UNIT-1

INTRODUCTION TO OOP AND JAVA

Overview of OOP – Object oriented programming paradigms – Features of Object-Oriented Programming – Java Buzzwords – Overview of Java – Data Types, Variables and Arrays – Operators – Control Statements – Programming Structures in Java – Defining classes in Java – Constructors-Methods -Access specifiers – Static members- Java Doc comments

1.1 OVERVIEW OF OOP

1.1.1 What is OOP

- ✓ OOP (Object-oriented programming) is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).

1.1.1 Procedure Oriented Programming vs Object Oriented Programming

Procedure Oriented Programming	**Object Oriented Programming**
Program is divided into functions	Program is divided into classes and objects
It deals with algorithms	It deals with data
Data move from function to function	Functions that operate on data are bind to form classes
It is a Top-Down Approach	It is a Bottom-up approach
Do not have any specific access specifiers	It has access specifiers like public, private and protected
Less Secure	More Secure
It follows No overloading	It follows operator overloading and function overloading
Importance is not given to data but to functions as well as sequence of actions to be done	Importance is given to data rather than procedures
Does not provide any support for new data types	Provides support to new Data types
Poor Modeling to Real world problems	Strong Modeling to Real world problems
Its in not easy to maintain project if it is too complex	It's easy to maintain project even if it is too complex
Productivity is Low	Productivity is High
Example: C, VB, Fortran, Pascal	Example: C++, JAVA, Python

1.1.2 Basic Concepts of OOPS

✓ List of OOPs Concepts in Java

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

❖ **Object**

✓ Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

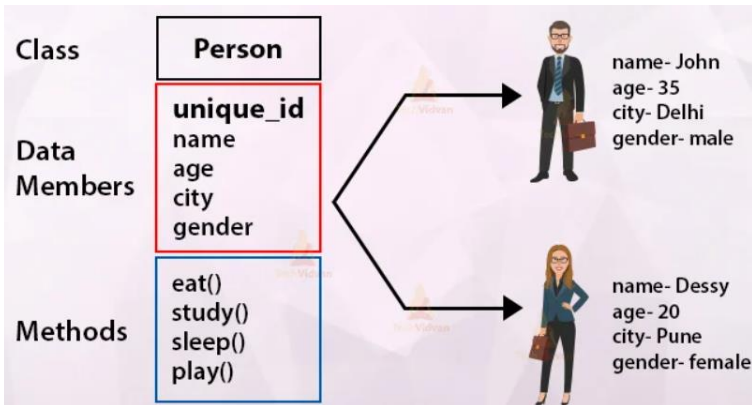
✓ Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

❖ **Class**

✓ Collection of objects is called class. It is a logical entity.

✓ A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

✓ Example: If you had a class called "Expensive Cars" it could have objects like Mercedes, BMW, Toyota, etc. Its properties(data) can be price or speed of these cars. While the methods may be performed with these cars are driving, reverse, braking etc.



❖ **Inheritance**

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

✓ Java supports the following four types of inheritance:

- Single Inheritance
- Multi-level Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

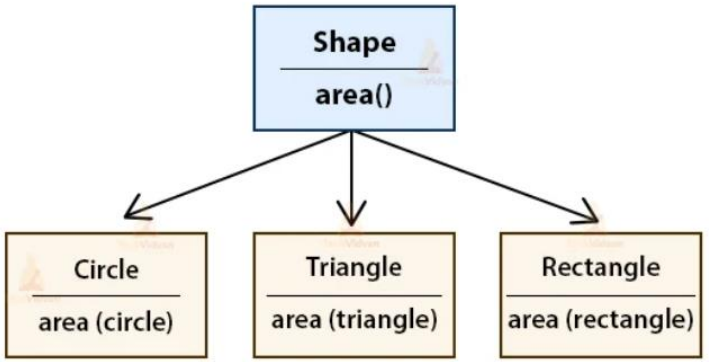
❖ **Polymorphism**

✓ If one task is performed in different ways, it is known as polymorphism.

✓ For example, a cat speaks meow, dog barks woof, etc.

✓ To draw something, for example, shape, triangle, rectangle, etc.

✓ In Java, we use method overloading and method overriding to achieve polymorphism.



❖ **Abstraction**

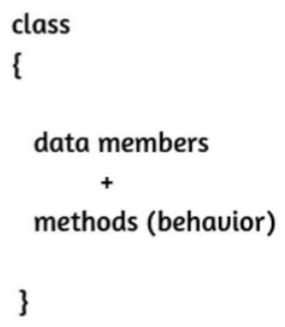
✓ Hiding internal details and showing functionality is known as abstraction.

✓ For example, while driving a car, you do not have to be concerned with its internal working. Here you just need to concern about parts like steering wheel, Gears, accelerator, etc.

✓ In Java, we use abstract class and interface to achieve abstraction.

❖ **Encapsulation**

- ✓ Binding (or wrapping) code and data together into a single unit are known as encapsulation.
- ✓ For example, a capsule, it is wrapped with different medicines.
- ✓ A java class is the example of encapsulation.



```
class
{

  data members
        +
  methods (behavior)

}
```

1.2 Features of Object-Oriented Programming or JAVA Buzzwords

- ✓ The Java programming language is a high-level language that can be characterized by all the following buzzwords:
  - Simple
  - Object-oriented
  - Distributed
  - Interpreted
  - Robust
  - Secure
  - Architecture neutral
  - Portable
  - High performance
  - Multithreaded
  - Dynamic

❖ **Simple**

- ✓ Java was designed to be easy for a professional programmer to learn and use effectively.
- ✓ It's simple and easy to learn if you already know the basic concepts of Object Oriented Programming.

