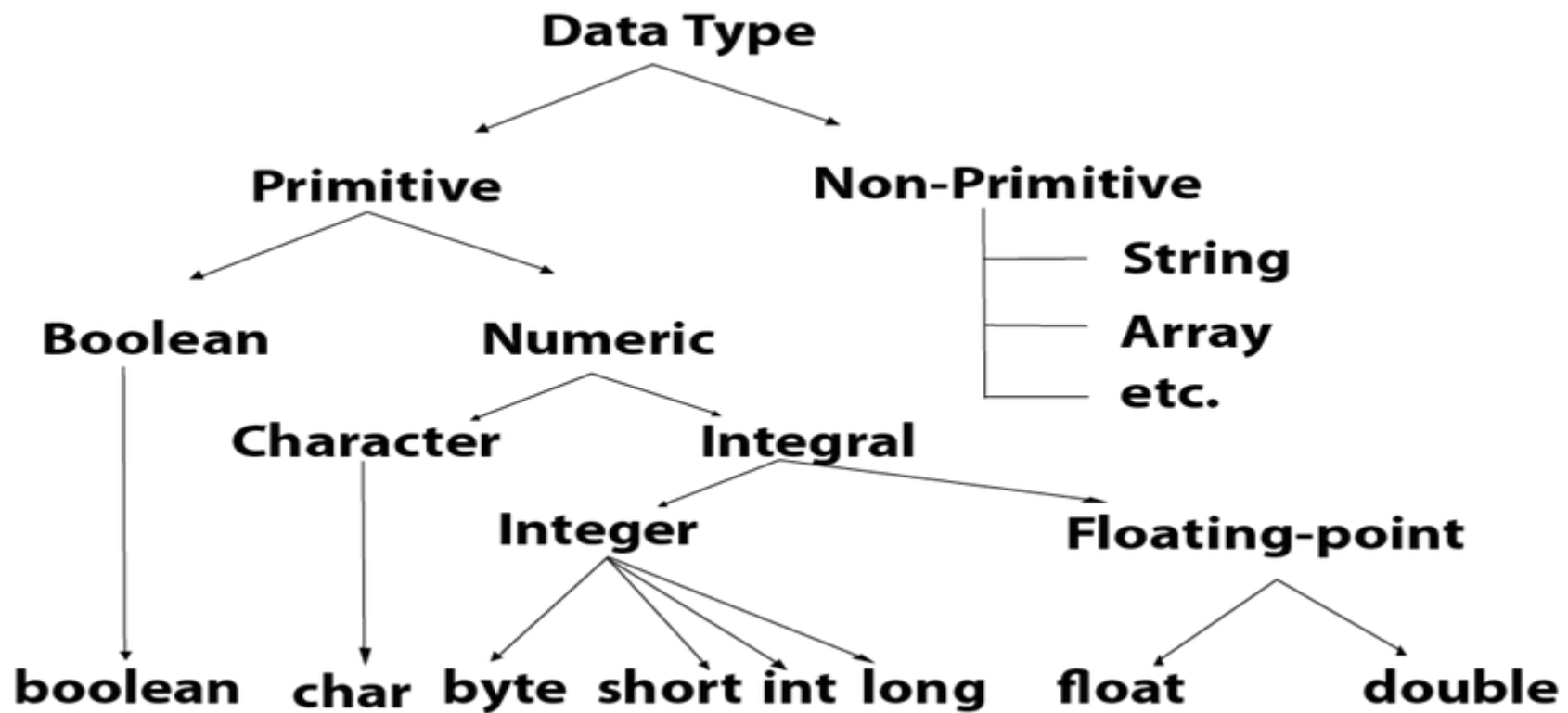
```
double d1 = 12.3
```

## **Char Data Type**

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive). The char data type is used to store characters.

**Example:**

**char** letterA = 'A'



**PROGRAM 2(DATA TYPE):**

```
package firstprogram;

public class datatype {
    public static void main(String[] args) {
        //STRING
        String studentName="Abirami";
        String lastName="Ramesh";
        System.out.println("the name of the student:" + studentName+lastName);
        //Integer
        int x=1500,y=250;
        int z=x*y;
        System.out.println("the multiplication of x and y is:" + z);
        /*final int
        final int value=26;
        value=66;
        System.out.println(value);*/
        //short
        short values;
        values=200;
        System.out.println("the short value is:"+values);
```

```
//long
long numbers=4356788975468L;
System.out.println("the long number is:"+numbers);
//double
double value=55.5d;
System.out.println("The double value is :"+value);
//float
float number=-5.5f;
System.out.println("the float number is:"+number);
//char
char firstLetter=66;
System.out.println("the character is:"+firstLetter);
//byte
byte num=50;
System.out.println("the value of byte is:"+num);
//boolean
boolean rainyDay=true;
boolean sunnyDay=false;
System.out.println("is it raning today"+rainyDay);
System.out.println("today is very sunny"+sunnyDay);
```

```
//declaring many variables
int ab=75; int bc=100; int ca=150;
System.out.println("The addition of the number is:"+(ab+bc+ca));
//one value to multiple variables
int a,b,c;
a=b=c=100;
System.out.println(a*b*c);

}
}
```



## OUTPUT:

```
terminated: datatype (java Application) C:\Program Files\Java\jdk-10.0.1\bin\java.exe
the name of the student:AbiramiRamesh
the multiplication of x and y is:37500
the short value is:200
the long number is:4356788975468
The double value is :55.5
the float number is:-5.5
the character is:B
the value of byte is:50
is it raining todaytrue
today is very sunnyfalse
The addition of the number is:325
1000000
```

## OPERATORS:

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

There are many types of operators in Java which are given below:

- Unary Operator,
- Arithmetic Operator,
- Shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

## **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 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.

## 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.

## PROGRAM 3(OPERATORS):

```
package firstprogram;
```

```
public class operators {
```

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

```
    //addition
```

```
    int add1=100,add2=add1+200,add3=add2+700;
```

```
    System.out.println(add1);
```

```
    System.out.println(add2);
```

```
    System.out.println(add3);
```

```
    System.out.println("=====");
```

```
//subtraction
int a=20000, b=50000, c=7500;
int d=a-b-c;
System.out.println("THE SUBTRACTION OF A,B,C IS:"+d);
System.out.println("=====");
//division
float x=150.7f,y=7;
System.out.println("THE DIVISION OF X,Y IS: "+(x/y));
System.out.println("=====");

//multiplication
int s=9,t=75,h=90;
System.out.println("THE MULTIPLICATION OF S,T,H IS: "+(s*t*h));
System.out.println("=====");
//modulo
int m=7,n=98;
System.out.println("THE MODULUS OF M,N IS: "+(m%n));
System.out.println("=====");
//increment
double g=27.5d, f=789.6d;
++g;
++f;
```

```
System.out.println("the increment is:"+f);
System.out.println("the increment is:"+g);
System.out.println("=====");
//decrement
float k=7.3f;
--k;
System.out.println("the decrement value is:"+k);

}
}
```

**OUTPUT:**

100  
300  
1000

=====

THE SUBTRACTION OF A,B,C IS:-37500

=====

THE DIVISION OF X,Y IS:21.52857

=====

THE MULTIPLICATION OF S,T,H IS:60750

=====

THE MODULUS OF M,N IS:7

=====

the increment is:790.6

the increment is:28.5

=====

the decrement value is:6.3

**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. There are also compound assignment operators in Java, which combine an arithmetic operation with assignment. These operators make it easier to perform an operation on a variable and then store the result back into the same variable.

**PROGRAM:**

```
package firstprogram;
```

```
public class assignmentoperator {  
    public static void main(String[] args) {  
        //increment  
        int x=6;  
        x+=4;  
        System.out.println("INCREMENT:"+x);  
        //decrement  
        int y=7;  
        y-=5;  
        System.out.println("DECREMENT:"+y);  
        //Multiply  
        int z=98;  
        z*=9;  
        System.out.println("MULTIPLY:"+z);  
        //DIVISION
```

```
float a=5.6f;
a/=2;
System.out.println("DIVISION:"+a);
//modulo
double b=50.5d;
b%=7;
System.out.println("MODULO:"+b);
//bitwise & operator
int d=767;
d&=4;
System.out.println("& OPERATOR:"+d);
// bitwise |= operator
int e=75;
e|=5;
System.out.println("|= operator:"+e);
//bitwise ^= operator
int g=6;
g^=3;
System.out.println("^= OPERATOR:"+g);
//bitwise right shift
int m=27;
m>>=3;
System.out.println("RIGHT SHIFT:"+m);
//bitwise left shift
int n=6;
```



```
n<<=2;  
System.out.println("LEFT SHIFT:"+n));  
//bitwise compliment  
int w=76,result;  
result= ~w;  
System.out.println("BITWISE COMPLIMENT:"+result);
```

```
}  
}
```

**OUTPUT:**

```
INCREMENT:10  
DECREMENT:2  
MULTIPLY:882  
DIVISION:2.8  
MODULO:1.5  
& OPERATOR:4  
|= operator:79  
^= OPERATOR:5  
RIGHT SHIFT:3  
LEFT SHIFT:24  
BITWISE COMPLIMENT:-77
```

## LOGICAL OPERATOR:

In Java, logical operators are used to perform logical operations on boolean values (true or false). These operators allow you to combine or manipulate boolean expressions to make more complex decisions in your code. Java provides three main logical operators: && (logical AND), || (logical OR), and ! (logical NOT).

### Logical AND (&&):

The logical AND operator (&&) returns true if both of its operands are true, otherwise, it returns false.

### Logical OR (||):

The logical OR operator (||) returns true if at least one of its operands is true, otherwise, it returns false.

**Logical NOT (!):**

The logical NOT operator (!) is a unary operator that reverses the value of its operand. If the operand is true, the NOT operator returns false, and if the operand is false, the NOT operator returns true.