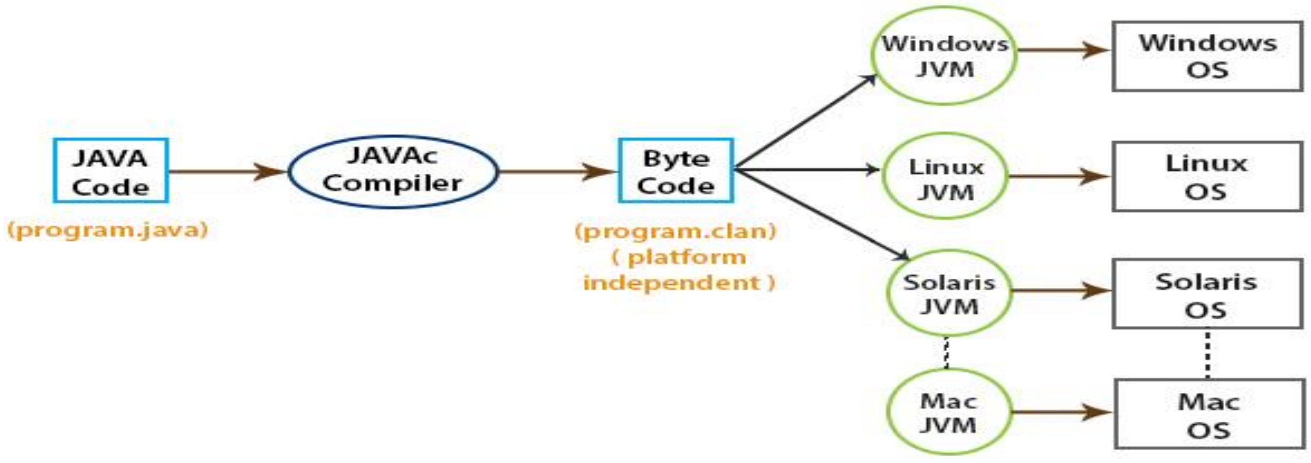
- ✓ Java has removed many complicated and rarely used features, for example, explicit pointers, operator overloading, etc.

❖ **Object Oriented**

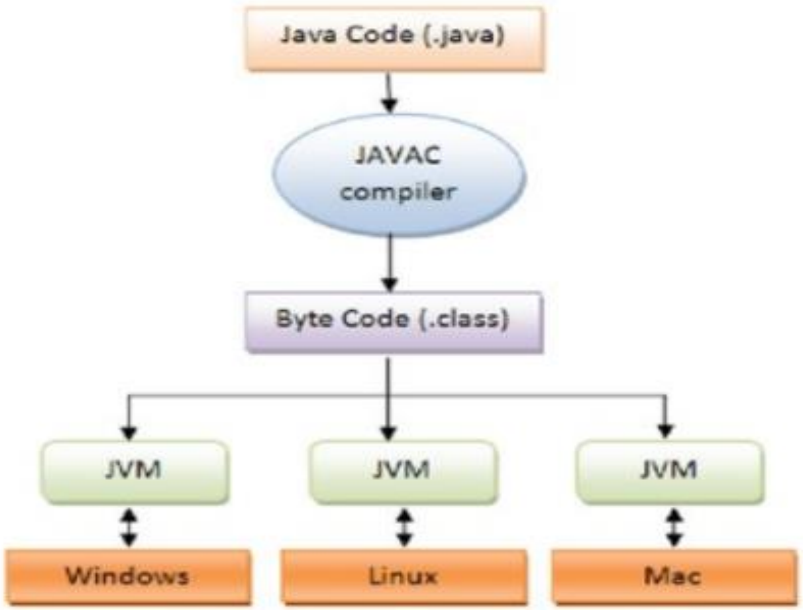
- ✓ Java is true object-oriented language.
- ✓ Almost "Everything is an Object"
- ✓ The object model in Java is simple and easy to extend.
- ✓ Basic concepts of OOPs are:
    - Object
    - Class
    - Inheritance
    - Polymorphism
    - Abstraction
    - Encapsulation

❖ **Platform Independent**

- ✓ Platform-independent means a program compiled on one machine can be executed on any machine in the world without any change. Java achieves platform independence by using the concept of the BYTE code.
- ✓ The Java Compiler converts the source code into an intermediate code called the byte code and this byte code is further translated to machine-dependent form by another layer of software called JVM (Java Virtual Machine).



- **Architecture-neutral**
  - ✓ The java byte code is independent of the underlying platform that the program is running on. For example, it doesn't matter if your operating system is 32-bit or 64-bit, the Java byte code is exactly the same.

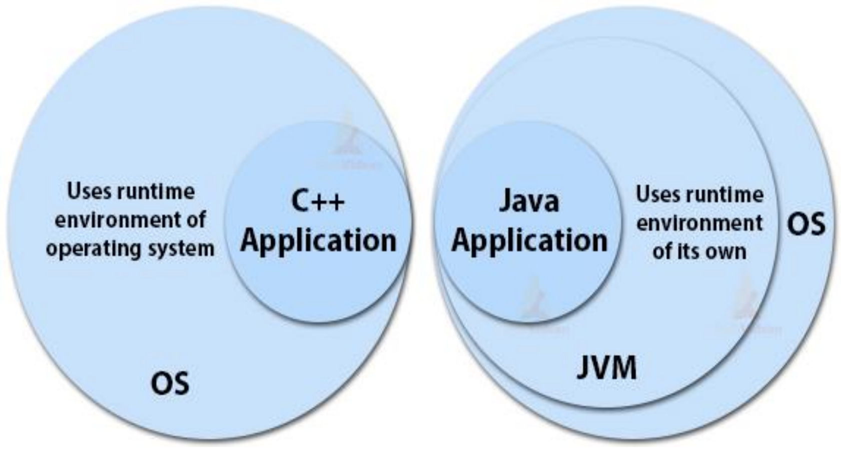


- **Portable**
  - ✓ Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.
- **Distributed**
  - ✓ Java is distributed because it facilitates users to create distributed applications in Java.
  - ✓ Java helps us to achieve this by providing the concept of RMI (Remote Method Invocation) and EJB (Enterprise JavaBeans).
- **Robust**
  - ✓ It uses strong memory management.
  - ✓ There is a lack of pointers that avoids security problems.
  - ✓ Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.

- ✓ Java also provides the concept of exception handling which identifies runtime errors and eliminates them.

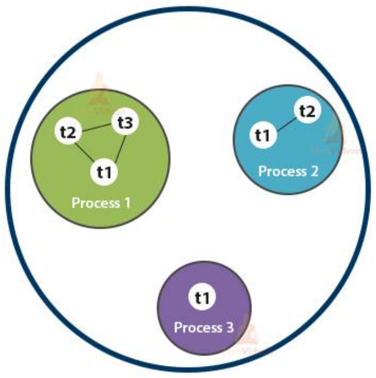
❖ **Secure**

- ✓ Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:
  - No explicit pointer
  - Java Programs run inside a virtual machine sandbox


Uses runtime environment of operating system
C++ Application
OS
Java Application
Uses runtime environment of its own
OS
JVM

❖ **Multi-threaded**

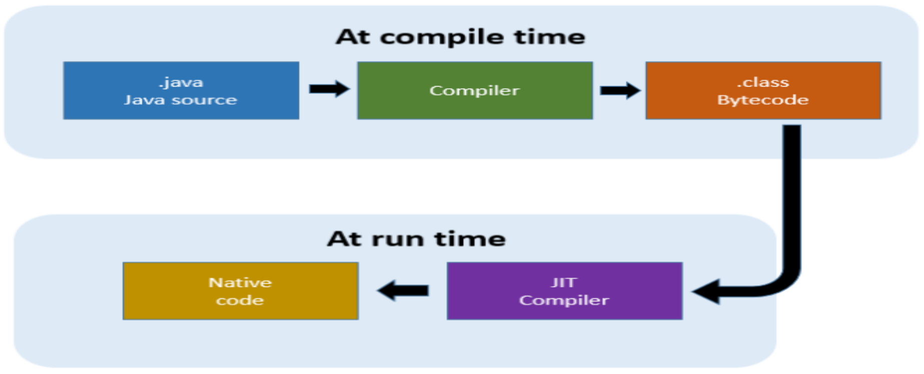
- ✓ A thread is like a separate program, executing concurrently.
- ✓ We can write Java programs that deal with many tasks at once by defining multiple threads.
- ✓ The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area.
- ✓ Threads are important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



Operating System

❖ **High Performance**

✓ Java provides high performance with the use of "JIT – Just In Time compiler", in which the compiler compiles the code on-demand basis, that is, it compiles only that method which is being called. This saves time and makes it more efficient.



❖ **Interactive**

✓ Java is interactive because its code supports effective CUI (Character User Interface) and GUI (Graphical User Interface) programs. It greatly improves the interactive performance of graphical applications.

❖ **Dynamic and Extensible**

✓ Java is dynamic and extensible means with the help of OOPs, we can add classes and add new methods to classes, creating new classes through subclasses. This makes it easier for us to expand our own classes and even modify them.

1.3 Overview of Java

1.3.1 Introduction to Java

- JAVA was developed by James Gosling at Sun Microsystems Inc in the year 1995, later acquired by Oracle Corporation.
- It is a simple programming language.
- Java makes writing, compiling, and debugging programming easy.
- It helps to create reusable code and modular programs.
- Java is a class-based, object-oriented programming language
- Java applications are compiled to byte code that can run on any Java Virtual Machine.
- The syntax of Java is similar to c/c++.

1.3.2 History of Java

- It is a programming language created in 1991.
- James Gosling, Mike Sheridan, and Patrick Naughton, a team of Sun engineers known as the Green team initiated the Java language in 1991.
- In 1995 Java was developed by James Gosling, who is known as the Father of Java.
- Sun Microsystems released its first public implementation in 1996 as Java 1.0. It promised Write Once, Run Anywhere (WORA), providing no-cost run-times on popular platforms.
- Java1.0 compiler was re-written in Java by Arthur Van Hoff to strictly comply with its specifications. With the arrival of Java 2, new versions had multiple configurations built for different types of platforms.

1.3.3 Java programming language is named JAVA. Why?

- Java is the name of an island in Indonesia where the first coffee (named java coffee) was produced.
- And this name was chosen by James Gosling while having coffee near his office. Java is just a name, not an acronym.
- Initially the name given was OAK

1.3.4 Java Terminology

- **Java Virtual Machine (JVM):**
  - There are three execution phases of a program. They are written, compile and run the program.
    - Writing a program is done by a java programmer.
    - JAVAC compiler which is a primary Java compiler included in the Java development kit (JDK). It takes the Java program as input and generates bytecode as output.
    - In the Running phase of a program, JVM executes the bytecode generated by the compiler.
    - Every Operating System has a different JVM but the output they produce after the execution of bytecode is the same across all the operating systems. This is why Java is known as a **platform-independent language.**

- **Bytecode in the Development process:**
  - Javac compiler of JDK compiles the java source code into bytecode so that it can be executed by JVM. It is saved as .class file by the compiler. To view the bytecode, a disassembler like javap can be used.
- **Java Development Kit (JDK):**
  - It is a complete Java development kit that includes everything including compiler, Java Runtime Environment (JRE), java debuggers, java docs, etc.
  - For the program to execute in java, we need to install JDK on our computer in order to create, compile and run the java program.
- **Java Runtime Environment (JRE):**
  - JDK includes JRE. JRE installation on our computers allows the java program to run, however, we cannot compile it.
  - JRE includes a browser, JVM, applet supports, and plugins. For running the java program, a computer needs JRE.
- **Garbage Collector:**
  - In Java, programmers can't delete the objects. To delete or recollect that memory JVM has a program called Garbage Collector.
  - Garbage Collectors can recollect the objects that are not referenced. So Java makes the life of a programmer easy by handling memory management.
- **ClassPath:**
  - The classpath is the file path where the java runtime and Java compiler look for .class files to load.
  - By default, JDK provides many libraries. If you want to include external libraries they should be added to the classpath.

1.3.5 Applications of JAVA

- Mobile applications (specially Android apps)
- Desktop applications
- Web applications
- Web servers and application servers
- Games
- Database connection