**PROGRAM:**

```
package firstprogram;

public class logicaloperator {
public static void main(String[] args) {
    //AND OPERATOR
    int x=98,y=65;
    System.out.println("positive condition:" + (x>y&& y<x));
    System.out.println("negative condition:"+(x>y&& y>x));
    //OR operator
    double a=7.5d,b=4.5d;
    System.out.println("OR condition:"+ (a>b||b>a));
    System.out.println("OR false condition:"+ (a<b||b>a));
    //NOT operator
    float c=7.20f,d=6.70f;
    System.out.println("OR false condition:"+ !(c>d||c<d));
    System.out.println("OR true condition:"+ !(c>d&& c<d));
}
}
```

**OUTPUT:**

```
positive condition:true  
negative condition:false  
OR condition:true  
OR false condition:false  
OR false condition:false  
OR true condition:true
```

## **COMPARSION OPERATOR:**

In Java, comparison operators are used to compare values and determine the relationship between them. These operators return boolean values (true or false) based on the result of the comparison. Here are the comparison operators in Java:

### **Equal to (==):**

The equality operator (==) checks if the values of two operands are equal. If they are equal, it returns true; otherwise, it returns false.

### **Not equal to (!=):**

The inequality operator (!=) checks if the values of two operands are not equal. If they are not equal, it returns true; otherwise, it returns false.

### **Greater than (>):**

The greater than operator (>) checks if the value of the left operand is greater than the value of the right operand. If it is greater, it returns true; otherwise, it returns false.

### **Less than (<):**

The less than operator (<) checks if the value of the left operand is less than the value of the right operand. If it is less, it returns true; otherwise, it returns false.

**Greater than or equal to (>=):**

The greater than or equal to operator (>=) checks if the value of the left operand is greater than or equal to the value of the right operand. If it is greater than or equal, it returns true; otherwise, it returns false.

**Less than or equal to (<=):**

The less than or equal to operator (<=) checks if the value of the left operand is less than or equal to the value of the right operand. If it is less than or equal, it returns true; otherwise, it returns false.

These comparison operators are commonly used in conditional statements, loops, and decision-making structures to control the flow of a Java program based on the relationships between values.

**PROGRAM:**

```
package firstprogram;

public class comparisonoperator {
    public static void main(String[] args) {
        //EQUAL TO
        int x=76,y=76;
        System.out.println("EQUAL TO CONDITION:"+ (x==y));
        //NOT EQUAL
        System.out.println("NOTEQUAL TO CONDITION:"+ (x!=y));
        //GREATER THAN
        int a=250,b=76;
        System.out.println("GREATER THAN CONDITION:"+ (a>b));
    }
}
```

```
//less than
System.out.println("GREATER THAN CONDITION:"+ (a<b));
//greater than or equal to
float c=5.5f,d=2.5f;
System.out.println("GREATER THAN OR EQUAL TO:"+ (c>=d));
//less than or equal to
System.out.println("LESS THAN OR EQUAL TO:"+ (c<=d));
}
}
```

### OUTPUT:

```
EQUAL TO CONDITION:true
NOTEQUAL TO CONDITION:false
GREATER THAN CONDITION:true
GREATER THAN CONDITION:false
GREATER THAN OR EQUAL TO:true
LESS THAN OR EQUAL TO:false
```

### UNARY OPERATOR:

In Java, unary operators are operators that perform operations on a single operand. They are used to manipulate the value of a single variable or expression. Java supports several unary operators:

### **Unary Plus (+):**

The unary plus operator (+) doesn't change the sign of the operand. It's rarely used because it has no practical effect on most numeric values.

### **Unary Minus (-):**

The unary minus operator (-) negates the value of the operand, effectively changing the sign.

### **Increment (++):**

The increment operator (++) is used to increase the value of a variable by 1. It can be used as a prefix or a postfix operator.

### **Decrement (--):**

The decrement operator (--) is used to decrease the value of a variable by 1. Like the increment operator, it can be used as a prefix or a postfix operator.

**Logical NOT (!):**

The logical NOT operator (!) is a unary operator that reverses the value of a boolean expression. It converts true to false and false to true.

**Bitwise Complement (~):**

The bitwise complement operator (~) inverts the bits of an integer value, effectively changing all 0 bits to 1 and all 1 bits to 0.

These unary operators are essential for various programming tasks, including manipulating variables, performing calculations, and controlling flow based on conditions.

**PROGRAM:**

```
package firstprogram;

public class unaryoperator {
    public static void main(String[] args) {
        //positive
        int x=+5;
        System.out.println("POSITIVE:"+x);
        //Negative
        int y=-345;
        System.out.println("NEGATIVE:"+y);
    }
}
```



```
//INCREMENT
int z=675;
z++;
System.out.println("INCREMENT:"+z);
//ANOTHER METHOD
int h=245;
System.out.println("INCREMENTING:"+(++h));
//DECREMENT
int ab=789;
ab--;
System.out.println("DECREMENT:"+ab);
//another method
int cd=675;
System.out.println("DEREMENTING:"+(--cd));
}
}
```

### OUTPUT:

```
POSITIVE:5
NEGATIVE:-345
INCREMENT:676
INCREMENTING:246
DECREMENT:788
DEREMENTING:674
```

## **TERNARY OPERATOR:**

In Java, the ternary operator, also known as the conditional operator, is a shorthand way to write simple if-else statements. It allows you to make a decision between two values based on a condition. The syntax of the ternary operator is as follows

```
variable = (condition) ? expression1 : expression2;
```

Here, condition is evaluated. If it is true, expression1 is assigned to the variable; otherwise, expression2 is assigned.

## **PROGRAM:**

```
package firstprogram;

public class ternaryoperator {
    public static void main(String[] args) {
        int marks=55;
        String output=(marks>50)?"Passed.":"Failed";
        System.out.println(output);
    }
}
```

## **OUTPUT:**

```
Passed.
```

## **DECISION STATEMENT:**

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.

## **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.

## **Java Conditions and If Statements**

You already know that Java supports the usual logical conditions from mathematics:

- Less than:  $a < b$
- Less than or equal to:  $a \leq b$
- Greater than:  $a > b$
- Greater than or equal to:  $a \geq b$
- Equal to  $a == b$
- Not Equal to:  $a != b$

You can use these conditions to perform different actions for different decisions.

Java has the following conditional statements:

- Use if to specify a block of code to be executed, if a specified condition is true
- Use else to specify a block of code to be executed, if the same condition is false
- Use else if to specify a new condition to test, if the first condition is false
- Use switch to specify many alternative blocks of code to be executed

### **PROGRAM:**

```
package decisionstatement;

public class ifstatement {
    public static void main(String[] args) {
        //if statement
        int salary=15000;
        if(salary<=30000) {
            System.out.println("SALARY IS LESS THAN 30000");
        }
    }
}
```

### **OUTPUT:**

```
SALARY IS LESS THAN 30000
```

## **JAVA IF-ELSE:**

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.

In Java, the if-else statement is used for conditional branching in your code. It allows you to execute different blocks of code based on whether a certain condition is true or false. The basic syntax of the if-else statement is as follows:

```
if (condition) {  
    // Code to execute if the condition is true  
} else {  
    // Code to execute if the condition is false  
}
```

Remember that the if-else statement evaluates conditions sequentially, so once a true condition is found, the corresponding block of code is executed, and the rest of the if-else blocks are skipped. If none of the conditions is true, the code inside the else block (if present) will be executed.

The if-else statement is fundamental for controlling the flow of your program based on different conditions and is widely used for decision-making in Java programming.

### **PROGRAM:**

```
package decisionstatement;
```

```
public class ifelsestatement {  
    public static void main(String[] args) {  
        //if else  
        int age=30;  
        if(age>45) {  
            System.out.println("ELIGIBLE FOR LOAN");  
        }  
        else {  
            System.out.println("NOT ELIGIBLE FOR LOAN");  
        }  
    }  
}
```

### **OUTPUT:**

```
NOT ELIGIBLE FOR LOAN
```

### **JAVA IF...ELSE IF:**

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.

```
if (condition1) {  
    // Code to execute if condition1 is true  
} else if (condition2) {  
    // Code to execute if condition2 is true  
} else if (condition3) {  
    // Code to execute if condition3 is true  
} else {  
    // Code to execute if none of the conditions are true  
}
```

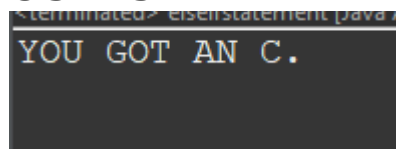
You can have any number of else if blocks in between the initial if and the final else block. The program will only execute the block associated with the first condition that evaluates to true. If none of the conditions is true, the code inside the else block (if present) will be executed.

### **PROGRAM:**

```
package decisionstatement;  
  
public class elseifstatement {  
    public static void main(String[] args) {  
        //if...else if  
        int marks=75;  
        if(marks>=90) {
```

```
System.out.println("YOU GOT AN A.");
}else if(marks>=80) {
    System.out.println("YOU GOT A B.");
}else if(marks>=70) {
    System.out.println("YOU GOT A C.");
}else if(marks>=60) {
    System.out.println("YOU GOT A D.");
}else {
    System.out.println("YOU GOT A F.");
}
}
}
```

### OUTPUT:



```
<terminated> - elselstatement.java
YOU GOT AN C.
```

### NESTED STATEMENT:

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

### PROGRAM:

```
package decisionstatement;
```



```
public class nestedstatement {  
    public static void main(String[] args) {  
        //nested statement  
        int age=25;  
        int salary=40000;  
        if(age>=18) {  
            System.out.println("You are an adult");  
            if(salary>30000) {  
                System.out.println("You can legal to get married");  
            }else {  
                System.out.println("You can illegal to get married");  
            }  
        }else {  
            System.out.println("YOU ARE A MINOR");  
        }  
    }  
}
```

### **OUTPUT:**

```
You are an adult  
You can legal to get married
```

## **WHILE LOOP:**

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**.

**The syntax of a while loop is as follows:**

java

Copy code

```
while (condition) {  
    // Code to execute as long as the condition is true  
}
```

It's important to ensure that the condition in a while loop eventually becomes false; otherwise, the loop will run indefinitely, causing a program to hang or crash.

The while loop is useful when you want to repeat an action based on a condition that can change during the loop's execution. Just make sure to update the variables inside the loop to ensure that the condition becomes false at some point and the loop terminates.