1.3.6 Advantages of using JAVA

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

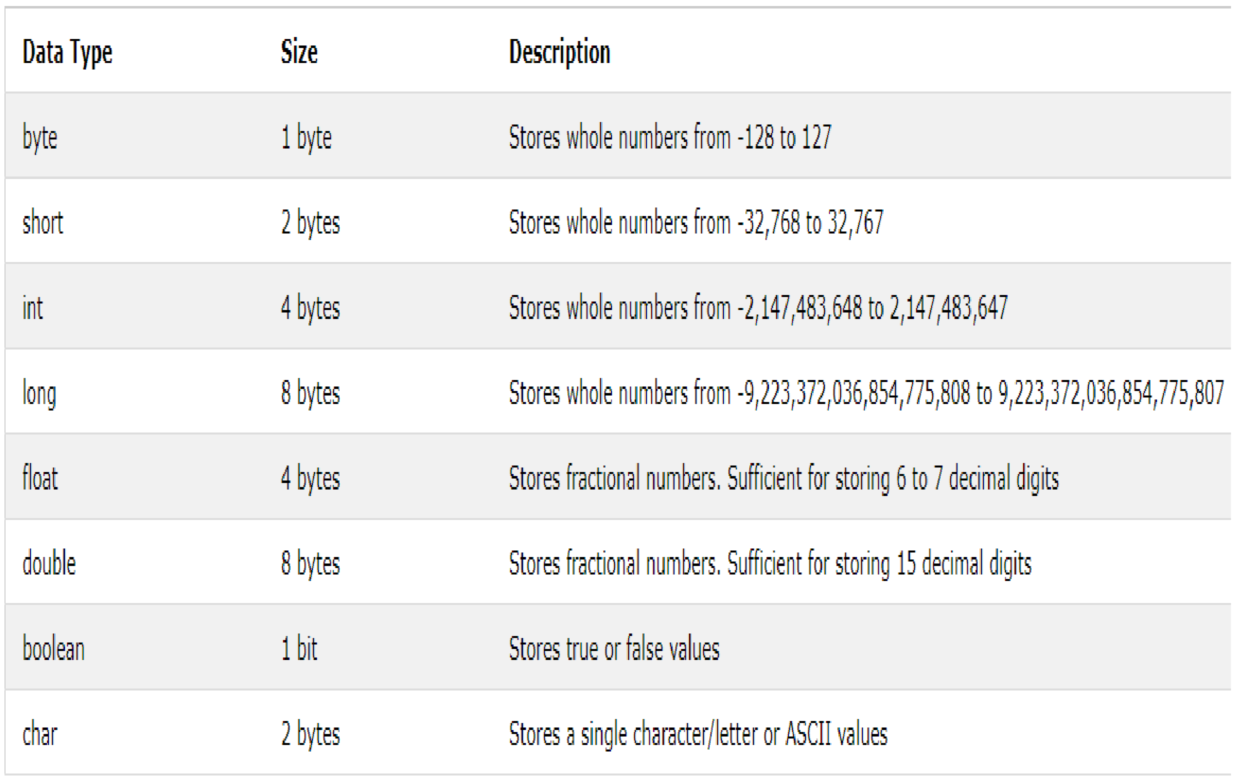
1.4 Data Types, Variables and Arrays

1.4.1 Java Data Types

- A data type is a classification of data which tells the compiler or interpreter how the programmer intends to use the data.
- Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:
  - **Primitive data types**:
  - **Non-primitive data types**:

1.4.1.1 Java Primitive Data Types

- The primitive data types are the predefined data types of Java. They specify the size and type of any standard values.
- **The primitive data types** include boolean, char, byte, short, int, long, float and double.
- **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.



Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values

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.
- 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 minimum value is - 2,147,483,648 and maximum value is 2,147,483,647.
- ➢ Its default value is 0.
- ➢ Example: **int a = 100000, int b = -200000 ;**

Long Data Type

- ➢ The long data type is a 64-bit two's complement integer.
- ➢ 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.
- ➢ 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.
- ➢ 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.
- ➢ Its default value is 0.0d.
- ➢ Example: **double d1 = 12.3;**

Char Data Type

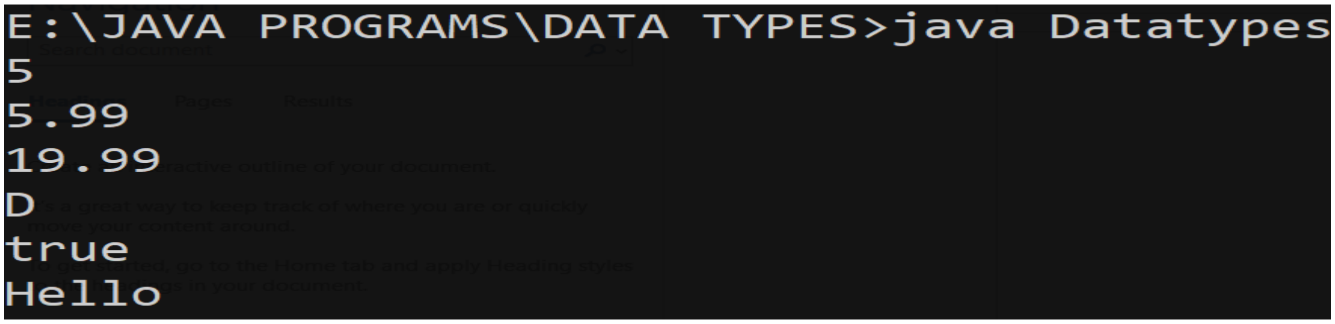
- ➢ The char data type is a single 16-bit Unicode character.
- ➢ Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).
- ➢ The char data type is used to store characters.

- Example: **char letterA = 'A';**

1.4.1.2 Example program using different Data Types in JAVA

```
class Datatypes
{
public static void main (String[] args)
{
int a = 5;              // integer (whole number)
float f = 5.99f;    // floating point number
double d=19.99d; // double
char ch = 'D';        // character
boolean B = true;      // boolean
String S = "Hello";     // String
System.out.println(a);
System.out.println(f);
System.out.println(d);
System.out.println(ch);
System.out.println(B);
System.out.println(S);
}
}
```

OUTPUT



```
E:\JAVA PROGRAMS\DATA TYPES>java Datatypes
5
5.99
19.99
D
true
Hello
```

1.4.1.3 Java Non-Primitive Data Types

- ➢ Non primitive datatypes are those which uses primitive datatype as base like array, class etc.
- ➢ Non-primitive data types are called reference types because they refer to objects.

Strings

- The String data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:
- Example:

```
String greeting = "Hello World";
System.out.println(greeting);
```

1.4.1.3 Difference between Primitive and Non-Primitive Data Types

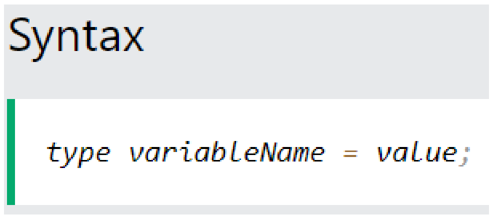
Primitive Data Type	Non-Primitive Data Type
Predefined in JAVA	Created by the programmer (except String)
Primitive types cannot be used to call methods	Non-primitive types can be used to call methods to perform certain operations
A primitive type always has a value. It can't be Null.	Non-primitive types can be null.
A primitive type starts with a lowercase letter	Non-primitive types start with an uppercase letter.
The size of a primitive type depends on the data type	Non-primitive types have all the same size.