## **PROGRAM:**

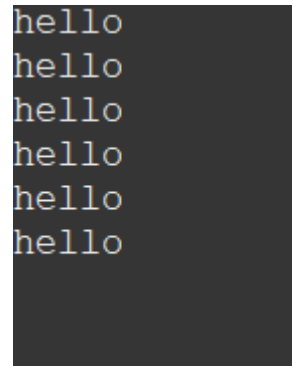
```
package whilestatement;
```

```
public class whileloop {  
    public static void main(String[] args) {  
        int i=0;  
        while(i<=5) {
```

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

```

### **OUTPUT:**



```
hello  
hello  
hello  
hello  
hello  
hello
```

### **DO WHILE LOOP:**

It's important to ensure that the condition in a while loop eventually becomes false; otherwise, the loop will run indefinitely, causing a program to hang or crash.

The while loop is useful when you want to repeat an action based on a condition that can change during the loop's execution. Just make sure to update the variables inside the loop to ensure that the condition becomes false at some point and the loop terminates.

```
do {  
    // Code to execute  
} while (condition);
```

The do-while loop is useful when you want to ensure that a certain block of code is executed at least once, even if the condition is false from the beginning. Just be cautious with the condition to prevent the loop from running indefinitely.

### **PROGRAM:**

```
package whilestatement;  
  
public class dowhile {  
    public static void main(String[] args) {  
        int i = 0;  
        do {  
            System.out.println(i);  
            i++;  
        }  
        while (i < 10);  
    }  
}
```

### **OUTPUT:**

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

## **BREAK :**

When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

The Java break statement is used to break loop or **switch** statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

## **CONTINUE:**

The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop.

The Java continue statement is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition. In case of an inner loop, it continues the inner loop only.

### **CONTINUE PROGRAM:**

```
package whilestatement;
```

```
public class breakloop {  
    public static void main(String[] args) {  
        //break  
        int i=5;  
        while(i<=20) {  
            System.out.println(i);  
            i++;  
            if(i==15) {  
                //break;  
                continue;  
            }  
        }  
    }  
}
```

### **OUTPUT:**

```
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20
```

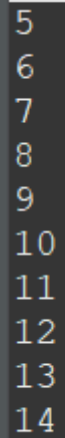
### **BREAK PROGRAM:**

```
package whilestatement;
```

```
public class breakloop {
```

```
public static void main(String[] args) {  
    //break  
    int i=5;  
    while(i<=20) {  
        System.out.println(i);  
        i++;  
        if(i==15) {  
            break;  
            //continue;  
        }  
    }  
}
```

### **OUTPUT:**



```
5  
6  
7  
8  
9  
10  
11  
12  
13  
14
```



## **FOR-LOOP INCREMENT:**

In Java, the for loop is a control flow statement that allows you to iterate over a sequence of values or perform a certain task a specific number of times. It's often used when you know the exact number of iterations you need. The for loop has a compact syntax that combines initialization, condition checking, and incrementing/decrementing in a single line. The basic syntax of a for loop is as follows:

```
for (initialization; condition; increment/decrement) {  
    // Code to execute in each iteration  
}
```

The for loop is versatile and widely used for various looping tasks in Java. It's particularly useful when you know the exact number of iterations you need or when iterating over arrays and collections.

## **PROGRAM:**

```
package whilestatement;  
  
public class forloop {  
    public static void main(String[] args) {  
        //for loop increment  
        for(int i=0;i<=20;i+=2) {  
            System.out.println(i);  
        }  
        System.out.println("GOOD MORNING");  
    }  
}
```

```
}  
}
```

### **OUTPUT:**

```
0  
2  
4  
6  
8  
10  
12  
14  
16  
18  
20  
GOOD MORNING
```

### **FOR-LOOP DECREMENT:**

#### **PROGRAM:**

```
package whilestatement;
```

```
public class forloopdec {  
    public static void main(String[] args) {  
        //for loop decrement
```

```
int i;  
for(i=30;i>=0;i-=3) {  
    System.out.println(i);  
}System.out.println(i);  
}
```

**OUTPUT:**

```
30  
27  
24  
21  
18  
15  
12  
9  
6  
3  
0  
-3
```

## **NESTED LOOP:**

A nested for loop in Java is a loop inside another loop. This is a technique used to iterate over multiple sets of data, typically in a two-dimensional structure like arrays or matrices. The syntax for a nested for loop is as follows:

```
java
Copy code
for (initialization; condition; increment/decrement) {
    // Outer loop code

    for (initialization; condition; increment/decrement) {
        // Inner loop code
    }
}
```

Nested loops can be nested further to handle more complex structures like three-dimensional arrays or matrices.

## **PROGRAM:**

```
package whilestatement;

public class nestedloop {
    public static void main(String[] args) {
        //nested loop
        for(int i=1;i<=5;i++){
            System.out.println("outside:"+""+i);
        }
    }
}
```

```
        //loop of j
        for(int j=1;j<=3;j++){
            System.out.println(i+" "+j);
        }
    }
}
```

**OUTPUT:**

outside:1

1 1

1 2

1 3

outside:2

2 1

2 2

2 3

outside:3

3 1

3 2

3 3

outside:4

4 1

4 2

4 3

outside:5

5 1

5 2

5 3

## **SWITCH STATEMENT:**

In Java, the switch statement is a control flow statement that allows you to compare the value of a variable against multiple possible cases and execute different blocks of code based on the matched case. The switch statement is useful when you have a single variable to check against multiple values. Here's the basic syntax of the switch statement:

```
switch (variable) {  
    case value1:  
        // Code to execute if variable matches value1  
        break;  
    case value2:  
        // Code to execute if variable matches value2  
        break;  
    // ... more cases ...  
    default:  
        // Code to execute if no case matches the variable  
}
```

A few key points to note about switch statements in Java:

Each case block should end with a break statement to prevent "fall-through" behavior, where subsequent cases are executed even if they don't match.

The default case is optional, and it's executed if none of the cases match the value of the variable.

The variable being tested in the switch statement must be of an integral type (byte, short, int, char) or an enumerated type.

Starting from Java 7, you can also use strings in the switch statement.

The switch statement provides a cleaner and more readable way to handle multiple cases compared to using multiple if-else statements.

### **PROGRAM:**

```
package switchstatement;
```

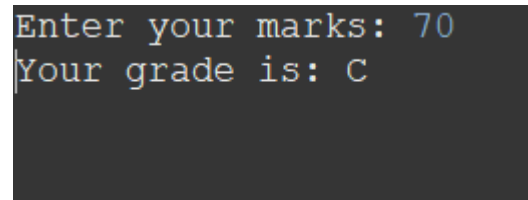
```
import java.util.Scanner;
```

```
public class switchloop {  
    public static void main(String[] args) {  
        //SWITCH STATEMENT  
        Scanner input = new Scanner(System.in);  
        System.out.print("Enter your marks: ");  
        int marks = input.nextInt();  
        char grade;  
        switch (marks) {  
            case 90:  
                grade = 'A';  
                break;  
            case 80:  
                grade = 'B';  
                break;  
        }  
    }  
}
```



```
        case 70:
            grade = 'C';
            break;
        case 60:
            grade = 'D';
            break;
        default:
            grade = 'F';
            break;
    }
    System.out.println("Your grade is: " + grade);
}
}
```

### **OUTPUT:**

A screenshot of a terminal window with a dark background. It shows the program's output: 'Enter your marks: 70' followed by 'Your grade is: C'.

```
Enter your marks: 70
Your grade is: C
```

### **SCANNER TO GET INPUT FROM USER:**

The Scanner class is used to get user input, and it is found in the java.util package.

To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the `nextLine()` method, which is used to read Strings:

### **PROGRAM:**

```
package switchstatement;
```

```
import java.util.Scanner;
```

```
public class scannerconsole {  
    public static void main(String[] args) {  
        Scanner in = new Scanner(System.in);  
        System.out.println("Enter your name:");  
        String name= in.next();  
        System.out.println("The name you have entered is:" + name);  
        //float number  
        Scanner out = new Scanner(System.in);  
        System.out.println("Enter a number:");  
        float num = out.nextFloat();  
        System.out.println("The number you have entered is float number:" + num);  
        //double number  
        Scanner a = new Scanner(System.in) ;  
        System.out.println("Enter you value:");  
        double val = a.nextDouble();  
    }  
}
```

```
        System.out.println("The value your entered is:"+val);  
    }  
}
```

### **OUTPUT:**

```
Enter your name:  
SHAN  
The name you have entered is:SHAN  
Enter a number:  
78  
The number you have entered is float  
Enter you value:  
678  
The value your entered is:678.0
```

### **ARRAY:**

In Java, an array is a data structure that allows you to store multiple values of the same data type under a single variable name. Arrays provide a way to manage and manipulate collections of data efficiently. Each individual value in an array is called an element, and each element is accessed using an index. The index starts from 0 for the first element and goes up to `array.length - 1` for the last element.

Arrays in Java are fixed in size once they are created. If you need a dynamic collection that can grow or shrink in size, you might want to consider using collections like ArrayList.

Arrays are an essential concept in programming and are used extensively to store and manipulate data in various applications.

### **PROGRAM:**

```
package arraypgm;

public class array1 {
    public static void main(String[] args) {
        //array
        String[] students = {
            "Hari", "Santhosh", "Abirami", "Divya", "Shanav"
        };
        System.out.println("indexof:");
        System.out.println(students[3]);
        System.out.println(students[0]);
        System.out.println("=====");
        //length
        System.out.println("length:");
        for( int i=0;i<students.length;i++) {
            System.out.println("Students name are:"+""+students[i]);
        }
    }
}
```

```
}  
}
```

## OUTPUT:

```
indexof:  
Divya  
Hari  
=====  
length:  
Students name are:Hari  
Students name are:Santhosh  
Students name are:Abirami  
Students name are:Divya  
Students name are:Shanav
```

## FOR-EACH LOOP:

In Java, the **for-each** loop is used to iterate through elements of [arrays](#) and collections (like [ArrayList](#)). In Java, the enhanced for loop, also known as the for-each loop, provides a convenient way to iterate over elements in an array, collection, or any object that implements the Iterable interface. It simplifies the process of iterating through the elements without needing to manage indices. The syntax of the enhanced for loop is as follows:

```
java  
Copy code  
for (elementType element : collection) {
```

```
// Code to process the element  
}
```

The enhanced for loop is especially useful when you want to iterate over all elements in a collection without worrying about indices or bounds. However, it's important to note that the enhanced for loop doesn't provide access to the current index, which might be needed in certain scenarios. For those situations, you would still use the traditional for loop with an index variable.

### **PROGRAM:**

```
package arraypgm;  
  
public class foreachloop {  
    public static void main(String[] args) {  
        //for each  
        int[] num = {75,90,76,45};  
        for(int items : num) {  
            System.out.println(items);  
        }  
    }  
}
```

### **OUTPUT:**

```
75  
90  
76  
45
```

## MULTIDIMENSIONAL ARRAY:

In Java, a multidimensional array is an array of arrays, or more generally, an array of arrays of arrays, and so on. This allows you to create structures that represent tables, matrices, or other multi-dimensional data. Common examples include 2D arrays, which are arrays organized into rows and columns, and 3D arrays, which can be thought of as arrays organized into multiple 2D arrays. Here's how you can declare and use multidimensional arrays:

To create a two-dimensional array, add each array within its own set of **curly braces**:

### Two-Dimensional (2D) Array:

A 2D array is an array of arrays, where each element of the main array is itself an array. It forms a grid-like structure with rows and columns.

### Three-Dimensional (3D) Array:

A 3D array is an array of 2D arrays, where each element of the main array is a 2D array.

However, as the number of dimensions increases, the complexity of managing and understanding the data also increases. In most cases, 2D arrays are the most commonly used for representing structured data like grids or matrices, while 3D arrays and beyond are used in more specialized applications.

### **PROGRAM USING INTEGER:**

```
package multidimensionalpgm;

public class multiloop {
public static void main(String[] args) {
    //loop
    int[][] grades= {{90,67,34},{87,96,56,78}};
    for(int i=0;i<grades.length;i++) {
    for(int j=0;j<grades[i].length;j++) {
    System.out.println(grades[i][j]);
    }
    }
}
```

### **OUTPUT:**



```
90  
67  
34  
87  
96  
56  
78
```

### **PROGRAM USING STRING:**

```
package multidimensionalpgm;
```

```
public class multipgm {  
    public static void main(String[] args) {  
        //multidimensional array  
        String[][]names= {"nandhini","sam","adam","miguel"  
}, {"hari","susi","ganesh","divya"};  
        System.out.println(names[1][0]);  
  
        //update  
        System.out.println(names[0][3]);  
        System.out.println("After updation:");  
        names[0][3]="victor";  
        System.out.println(names[0][3]);  
        System.out.println(names[1][5]);  
    }  
}
```

```
}  
}
```

## OUTPUT:

```
hari  
miguel  
After updation:  
victor  
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 4  
    at whatsapp/multidimensionalpgm.multipgm.main(multipgm.java:15)
```

## OOP CONCEPT:

OOP stands for **Object-Oriented Programming**.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

## Java Classes/Objects

Java is an object-oriented programming language.

Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects.

### Create a Class

To create a class, use the keyword **class**:

### PROGRAM:

#### CREATING SINGLE AND MULTIPLE OBJECT:

```
package ooppgm;
```

```
public class objectcode {  
    String name = "shanthini";//for single object  
    int a=10,b=27;  
    int c=a*b;//for multiple object1  
}
```

```
int x=7,y=76;  
int z=x*y;//for multiple object2  
//single object  
public static void main(String[] args) {  
    objectcode student = new objectcode();  
    System.out.println(student.name);  
    System.out.println("=====");  
    //Multiple object  
    objectcode math = new objectcode();  
    objectcode modulo = new objectcode();  
    System.out.println(math.c);  
    System.out.println(modulo.z);  
}  
}
```

### **OUTPUT:**

```
shanthini  
=====  
270  
7
```

## **MULTIPLE CLASS:**

In Java, you can create multiple classes within the same source file, but only one of those classes can be declared as public and its name must match the file name. The other classes can be non-public, and they are usually used for internal implementation details, helper classes, or related functionality. Let's take a look at an example:

Suppose you have a file named Main.java, and you want to define multiple classes within this file.

remember these points when working with multiple classes in a single file:

Only one class can be declared as public, and its name must match the file name. This class can be accessed from outside the package.

Other non-public classes in the same file can only be accessed within the same package.

You can have multiple non-public classes within the same source file.

Public classes must be defined in their own separate files with a name that matches the class name.

It's generally good practice to follow the principle of having one class per source file, especially for larger projects, as it helps maintain clarity and organization in your codebase.

## **PROGRAM:**

### **NORMAL CLASS WITH VARIABLE DECLARATION:**

```
package ooppgm;
```

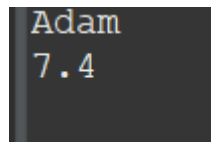
```
public class object2code {  
    String name = "Adam";  
}
```

## TO CALL THE ABOVE CLASS(object2code) VARIABLE IN ANOTHER CLASS(object3code)

```
package ooppgm;
```

```
public class object3code {  
    float num=7.4f;  
    public static void main(String[] args) {  
        object2code obj = new object2code();//calling class in another class  
        System.out.println(obj.name);  
        object3code obj1 = new object3code();//class within the class  
        System.out.println(obj1.num);  
    }  
}
```

### OUTPUT:



```
Adam  
7.4
```

### ATTRIBUTES:

In the previous chapter, we used the term "variable" for **x** in the example (as shown below). It is actually an **attribute** of the class. Or you could say that class attributes are variables within a class:

Attributes encapsulate the data within objects and enable you to work with object-oriented principles like encapsulation and abstraction. You can define access modifiers (such as public, private, protected, or default access) to control the visibility and accessibility of attributes to other classes.

Remember that attributes represent the state of an object, while methods represent the behavior. Together, they encapsulate the behavior and data of a class, allowing you to model real-world entities and their interactions in your Java programs.

### **PROGRAM:**

```
package ooppgm;

public class object4code {
    String name;//attribute value
    String student="Josh";//attributes override
    float a=7.8f;//final
    int x=30,y=27;
    int z=x/y;//multiple object1
    String trainer = "xyz";//multiple object2
    int b=89;
    String firstname = "Sri";
    String lastname = "Shanthini";
    int id = 1798544;// multiple attributes
    public static void main(String[] args) {
        //attribute
        object4code val = new object4code();
        val.name = "Adam";
        System.out.println(val.name);
        //override
    }
}
```

```
System.out.println("*****");
object4code a = new object4code();
a.student = "Victor";
System.out.println(a.student);
System.out.println("*****");
//final keyword
object4code max = new object4code();
max.a=29.9f;
System.out.println(max.a);
System.out.println("*****");
```

```
//multiple object
object4code div = new object4code();
object4code div2= new object4code();
object4code div3= new object4code();
div2.trainer="Avinesh";
div3.b=95;
System.out.println(div.z);//output 1
System.out.println(div2.trainer);// avinesh
System.out.println(div3.b);// update to 95
System.out.println("*****");
```

```
//multiple attributes
object4code name1 = new object4code();
System.out.println("Name:"+name1.firstname+" "+name1.lastname);
```



```
        System.out.println("id:"+name1.id);
    }
}
```

### **OUTPUT:**

```
Adam
*****
Victor
*****
29.9
*****
1
Avinesh
95
*****
Name:SriShanthini
id:1798544
```

### **STATIC AND NON-STATIC METHOD:**

Static methods are associated with the class itself rather than with instances of the class. They are declared using the static keyword. These methods are often used for utility functions, calculations, or operations that don't require access to instance-specific data. Static methods can be called directly using the class name without creating an instance of the class.

Non-static methods are associated with instances of the class. They are not declared as static. These methods can access and operate on instance variables and other instance-specific data. To call a non-static method, you need to create an instance of the class and call the method using that instance.

Static methods belong to the class itself and can be called using the class name.

Non-static methods belong to instances of the class and require an instance to be created before they can be called.

Choose between static and non-static methods based on whether the method's behavior or operation is specific to an instance or is general enough to be applicable to the entire class.

### **PROGRAM:**

```
package methodpgm;

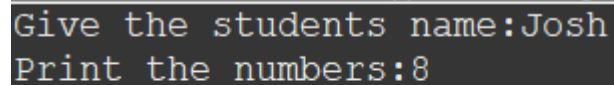
public class method1code {

    public static void name(){//static method
        String student = "Josh";
        System.out.println("Give the students name:"+""+student);
    }

    public void number() { //non-static method
        int a=89,b=9;
        int c=a%b;
        System.out.println("Print the numbers:"+""+c);
    }
}
```

```
public static void main(String[] args) {  
    name();  
    //object  
    method1code num = new method1code();//for calling non-static method  
    num.number();  
}  
}
```

### **OUTPUT:**



```
Give the students name:Josh  
Print the numbers:8
```

## **METHOD PASSING ARGUMENTS AND WITHOUT PASSING ARGUMENTS:**

### **PROGRAM:**

```
package methodpgm;  
  
public class method2code {  
    //method without passing an argument  
    public static void greet() {  
        System.out.println("Welcome all");  
    }  
    //method with passing arguments  
    public static void students(String sname,int age) {
```

```

    System.out.println("Welcome to today class"+" "+sname+" "+age);
}
//return by declaring data type
static int add(int a,int b) {
return 9+a+b;

}
//void without declaring data type
static void multi(int x,int y,int z) {
    int num = x*y*z;
    System.out.println("Te multiple of three numbers are:"+""+num);
}
//non-static method
public void sub(int h,int j) {
    int value = h - j;
    System.out.println("The sub of two numbers are:"+""+value);
}
//main function
public static void main(String[] args) {
    greet();
    students("Shanvika",23);
    students("Kajol",24);
    students("Lakshmipriya",23);
    students("Ruthra",24);
    students("Aishwaraya",24);
}

```