1.4.2 Variables in JAVA

- ➢ Variables are containers for storing data values.
- ➢ In Java, there are different types of variables, for example:
  - String - stores text, such as "Hello". String values are surrounded by double quotes
  - int - stores integers (whole numbers), without decimals, such as 123 or -123

- float - stores floating point numbers, with decimals, such as 19.99 or -19.99
- char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- boolean - stores values with two states: true or false

1.4.2.1 Declaring (Creating) Variables



Syntax

```
type variableName = value;
```

➢ Where type is one of Java's types (such as int or String), and variable Name is the name of the variable (such as x or name). The equal sign is used to assign values to the variable.

1.4.2.2 Types of Variables

Local Variable

➢ These are the variables declared within a method. Within the method we can directly access the variables. Outside the method we can't access these variables. Variables can be directly accessed without creating objects.

Instance Variable

➢ Instance variables are declared inside the class, but not inside the method. For accessing instance Variable, we have to create an object. Without creating object, we can't access the Instance variable.

Static Variables

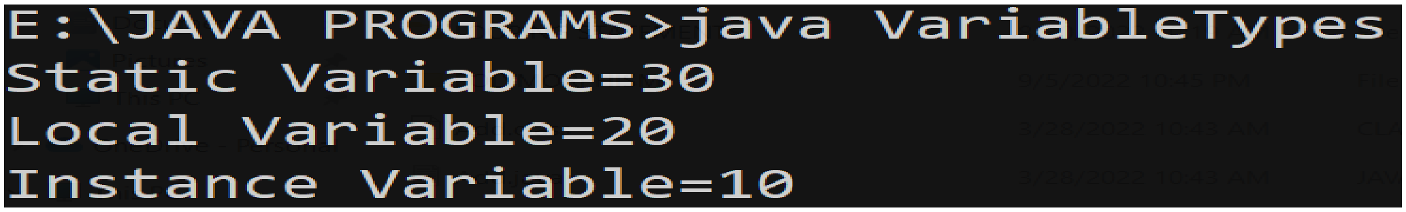
➢ Declared using Static Keyword. For Static Variables Memory is allotted only once. Static Variables can be directly accessed without creating objects.

Example Program for illustrating the different types of variables

```
class Variables
 {
 static int c=30;          //Static Variable
 int a=10;                // Instance Variable
```

```
        public static void main (String args[])
          {
           int b=20;              //Local Variables
           System.out.println("Static Variable=" +c);
           System.out.println("Local Variable=" +b);
           Variables V=new Variables();   //Object Creation
    System.out.println("Instance Variable=" +V.a);   //Accessing instance variable using object
          }
      }
```

OUTPUT



```
E:\JAVA PROGRAMS>java VariableTypes
Static Variable=30
Local Variable=20
Instance Variable=10
```

1.4.3 JAVA Arrays

- Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.
- To declare an array, define the variable type with square brackets:

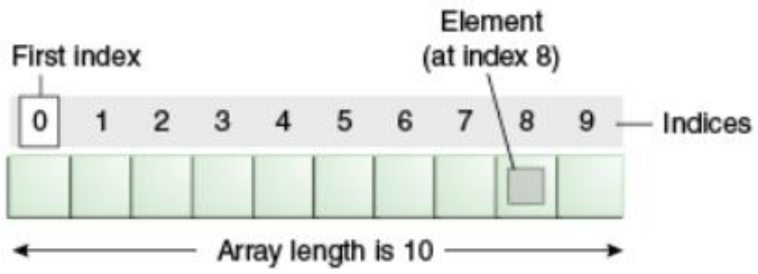


Fig: An array of 10 elements.

- Each item in an array is called an element, and each element is accessed by its numerical index.
- As shown in the above figure, numbering begins with 0. The 9th element, for example, would therefore be accessed at index 8.

1.4.3.1 Types of Arrays

- Single Dimensional Array
- Multidimensional Array

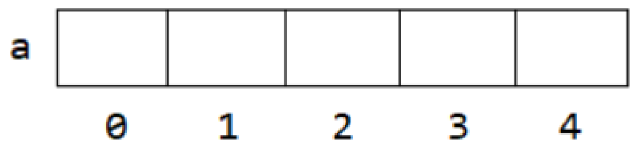
1.4.3.1.1 Single Dimensional Array

- A Single-Dimensional Array in Java programming is a special type of variable that can store multiple values of only a single data type such as int, float, double, char, structure, pointer, etc. at a contagious location in computer memory.
- Here contagious location means at a fixed gap in computer memory.

1.4.3.1.2 Declaration Syntax of a One-Dimensional Array in Java

- Method 1: datatype variable_name[ ] = new datatype[size];
- Method 2: datatype[ ] variable_name = new datatype[size];
    - Here, size is the number of elements we want to store in the array.

- Example
    - int a[ ]=new int[5];
    - int [ ] a=new int[5];
- Once we declare the 1D Array, it will look like as shown in the picture below:

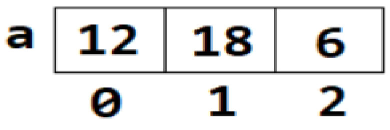


- In above image we can see that the name of the one-dimensional array is a and it can store 5 integer numbers. Size of the array is 5. Index of the array is 0, 1, 2, 3 and 4.
- The first index is called Lower Bound, and the last index is called an Upper Bound. Upper Bound of a one dimensional is always Size – 1.

1.4.3.1.3 Declaration and Initialization of a One-Dimensional Array in Java

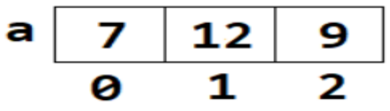
- In Java programming a one-dimensional array can be declared and initialized in several ways.
- Method 1:

- int a[ ]=new int[ ] {12,18,6};



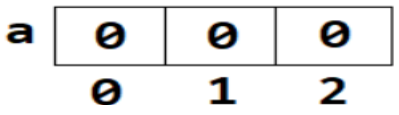
- Method 2

- int a[ ]={7,12,9};