```
int c=add(9,7);
System.out.println("The sum of two numbers is:"+c);
multi(99,65,89);
//creating an object to call non static method
method2code numbers = new method2code();
numbers.sub(78,67);
multi(9,7,6);
}
}
```

### OUTPUT:

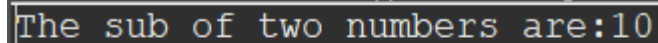
```
Welcome all
Welcome to today class Shanvika 23
Welcome to today class Kajol 24
Welcome to today class Lakshmipriya 23
Welcome to today class Ruthra 24
Welcome to today class Aishwaraya 24
The sum of two numbers is:25
Te multiple of three numbers are:572715
The sub of two numbers are:11
Te multiple of three numbers are:378
```

### CALLING NON-STATIC METHOD BY CREATING OBJECT IN ANOTHER CLASS FOR ABOVE PROGRAM:

```
package methodpgm;

public class secondcode {
    public static void main(String[] args) {
        method2code ab = new method2code();
        ab.sub(79,69);
    }
}
```

### OUTPUT:

A screenshot of a terminal window with a dark background. The text 'The sub of two numbers are:10' is displayed in a light-colored monospace font. A cursor is visible at the end of the line.

### METHOD-OVERLOADING:

Method overloading in Java is a feature that allows you to define multiple methods in a class with the same name but different parameter lists. These methods can have different numbers or types of parameters. Method overloading is a form of polymorphism and enables you to create more readable and flexible code by providing multiple ways to call a method based on different parameter requirements. The method that's executed depends on the arguments passed to it.

To perform method overloading, you need to follow these rules:

The methods must have the same name.

The methods must be in the same class.

The methods must have different parameter lists (number, type, or both).

that return type alone is not enough to distinguish between overloaded methods. Only the parameter lists are considered for method resolution.

### **PROGRAM:**

```
package methodoverloadingcode;
```

```
public class method1program {  
    public static int add(int a,int b) { //integer  
        return a+b;  
    }  
    public static float add(float x,float y) { //float values  
        return x*y;  
    }  
    public static double add(double a,double b,double c,double d) { //double values  
        return a-b-c-d;  
    }  
    public static void main(String[] args) {  
        int num1 = add(98,45);  
        float num2 = add(9.8f,89.9f);  
        double num3 = add(123.8d,67.9,90.7,76.6);  
    }  
}
```

```
System.out.println("the addition value of two numbers are:"+""+num1);  
System.out.println("the float value of two numbers are:"+""+num2);  
System.out.println("the double value of two numbers are:"+""+num3);  
}  
}
```

### **OUTPUT:**

```
the addition value of two numbers are:143  
the float value of two numbers are:881.02  
the double value of two numbers are:-111.4
```

### **CONSTRUCTOR :**

In Java, a constructor is a special type of method that is automatically called when an object of a class is created. It is used to initialize the object's state and perform any necessary setup. Constructors have the same name as the class and do not have a return type, not even void. Constructors can be overloaded, meaning you can define multiple constructors within the same class, each with a different set of parameters.

You can use constructors to perform various initialization tasks, such as setting default values, validating inputs, and establishing connections. Constructors play a crucial role in object-oriented programming as they ensure that objects are properly initialized and ready for use.



**PROGRAM:**

```
package methodoverloadingcode;

public class cons2code {
//default
    float a=8.7f;
    cons2code(){
    }
    //parameter passing
    public cons2code(String name,int age) {
        System.out.println("the name of the student is"+" "+name+" "+" "+"and is age"+" "+age);
    }

    public static void main(String[] args) {
        cons2code ab = new cons2code();
        System.out.println(ab.a);
        cons2code cd = new cons2code("Josh",23);

    }
}
```

**OUTPUT:**

8.7

```
the name of the student is Josh and is age23
```

### **WITH NO PASSING ARGUMENTS:**

```
package methodoverloadingcode;
```

```
public class constructorcode { //no arguments
    String name;
    public constructorcode() {
        name = "Victor";
    }
    public static void main(String[] args) {
        constructorcode ob = new constructorcode();
        System.out.println("My student name is:"+" "+ob.name);
    }
}
```

### **OUTPUT:**

```
My student name is: Victor
```

## **CONSTRUCTOR OVERLOADING:**

Constructor overloading in Java allows you to define multiple constructors in a class with different parameter lists. Each constructor can initialize the object's state in a different way, providing flexibility when creating objects. Constructor overloading follows the same rules as method overloading; constructors must have the same name but different parameter lists.

Depending on the arguments you provide when creating an object, the appropriate constructor will be called.

Constructor overloading provides a way to create objects with different initialization options and makes your classes more flexible and user-friendly.

## **PROGRAM:**

```
package methodoverloadingcode;

public class consoverloadcode {
    int code;
    String school;

    public consoverloadcode() { //default
        this.school="NRGHSS";
    }
    public consoverloadcode(String school) { //single argument
        this.school=school;
    }
    public consoverloadcode(String school,int code) { //two arguments
```

```

        this.school=school;
        this.code=code;
    }
    public void getName() {
        System.out.println(" Top schools:"+" "+this.school+this.code);
    }
    public static void main(String[] args) {
        consoverloadcode sch1 = new consoverloadcode();//default constructor
        consoverloadcode sch2 = new consoverloadcode("SRVS");//single argument constructor
        consoverloadcode sch3 = new consoverloadcode("SERVITE HIGH SCHOOL."+" "+
        "Code of the school"+"",78654);
        sch1.getName();
        sch2.getName();
        sch3.getName();
    }
}

```

## OUTPUT:

```

Top schools: NRGHSS0
Top schools: SRVS0
Top schools: SERVITE HIGH SCHOOL. Code of the school78654

```

## INHERITANCE:

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- **subclass** (child) - the class that inherits from another class
- **superclass** (parent) - the class being inherited from

To inherit from a class, use the **extends** keyword.

## SINGLE INHERITANCE:

In Java, single inheritance refers to the ability of a class to inherit properties and behavior from a single parent class. This means that a subclass can extend only one parent class. Java supports single inheritance to maintain simplicity, reduce ambiguity, and avoid the "diamond problem" that can arise in languages that support multiple inheritance.

Java allows you to build complex class hierarchies through single inheritance and composition (combining classes). If you need to share behavior between multiple classes that don't have a clear hierarchical relationship, you can use interfaces to achieve a form of multiple inheritance, where a class can implement multiple interfaces.

Remember that single inheritance simplifies the language and reduces potential conflicts that can arise from multiple inheritance, making Java's object-oriented programming model easier to understand and maintain.

**PROGRAM:**

```
package inheritanceprogram;
class single{//parent
    String student = "Adam";
    public void name() {
        System.out.println("students:"+""+student);
    }
}

class singlecode extends single{//child
    String student2="Josh";
    public void name1() {
        System.out.println("learner:"+""+student2);
    }
    public static void main(String[] args) {
        singlecode num = new singlecode();//method for calling parent or child
        num.name();
        num.name1();
    }
}
```

**OUTPUT:**

```
students:Adam
learner:Josh
```

## **MULTILEVEL INHERITANCE:**

Multilevel inheritance in Java refers to the situation where a subclass inherits properties and behavior from its parent class, which in turn inherits from another parent class. This creates a chain of inheritance where each class is both a subclass and a parent class simultaneously. Java supports multilevel inheritance, and it allows you to build more complex class hierarchies.

It's important to note that Java supports single inheritance, which means a class can only extend one parent class directly. However, through multilevel inheritance, you can still create complex class hierarchies by building chains of inheritance.

## **PROGRAM:**

```
package inheritanceprogram;
```

```
class person
{
public void present() { //parent class
    String name = "Sam";
    System.out.println("Present :"+""+name);
}
}
class absent extends person { //child class1
    public void absentees() {
        String name1 = "Ryan";
        System.out.println("Absent:"+""+name1);
    }
}
```

```
    }  
}  
//child class2  
class values extends absent {  
    public void total() {  
        int a=2;  
        System.out.println("students of total"+" "+a);  
    }  
}  
public class multilevelcode {  
    public static void main(String[] args) {  
        values val = new values();  
        val.present();  
        val.absentees();  
        val.total();  
    }  
}
```

### **OUTPUT:**

```
Present :Sam  
Absent:Ryan  
students of total 2
```



## **Hierarchical Inheritance:**

Hierarchical inheritance in Java refers to a situation where multiple subclasses inherit from a single parent class. In other words, the parent class serves as the root of a hierarchy, and multiple subclasses extend it. Each subclass inherits the properties and behavior of the parent class while also having the ability to define its own unique properties and behavior.

With hierarchical inheritance, you can create a structure in which different subclasses share common behavior from a single parent class while still having the ability to introduce their own specialized behavior.

It's worth noting that Java supports both hierarchical and multilevel inheritance, but not multiple inheritance (directly inheriting from multiple parent classes). Hierarchical inheritance is a simple and effective way to organize classes in a hierarchy with a single root class.

## **PROGRAM:**

```
package inheritanceprogram;
class vechile {
void vechiles()//parent class
{
    System.out.println("types of vechile");
}
}
class car extends vechile { //child 1
    void audi() {
        System.out.println("Audi is my favourite car");
    }
}
```

```

    }
}
class cycle extends vechile{//child 2
    void breeze() {
        System.out.println("Good for ladies");
    }
}
class bike extends vechile{//child 3
    void yamaha() {
        System.out.println("Yamaha is a race bike");
    }
}
public class herachaicalcode{
    public static void main(String[] args) {
        car num1 = new car();
        num1.audi();
        bike num2 = new bike();
        num2.yamaha();
        cycle num3= new cycle ();
        num3.breeze();
    }
}

```

**OUTPUT:**