- Method 3

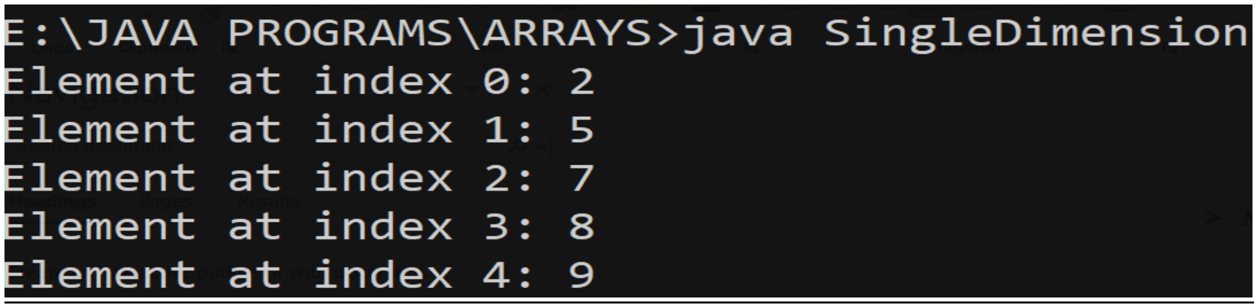
- int a[ ]=new int[3];



1.4.3.1.4 Example Program using Single-Dimensional Arrays

```
class SingleDimension
 {
 public static void main(String ar[])
  {
   int age[]={2,5,7,8,9};
   for(int i=0;i<=4;i++)
   System.out.println("Element at index " + i +": " + age[i]);
  }
 }
```

Output

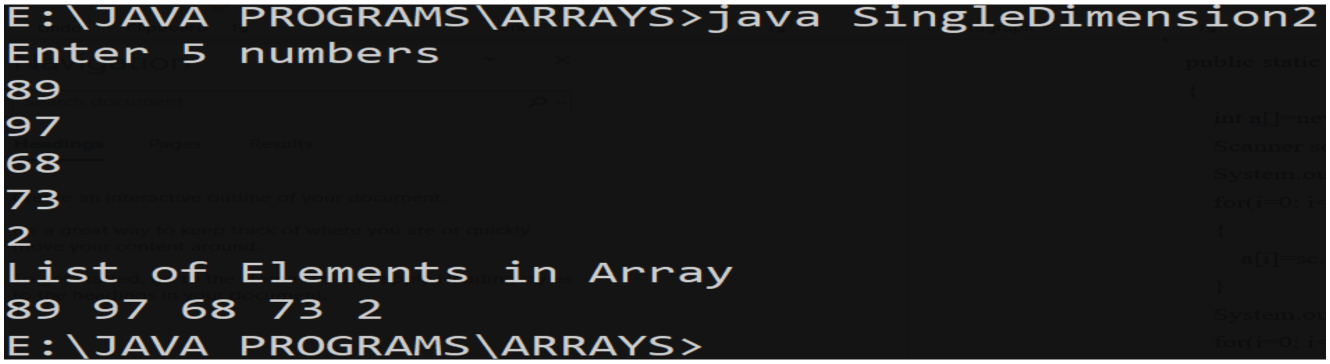


```
E:\JAVA PROGRAMS\ARRAYS>java SingleDimension
Element at index 0: 2
Element at index 1: 5
Element at index 2: 7
Element at index 3: 8
Element at index 4: 9
```

1.4.3.1.2 Program to input 5 numbers in an array and display the numbers present in the array.

```
import java.util.Scanner;
public class SingleDimension2
{
   public static void main(String args[])
   {
      int a[]=new int[5], i;
      Scanner sc=new Scanner(System.in);
      System.out.println("Enter 5 numbers");
      for(i=0; i<5; i++)
      {
         a[i]=sc.nextInt();
      }
      System.out.println("List of Elements in Array");
      for(i=0; i<5; i++)
      {
            System.out.print(a[i]+" ");
      }
   }
}
```

OUTPUT



```
E:\JAVA PROGRAMS\ARRAYS>java SingleDimension2
Enter 5 numbers
89
97
68
73
2
List of Elements in Array
89 97 68 73 2
E:\JAVA PROGRAMS\ARRAYS>
```

1.4.3.2 Multi-Dimensional Array

- In Java, a multi-dimensional array is nothing but an array of arrays. A two-dimensional array in Java is represented as an array of one-dimensional arrays of the same type.
- It consists of rows and columns and looks like a table. A 2D array is also known as Matrix.

1.4.3.2.1 Declaration Syntax of a Two-Dimensional Array in Java

- **Method 1:**

  datatype variable_name[][] = new datatype[row_size][column_size];

- **Method 2:**

  datatype[][] variable_name = new datatype[row_size][column_size];

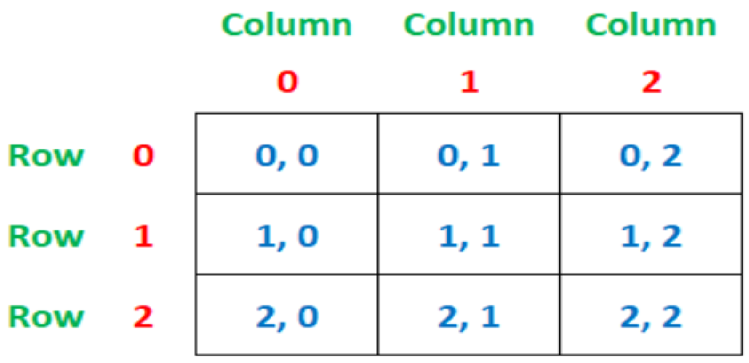
  - Here row_size is the number of rows we want to create in a 2D array and, column_size is the number of columns in each row.
  - **Example 1**

    int a[][]=new int[3][3];

  - **Example 2**

    int[][] a=new int[3][3];

  - Once we declare the 2D Array, it will look like as shown in the picture below:



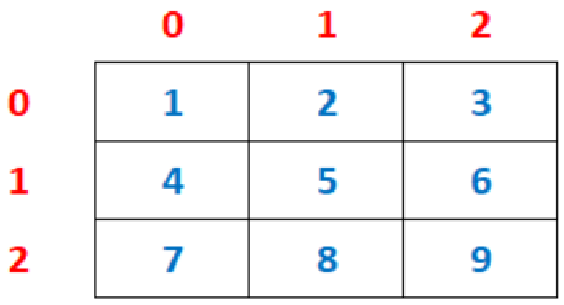
	Column 0	Column 1	Column 2
Row 0	0, 0	0, 1	0, 2
Row 1	1, 0	1, 1	1, 2
Row 2	2, 0	2, 1	2, 2

- In the above image, you can see that we have created a 2D Array having 3 rows and 3 columns. We can call the above array as a 3x3 Matrix.
- The first row and column always start with index 0. 2D array is used to store data in the form of a table.