```
Audi is my favourite car  
Yamaha is a race bike  
Good for ladies
```

## **ENCAPSULATION:**

Encapsulation is one of the four fundamental principles of object-oriented programming (OOP) and is a concept that focuses on bundling the data (attributes) and methods (functions) that operate on the data into a single unit, known as a class. It helps in controlling the access to the internal state of an object and prevents unauthorized external access or modification.

In Java, encapsulation is achieved through the use of access modifiers (private, protected, and public) and getter and setter methods. Here's how encapsulation works:

**Private Access Modifier:** By declaring attributes as private, you prevent direct external access to them. Private attributes can only be accessed within the same class.

**Public Access Modifier:** Methods that are used to interact with the attributes (getters and setters) are usually declared as public. This allows controlled access from outside the class.

### **Encapsulation provides several benefits:**

It prevents accidental modification of data by restricting access.

It allows controlled modification of data through setter methods.

It helps in maintaining consistent data by incorporating validation within setter methods.

It enables the ability to change the internal implementation of the class without affecting other parts of the code. Encapsulation is a crucial concept in OOP, promoting modularity and making it easier to manage and maintain code.

**PROGRAM:**

```
package encapsulationcode;

public class encap1 {
    private String bike;
    private int number;
    private double speed;
    private float Cubic;

    //set method
    public void setBikename(String name) {
        bike=name;
    }
    public void setId(int value) {
        number = value;
    }
    public void setSpeeds(double numb) {
        speed=numb;
    }
    public void setCapacity(float val) {
        Cubic = val;
    }
}
```

```
//get method
public String getBikename() { //string
    return bike;
}
public int getId() { //integer
    return number;
}
public double getKm() { //double
    return speed;
}
public float getCapacity() { //float
    return Cubic;
}
}
```

### **CALLING ABOVE ENCAPSULATION PROGRAM IN ANOTHER CLASS**

```
package encapsulationcode;

public class encap2 {
    public static void main(String[] args) {
        encap1 num = new encap1();
        num.setBikename("R15");
        num.setId(9551);
        num.setSpeeds(136);
        num.setCapacity(155.1f);
    }
}
```

```
System.out.println("Bike Name:"+num.getBikename());  
System.out.println("Bike Number:"+num.getId());  
System.out.println("The speed of R15 is:"+num.getKm()+"Kmph");  
System.out.println("The Cubic Capacity(CC) of R15 is:"+num.getCapacity()+" "+"CC");  
}  
}
```

### **OUTPUT:**

```
Bike Name:R15  
Bike Number:9551  
The speed of R15 is:136.0Kmph  
The Cubic Capacity(CC) of R15 is:155.1 CC
```

### **POLYMORPHISM:**

Polymorphism is a fundamental concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. It enables you to write code that can work with different types of objects in a consistent and flexible manner. Polymorphism is achieved through two main mechanisms: method overriding and method overloading.

#### **Method Overriding:**

Method overriding allows a subclass to provide a specific implementation for a method that is already defined in its superclass. The method in the subclass must have the same name, return

type, and parameter list as the method in the superclass. The @Override annotation is often used to indicate that a method is intended to override a superclass method.

Polymorphism simplifies code and makes it more flexible. It allows you to write generic code that can work with different types of objects without needing to know their specific classes.

### **PROGRAM:**

```
package polymorphismcode;
class books{
    void sub() {
        System.out.println("I am the book");
    }
}
class english extends books{
    void sub() {
        System.out.println("I am English book");
    }
}
class physics extends books{
    void sub() {
        System.out.println("I am Physics book");
    }
}
public class poly1 {
    public static void main(String[] args) {
        books myBook = new books();
    }
}
```

```
books myEnglish = new english();  
books myPhysics = new physics();  
myBook.sub();  
myEnglish.sub();  
myPhysics.sub();  
}  
}  
}
```

### **OUTPUT:**

```
I am the book  
I am English book  
I am Physics book
```

### **POLYMORPHISM OVERLOADING:**

Method overloading allows multiple methods in the same class to have the same name but different parameter lists. Java decides which method to call based on the arguments provided during the method call.

### **PROGRAM:**

```
package polymorphismcode;  
class overload{  
    int num1;
```



```
int num2;  
int result;  
void sum(int a,int b) {  
    num1=a;  
    num2=b;  
    result=num1+num2;  
    System.out.println("The int value :"+" "+result);  
}
```

```
void sum(int a,double b) {  
    num1=a;  
    num2=(int)b;  
    result=num1+num2;  
    System.out.println("The double and int value:"+" "+result);  
}
```

```
void sum(double a,double b) {  
    num1=(int)a;  
    num2=(int)b;  
    result=num1+num2;  
    System.out.println("The double value:"+" "+result);  
}
```

```
public class poly2code {  
    public static void main(String[] args) {  
        overload obj = new overload();  
        obj.sum(10,7);  
    }  
}
```

```
        obj.sum(6,20.3);  
        obj.sum(20.8,10.5 );  
    }  
}  
}
```

### **OUTPUT:**

```
The int value : 17  
The double and int value: 26  
The double value: 30
```

### **SUPER KEYWORD IN POLYMORPHISM:**

In Java, the super keyword is used to refer to the immediate parent class of a subclass. It is often used in the context of inheritance and polymorphism to access or call members (fields, methods, and constructors) of the superclass. The super keyword allows you to differentiate between members of the subclass and those of the superclass with the same name, or to call overridden methods in the superclass.

The super keyword is a useful tool in Java's inheritance mechanism, allowing you to work with both superclass and subclass members effectively, especially in cases of method overriding, constructor chaining, and hidden fields.

### **PROGRAM:**

```
package polymorphismcode;  
class humans{  
    public void people() {
```

```
        System.out.println("Shouting in the hall");
    }
}
class boys extends humans{
    public void people() {
        System.out.println("Boys");
        super.people();
    }
}
class girls extends humans{
    public void people() {
        super.people();
        System.out.println("Girls");
    }
}
public class poly3code {
    public static void main(String[] args) {
        humans num = new boys();
        humans num1 = new girls();
        num.people();
        num1.people();
    }
}
```

## OUTPUT:

```
Boys  
Shouting in the hall  
Shouting in the hall  
Girls
```

## ABSTRACTION:

Abstraction is one of the four fundamental principles of object-oriented programming (OOP). It involves the concept of representing essential features of an object while hiding the complex details. Abstraction allows you to focus on what an object does rather than how it does it. In Java, abstraction is achieved through abstract classes and interfaces.

### Abstract Classes:

An abstract class is a class that cannot be instantiated on its own and can have both abstract (unimplemented) and concrete (implemented) methods. Abstract methods are declared without a body and are meant to be implemented by subclasses. Abstract classes are used to define a common interface for a group of related classes.

abstraction allows you to define a common behavior (methods) that multiple classes can share, while the specific implementation details are left to the individual subclasses. Abstraction promotes code reusability, modular design, and separation of concerns, making your codebase easier to maintain and extend.

**PROGRAM:**

```
package abstractionpgm;
abstract class employee{
    public abstract void salary();//abstract method does not have body
    public abstract void bonus();

    public void credentials() {//regular method
        System.out.println("Normal method");
    }
}

class normal extends employee{//subclass inherits from employee
    public void salary() {
        System.out.println("Abstract method 1 salary in derived class");
    }
    public void bonus() {
        System.out.println("Abstract method 2 bonus in derived class");
    }
}

public class abs1code {
    public static void main(String[] args) {
        normal person = new normal();
        person.salary();
        person.bonus();
        person.credentials();
    }
}
```

```
}  
}
```

## OUTPUT:

```
Abstract method 1 salary in derived class  
Abstract method 2 bonus in derived class  
Normal method
```

## INTERFACE:

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

An interface is a collection of abstract methods that define a contract for classes implementing the interface. All methods in an interface are implicitly public and abstract. A class implementing an interface must provide implementations for all the methods declared in the interface.

The interface in Java is a mechanism to achieve **abstraction**. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple **inheritance in Java**.

**PROGRAM:**

```
package abstractionpgm;
interface fruits{//interface 1
    public void apples();//interface method 1
    public void blueberry();//interface does not have body
}
interface second{//interface 2
    public void raspberry();//interface method 2
}

class good implements fruits,second{
    public void apples() {
        System.out.println("Interface method 1");
    }
    public void raspberry() {
        System.out.println("Interface method 2");
    }
    public void blueberry() {
        System.out.println("Interface method");
    }
}

public class interfacecode {
    public static void main(String[] args) {
        fruits obj = new good();//object for interface 1
```

```
second obj1 = new good();//object for interface 2
obj.apples();
obj1.raspberry();
obj.blueberry();
}
}
```

### **OUTPUT:**

```
Interface method 1
Interface method 2
Interface method
```

### **EXCEPTION HANDLING:**

Exception handling in Java is a mechanism that allows you to handle errors and unexpected situations that can occur during the execution of a program. Exceptions are objects that represent errors, and they can occur for various reasons, such as invalid input, file not found, network issues, and more. By using exception handling, you can gracefully handle these errors and prevent your program from crashing.

#### **In Java, exceptions are divided into two categories:**

**Checked Exceptions:** These are exceptions that are checked at compile time. If a method can potentially throw a checked exception, the calling method must handle or declare that exception. Examples include `IOException`, `SQLException`, and `ClassNotFoundException`.



Unchecked Exceptions (RuntimeExceptions): These are exceptions that are not checked at compile time. They usually indicate programming errors or conditions that could not be foreseen. Examples include NullPointerException, ArrayIndexOutOfBoundsException, and ArithmeticException.

### **Key concepts in exception handling:**

try: The code within this block contains the statements that might cause an exception.

catch: The code within this block handles the exception. It is executed if an exception is thrown within the try block.

finally: This block is used to define code that should be executed regardless of whether an exception occurred or not. It's often used to release resources.

Java provides a wide range of built-in exceptions, but you can also create your own custom exceptions by extending the Exception class or one of its subclasses.

Exception handling helps your program recover gracefully from errors and improves the overall stability and user experience of your application.

### **PROGRAM:**

```
package exception;
```

```
public class excep1code {  
    public static void main(String[] args) {  
        try {  
            int[] num = {2,8,7,0};  
            int value = num[7];  
        }  
    }  
}
```

```
}catch(ArrayIndexOutOfBoundsException e) {  
    System.out.println("You Array is out of bounds");  
}catch(Exception e) {  
    System.out.println("Exception occurred in your code");  
}  
}  
}
```

### **OUTPUT:**

```
You Array is out of bounds
```

### **FINALLY KEYWORD:**

In Java, the finally keyword is used in conjunction with the try and catch blocks to define a block of code that will be executed regardless of whether an exception was thrown or not. This ensures that certain cleanup or resource release operations are performed regardless of whether an exception was caught or propagated.

The general structure of using the finally block is as follows:

```
try {  
    // Code that might throw an exception  
} catch (ExceptionType ex) {  
    // Code to handle the exception  
} finally {  
    // Code that will always be executed
```

```
}
```

Here's how the finally block works:

The code within the try block contains the statements that might throw an exception.

If an exception is thrown within the try block, the appropriate catch block (if available) is executed to handle the exception.

After the catch block (if present), the code within the finally block is executed, regardless of whether an exception occurred or not.

If no exception was thrown, the code within the finally block is executed after the try block completes.

Common use cases for the finally block include:

Releasing resources like file handles, database connections, or network sockets.

Closing streams or cleaning up after an operation.

Finalizing any pending operations that need to be done regardless of whether an exception was thrown.

Using the finally block ensures that critical cleanup tasks are performed, contributing to more reliable and robust code.

### **PROGRAM:**

```
package exception;
```

```
public class excep2 {  
    public static void main(String[] args) {  
        try {  
            int result=100/0;  
        }catch(ArithmeticException e) {
```

```
        System.out.println("Your Arithmetic expression is invalid");
    }
    finally {
        System.out.println("This is finally code block");
    }
}
}
```

### **OUTPUT:**

```
Your Arithmetic expression is invalid
This is finally code block
```

### **EXCEPTIONS:**

#### **PROGRAM:**

```
package exception;

public class excep3 {
    public static void main(String[] args) {
        try {
            String name = null;
```

```
        int length=name.length();
    }catch(NullPointerException e) {
System.out.println("Null pointer exception error occurred in your code");
    }try {
        String names = "welcome";
        char c = names.charAt(10);
    }catch(IndexOutOfBoundsException e) {
        System.out.println("Index out of bound");
    }
}
```

### **OUTPUT:**

```
Null pointer exception error occurred in your code
Index out of bound
```

### **THROW:**

In Java, the throw keyword is used to explicitly throw an exception. It allows you to signal that a certain condition or situation has occurred that requires special handling. When you throw an exception, you're indicating that the normal flow of the program cannot continue, and you're transferring control to an appropriate catch block or an exception handler up the call stack.

The syntax for using the throw keyword is as follows:  
throw exceptionObject;

### **Common scenarios for using the throw keyword include:**

Validating input and throwing an exception when the input doesn't meet certain criteria.

Indicating an error or exceptional condition that should be handled by the calling code.

Propagating exceptions up the call stack for handling at a higher level.

When you throw an exception, it's important to ensure that the exception is properly caught and handled by appropriate catch blocks or handled in an appropriate manner according to your application's requirements.

### **PROGRAM:**

```
package exception;
```

```
public class throw1 {  
    static void employee(int salary) {  
        if(salary<30000) {  
            throw new ArithmeticException("Your not eligible for applying to other company");  
        }else {  
            System.out.println("You eligible to apply");  
        }  
    }  
}  
public static void main(String[] args) {  
    employee(40000);  
}
```

```
}
```

### **OUTPUT:**

```
You eligible to apply
```

### **THROWS:**

In Java, the throws keyword is used in method declarations to indicate that the method might throw one or more types of exceptions. It provides information to the caller of the method about the potential exceptions that can occur during its execution. This allows the caller to handle these exceptions appropriately using try-catch blocks or propagate the exceptions further up the call stack using the throws keyword.

The throws keyword is part of a method's signature and is followed by the names of the exception classes that the method might throw. Multiple exception classes can be listed, separated by commas.

Syntax of using the throws keyword:

```
returnType methodName(parameters) throws ExceptionType1, ExceptionType2, ... {  
    // Method implementation  
}
```

The throws keyword is useful when you want to indicate to the caller of a method that certain exceptions might occur, allowing the caller to handle those exceptions appropriately. It's also useful in scenarios where you want to avoid catching exceptions within a method and instead propagate them to the calling code for handling.

### **PROGRAM:**

```
package exception;

import java.io.IOException;

class Myclass{
    void method()throws IOException{
        throw new IOException("error");
    }
}

public class excep4 {
    public static void main(String[] args) {
        try {
            Myclass file = new Myclass();
            file.method();
        }catch(Exception e) {
            System.out.println("Exception Handled");
        }
    }
}
```

### **OUTPUT:**



```
Exception Handled
```

## **COLLECTION IN JAVA:**

In Java, the Collections Framework provides a set of classes and interfaces to work with groups of objects as a single unit, making it easier to manage, manipulate, and store data. The framework offers various types of collections, such as lists, sets, maps, and queues, each designed to cater to specific use cases. The primary goal of the Collections Framework is to provide a standardized and efficient way to handle collections of objects.

### **Here are some key components of the Java Collections Framework:**

#### **Interfaces:**

**Collection:** The root interface for all collection classes. It defines the basic methods that all collections should have.

**List:** An ordered collection that allows duplicate elements. Common implementations include ArrayList and LinkedList.

**Set:** A collection that does not allow duplicate elements. Common implementations include HashSet and TreeSet.

**Map:** A collection that stores key-value pairs. Common implementations include HashMap and TreeMap.

**Queue:** A collection that represents a first-in-first-out (FIFO) queue. Common implementations include LinkedList and PriorityQueue.

**Classes:**

**ArrayList:** A resizable array-based implementation of the List interface.

**LinkedList:** A doubly-linked list implementation of the List and Queue interfaces.

**HashSet:** A hash-based implementation of the Set interface that does not allow duplicates.

**TreeSet:** A tree-based implementation of the Set interface that maintains elements in sorted order.

**HashMap:** A hash-based implementation of the Map interface that stores key-value pairs.

**TreeMap:** A tree-based implementation of the Map interface that maintains keys in sorted order.

**PriorityQueue:** An implementation of the Queue interface that stores elements based on their natural ordering or a provided comparator.

**Utility Classes:**

**Collections:** Provides various utility methods for working with collections, such as sorting, searching, and synchronization.

The Java Collections Framework simplifies working with collections of objects by providing standardized interfaces and implementations. This enables developers to write more efficient and maintainable code when dealing with data structures like lists, sets, maps, and queues.

## **ARRAYLIST:**

ArrayList is one of the most commonly used classes from the Java Collections Framework. It is part of the java.util package and provides a dynamic array implementation that can dynamically grow or shrink in size as needed. An ArrayList is essentially a resizable array, allowing you to store and manipulate a list of objects.

ArrayList provides a wide range of methods for managing the list, such as sorting, searching, adding, removing, and more. It's important to note that ArrayList can only store objects, not primitive types like int, double, etc. For primitive types, you can use their corresponding wrapper classes (e.g., Integer, Double) or use Java's autoboxing feature.

Keep in mind that while ArrayList provides dynamic resizing, it's important to choose the appropriate initial capacity for better performance if you have an estimate of the number of elements you'll be storing.

ArrayList is a versatile and powerful class that simplifies working with collections of objects, making it a fundamental tool in Java programming.

## **ARRAYLIST PROGRAM:**

```
package javaproject;
```

```
import java.util.ArrayList;
```

```
public class arrlistprogram {  
    public static void main(String[] args) {  
        ArrayList<String> students = new ArrayList<String>();  
        // adding items
```

```
students.add("Shanav");
students.add("Ruthra");
students.add("Lakshmi");
students.add("Aishwarya");
students.add("Kajol");
students.add("Abirami");
students.add("Sushil");
System.out.println(students);
System.out.println(students.get(5));
students.set(5, "Uma");//updating
System.out.println(students);
students.remove(6);//remove
System.out.println(students);
System.out.println("*****for-loop*****");
//loop through an arraylist
for(int i=0;i<students.size();i++) {
    System.out.println(students.get(i));
}
System.out.println("*****For-each loop*****");
//for each loop
for(String i:students) {
    System.out.println(i);
}
System.out.println("*****Arraylist using float datatype*****");
ArrayList<Float> num = new ArrayList<Float>();
```

```
num.add(10.8f);  
num.add(11.6f);  
num.add(34.5f);  
num.add(89.7f);  
num.add(56.2f);  
num.add(78.1f);  
for(int j=0;j<num.size();j++) {  
    System.out.println(num.get(j));  
}
```

```
}}
```

**OUTPUT:**

```
[Shanav, Ruthra, Lakshmi, Aishwarya, Kajol, Abirami, Sushil]
```

```
Abirami
```

```
[Shanav, Ruthra, Lakshmi, Aishwarya, Kajol, Uma, Sushil]
```

```
[Shanav, Ruthra, Lakshmi, Aishwarya, Kajol, Uma]
```

```
*****for-loop*****
```

```
Shanav
```

```
Ruthra
```

```
Lakshmi
```

```
Aishwarya
```

```
Kajol
```

```
Uma
```

```
*****For-each loop*****
```

```
Shanav
```

```
Ruthra
```

```
Lakshmi
```

```
Aishwarya
```

```
Kajol
```

```
Uma
```

```
*****Arraylist using float datatype*****
```

```
10.8
```

```
11.6
```

```
34.5
```

```
89.7
```

```
56.2
```

```
78.1
```

## **LINKEDLIST:**

LinkedList is another class from the Java Collections Framework that provides an implementation of a doubly-linked list. Unlike ArrayList, which is implemented as a dynamic array, LinkedList is implemented as a linked list data structure. It allows for efficient insertion and deletion operations in comparison to ArrayList, especially when elements are frequently added or removed from the middle of the list.

LinkedList provides fast insertions and deletions in comparison to ArrayList. However, accessing elements by index is generally slower due to the need to traverse the linked structure. It's important to choose the appropriate collection class based on the requirements of your specific use case.

LinkedList is a valuable option when you need efficient insertions and deletions, especially in scenarios where elements are frequently added or removed from the middle of the list.

## **LINKEDLIST PROGRAM:**

```
package javaproject;

import java.util.LinkedList;

public class linklistprogram {
    public static void main(String[] args) {
        LinkedList<String> names = new LinkedList<String>();
        names.add("Ruthra");
        names.add("Kajol");
        names.add("Lakshmi");
        names.add("Sri");
    }
}
```

```
System.out.println("My Friends Names are : "+ names.get(3));  
    names.addFirst("Sam");//add at first  
System.out.println(names);  
names.addLast("Josh");//add at last  
System.out.println(names);  
names.removeFirst();//remove first element  
System.out.println(names);  
names.removeLast();//remove last element  
System.out.println(names);  
System.out.println(names.getFirst());  
System.out.println(names.getLast());
```

```
}}
```



## OUTPUT:

```
My Friends Names are : Sri  
[Sam, Ruthra, Kajol, Lakshmi, Sri]  
[Sam, Ruthra, Kajol, Lakshmi, Sri, Josh]  
[Ruthra, Kajol, Lakshmi, Sri, Josh]  
[Ruthra, Kajol, Lakshmi, Sri]  
Ruthra  
Sri
```

## HASHMAP:

HashMap is a class from the Java Collections Framework that provides an implementation of the Map interface. It allows you to store key-value pairs and provides fast and efficient access to values based on their associated keys. HashMap is based on a hash table data structure, which enables constant-time average complexity for basic operations like put, get, and remove.

HashMap provides efficient access to values using keys and is a great choice when you need to store and retrieve key-value pairs quickly. It's important to note that HashMap allows duplicate values but not duplicate keys. If you need a map that maintains the order of key insertion, you can use the LinkedHashMap class.

HashMap is a versatile class that allows you to efficiently manage key-value pairs and provides a powerful tool for various applications, such as caching, indexing, and data retrieval.

## PROGRAM:

```
package javaproject;
```

```
import java.util.HashMap;

public class hashmapprogram {
    public static void main(String[] args) {
        HashMap<String,Integer> details = new HashMap<String,Integer>();
        details.put("Sam", 23);
        details.put("Adam",25);
        details.put("Cole",27);
        details.put("David",22);
        details.put("Josh",21);
        details.put("Ryan",24);
        System.out.println(details);
        System.out.println(details.get("Sam"));
        details.remove("David");
        System.out.println(details);

        System.out.println("*****for-each loop*****");
        for(String i:details.keySet()) {
            System.out.println(" Name: "+i+" Age: "+details.get(i));
        }
    }
}
```

**OUTPUT:**

```
{Adam=25, Ryan=24, Cole=27, David=22, Josh=21, Sam=23}
23
{Adam=25, Ryan=24, Cole=27, Josh=21, Sam=23}
*****for-each loop*****
Name: Adam Age: 25
Name: Ryan Age: 24
Name: Cole Age: 27
Name: Josh Age: 21
Name: Sam Age: 23
```

## **HASHSET:**

HashSet is a class from the Java Collections Framework that provides an implementation of the Set interface. It represents an unordered collection of unique elements. HashSet uses a hash table data structure to store elements, which allows for fast and efficient operations like add, contains, and remove. It's commonly used when you need to store a collection of distinct elements without any specific order.

HashSet ensures that each element is unique and enforces this uniqueness based on the elements' hash codes and equals method. The order of elements in a HashSet is not guaranteed to be in any particular sequence.

HashSet is a useful class when you need to store a collection of unique elements without caring about their order. It provides efficient operations for checking containment and adding/removing elements, making it a valuable tool for many different scenarios.

**The important points about Java HashSet class are:**

- HashSet stores the elements by using a mechanism called **hashing**.
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.
- The initial default capacity of HashSet is 16, and the load factor is 0.75.

**PROGRAM:**

```
package javaproject;
```

```
import java.util.HashSet;
```

```
public class hashsetprogram {  
    public static void main(String[] args) {  
        HashSet<String> employee = new HashSet<String>();  
        employee.add("Ram");  
        employee.add("Nathen");  
        employee.add("Camila");  
        employee.add("Sean");  
    }  
}
```

```
System.out.println(employee);  
System.out.println(employee.contains("Camila"));  
employee.remove("Nathen");  
System.out.println(employee);  
System.out.println("*****For-each loop*****");  
for(String i : employee) {  
    System.out.println(i);  
}  
}}
```

### **OUTPUT:**

```
[Camila, Nathen, Sean, Ram]  
true  
[Camila, Sean, Ram]  
*****For-each loop*****  
Camila  
Sean  
Ram
```

## **ITERATOR:**

An iterator in Java is an object that allows you to traverse through the elements of a collection, such as List, Set, or Map, and perform operations on each element without exposing the underlying implementation details of the collection. Iterators provide a way to sequentially access elements in a collection, making it easier to loop through and manipulate the collection's contents.

The Iterator interface is part of the Java Collections Framework and defines the methods that any iterator must implement. Here's a basic overview of how to use iterators in Java:

### **Obtain an Iterator:**

To obtain an iterator for a collection, you typically call the `iterator()` method on the collection object.

### **Use Iterator Methods:**

The Iterator interface provides several methods for iterating through a collection:

`hasNext()`: Returns true if there are more elements to iterate, false otherwise.

`next()`: Returns the next element in the collection and advances the iterator.

`remove()`: Removes the last element returned by `next()` from the underlying collection.

The use of iterators is not limited to just lists; you can also use them with other collection types such as sets and maps. Using iterators helps abstract away the details of how the collection is internally structured and allows you to focus on working with the elements.

Keep in mind that once you've used an iterator to iterate through a collection, you typically cannot reuse the same iterator to traverse the collection again. If you need to iterate through the collection multiple times, you'll need to obtain a new iterator.

**PROGRAM:**

```
package javaproject;
```

```
import java.util.ArrayList;
```

```
import java.util.Iterator;
```

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

```
ArrayList<String> person = new ArrayList<String>();  
    person.add("Josh");  
    person.add("Victor");  
    person.add("kygo");  
    person.add("Alessia");  
    Iterator<String> str = person.iterator();  
    System.out.println(str.next());  
    //loop through collection  
    while(str.hasNext()) {  
        System.out.println(str.next());  
    }  
    ArrayList<Integer> num = new ArrayList<Integer>();  
    num.add(10);  
    num.add(80);
```



```
num.add(25);  
num.add(203);  
Iterator<Integer> it = num.iterator();  
while(it.hasNext()) {  
    Integer i = it.next();  
    if(i < 20) {  
        it.remove();  
    }  
}  
System.out.println(num);  
}  
}
```

**OUTPUT:**

```
Josh  
Victor  
kygo  
Alessia  
[80, 25, 203]
```

## **JAVA QUEUE:**

In Java, a Queue is an interface from the Java Collections Framework that represents a collection designed for holding elements prior to processing. It follows the First-In-First-Out (FIFO) principle, meaning that the element that has been in the queue the longest will be the first one to be removed. The Queue interface provides methods for adding, removing, and examining elements in a queue.

### **Several classes implement the Queue interface, each with its own characteristics:**

**LinkedList:** LinkedList implements both the List and Queue interfaces. It can function as a queue with methods like add, remove, and peek.

**ArrayDeque:** ArrayDeque is a resizable-array implementation of the Deque interface (which extends the Queue interface). It can be used as a double-ended queue or as a queue. It is generally faster than LinkedList for queue operations.

**PriorityQueue:** PriorityQueue is a priority queue implementation, where elements are ordered based on their natural ordering or a provided comparator. It does not strictly adhere to the FIFO principle, as elements are ordered according to their priority.

the specific behavior of a queue can vary depending on the implementation class you use. If you need a regular FIFO queue, both LinkedList and ArrayDeque are good options. If you need priority-based ordering, then PriorityQueue is more appropriate.

### **PROGRAM:**

```
package javaproject;

import java.util.LinkedList;
import java.util.Queue;

public class queueprogram {

    public static void main(String[] args) {
        // Creating Queue using the LinkedList class
        Queue<Integer> numbers = new LinkedList<Integer>();

        // offer elements to the Queue
        numbers.offer(10);
        numbers.offer(25);
        numbers.offer(39);
        System.out.println("Queue: " + numbers);
    }
}
```

```
// Access elements of the Queue
int accessedNumber = numbers.peek();
System.out.println("Accessed Element: " + accessedNumber);

// Remove elements from the Queue
int removedNumber = numbers.poll();
System.out.println("Removed Element: " + removedNumber);

System.out.println("Updated Queue: " + numbers);
}}
```

### **OUTPUT:**

```
Queue: [10, 25, 39]
Accessed Element: 10
Removed Element: 10
Updated Queue: [25, 39]
```

## **PRIORITY QUEUE:**

In Java, a PriorityQueue is an implementation of the Queue interface that maintains its elements in a priority order. The priority of elements is determined either by their natural ordering (using their Comparable implementation) or by a provided comparator. The element with the highest priority is the one that will be removed first from the queue. In other words, elements with the highest priority "float" to the top of the queue.

PriorityQueue is a useful class for scenarios where you need to process elements based on their priority. Elements are maintained in the order defined by their priority, which can be natural ordering or based on a provided comparator. It's important to note that a PriorityQueue does not strictly adhere to the First-In-First-Out (FIFO) principle that regular queues follow.

PriorityQueue is particularly useful for applications involving task scheduling, shortest path algorithms, and any situation where elements need to be processed based on their relative priority.

## **PROGRAM:**

```
package javaproject;

import java.util.Iterator;
import java.util.PriorityQueue;

public class priorityqueuepgm {
    public static void main(String args[]){
        PriorityQueue<String> queue=new PriorityQueue<String>();
        queue.add("Amit");
        queue.add("Vijay");
```

```
queue.add("Karan");
queue.add("Jai");
queue.add("Rahul");
System.out.println("head:"+queue.element());
System.out.println("head:"+queue.peek());
System.out.println("iterating the queue elements:");
Iterator itr=queue.iterator();
while(itr.hasNext()){
    System.out.println(itr.next());
}
queue.remove();
queue.poll();
System.out.println("after removing two elements:");
Iterator<String> itr2=queue.iterator();
while(itr2.hasNext()){
    System.out.println(itr2.next());
}
}
}
```

**OUTPUT:**

```
head:Amit  
head:Amit  
iterating the queue elements:  
Amit  
Jai  
Karan  
Vijay  
Rahul  
after removing two elements:  
Karan  
Rahul  
Vijay
```