1.4.3.2.2 Declaration and Initialization of a Two-Dimensional Array in Java

Example 1:

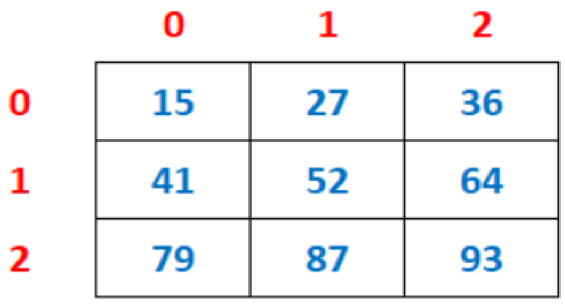
int a[][]= new int[][] {{1,2,3}, {4,5,6}, {7,8,9}};



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

Example 2:

int a[][]= {{15,27,36}, {41,52,64}, {79,87,93}};



	0	1	2
0	15	27	36
1	41	52	64
2	79	87	93

Example 3:

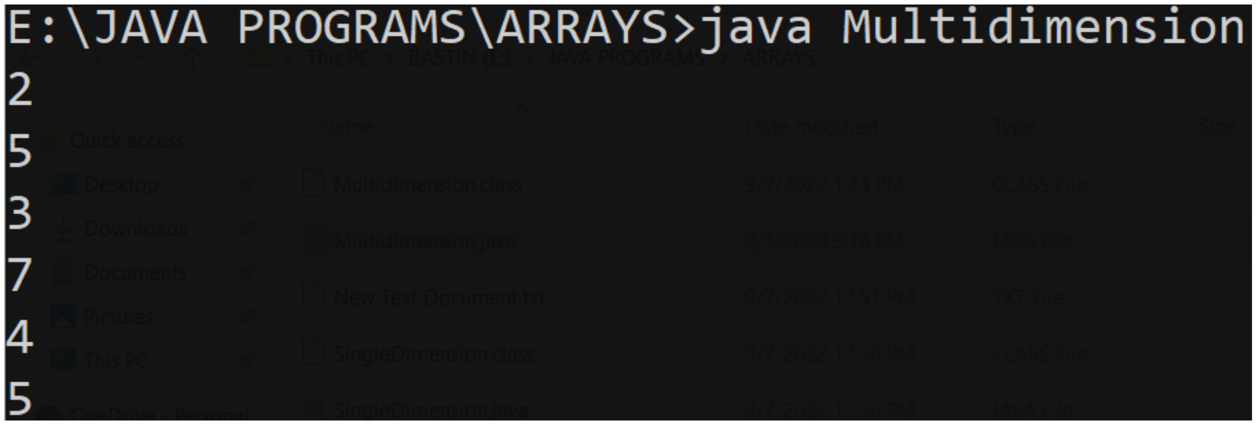
int a[][]= new int[3][3];

$$\begin{array}{c|c|c|c|}
 & 0 & 1 & 2 \\ \hline
0 & 0 & 0 & 0 \\ \hline
1 & 0 & 0 & 0 \\ \hline
2 & 0 & 0 & 0 \\ \hline
\end{array}$$

1.4.3.2.3 Example Program for Multi-Dimensional Array

```
   class Multidimension
    {
    public static void main(String arg[])
      {
       int arraynum[][]={{2,5},{3,7},{4,5}};
       for(int a=0;a<3;a++)
        {
         for(int b=0;b<2;b++)
           {
            System.out.println(arraynum[a][b]);
           }
         }
  }
}
```

OUTPUT



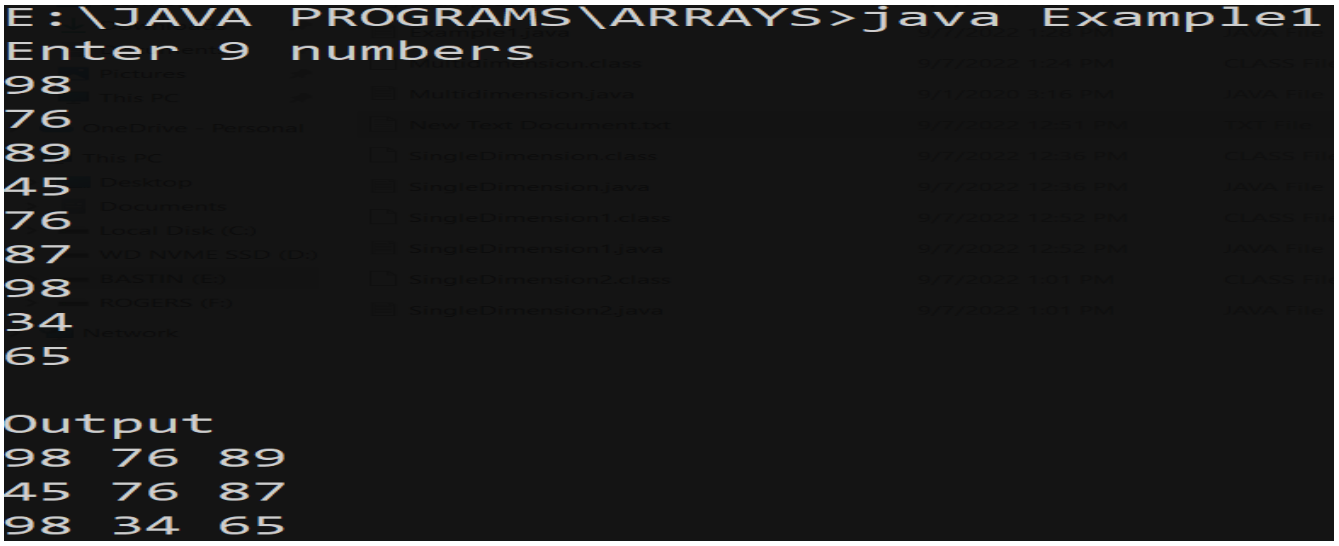
```
E:\JAVA PROGRAMS\ARRAYS>java Multidimension
2
5
3
7
4
5
```

1.4.3.2.4 Program to input numbers in a 3x3 Matrix and display the numbers in a table format.

Program

```
import java.util.Scanner;
public class Example
{
    public static void main(String args[])
    {
        int a[][]=new int[3][3];
        Scanner sc=new Scanner(System.in);
        int r,c;
        System.out.println("Enter 9 numbers");
        for(r=0; r<3; r++)
        {
            for(c=0; c<3; c++)
            {
                a[r][c]=sc.nextInt();
            }
        }
        System.out.println("\nOutput");
        for(r=0; r<3; r++)                          // this loop is for row
        {
            for(c=0; c<3; c++)                  // this loop will print 3 numbers in each row
            {
                System.out.print(a[r][c]+" ");
            }
            System.out.println();  // break the line after printing the numbers in a row
        }
    }
}
```

OUTPUT



```
E:\JAVA PROGRAMS\ARRAYS>java Example1
Enter 9 numbers
98
76
89
45
76
87
98
34
65

Output
98 76 89
45 76 87
98 34 65
```

OPERATORS

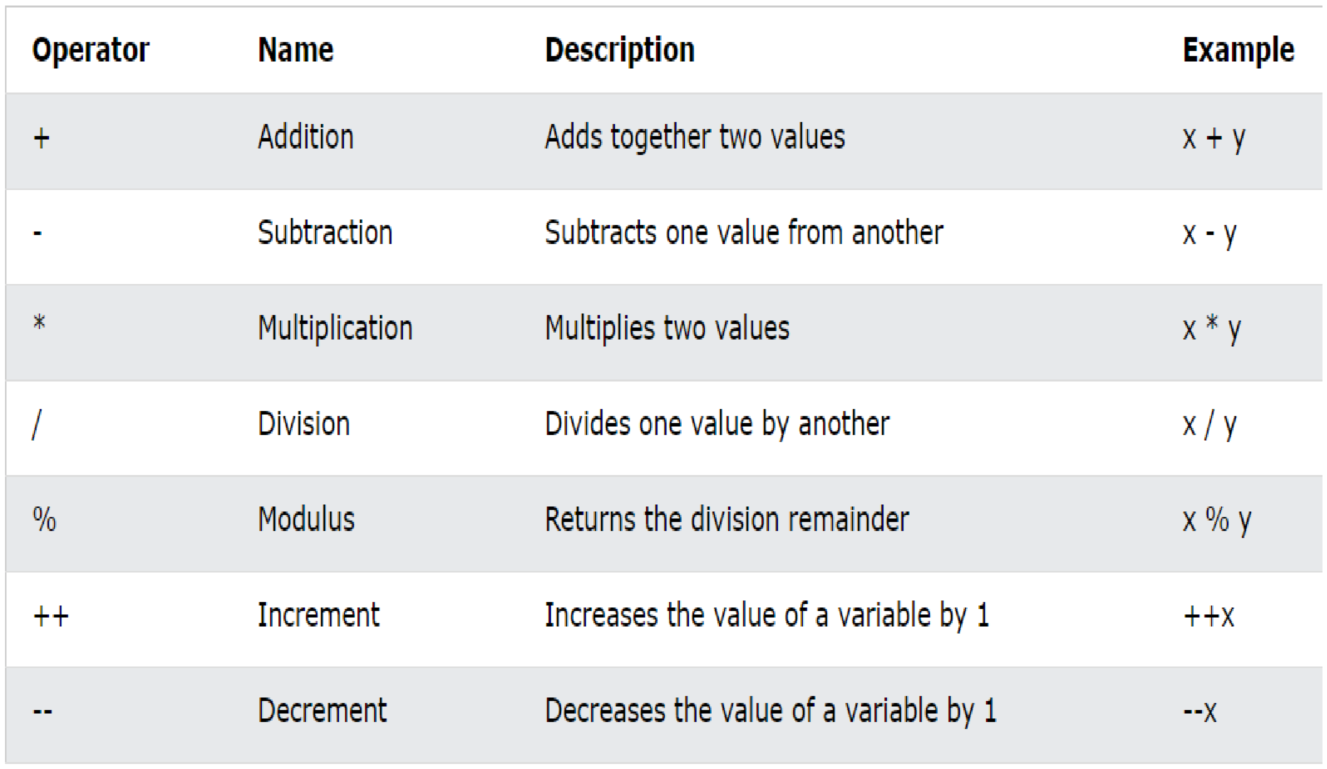
- ➢ Operators are symbols that perform operations on variables and values. For example, + is an operator used for addition, while * is also an operator used for multiplication.

1.4.4 Types of Operators

- ✓ Arithmetic Operators
- ✓ Assignment Operators
- ✓ Relational Operators
- ✓ Logical Operators
- ✓ Unary Operators
- ✓ Bitwise Operators

1.4.4.1 Java Arithmetic Operators

- ❖ Arithmetic operators are used to perform arithmetic operations on variables and data.
- ❖ For example, a + b; Here, the + operator is used to add two variables a and b. Similarly, there are various other arithmetic operators in Java.



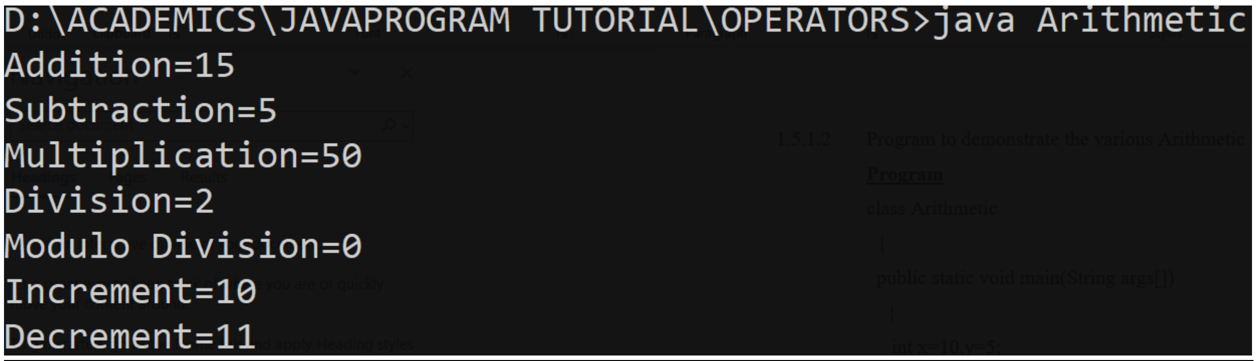
Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

1.4.4.2 Program to demonstrate the various Arithmetic Operations using JAVA

Program

```
class Arithmetic
 {
 public static void main(String args[])
  {
   int x=10,y=5;
   System.out.println("Addition="+(x+y));         //Perform Addition
   System.out.println("Subtraction="+(x-y));      //Perform Subtraction
   System.out.println("Multiplication="+(x*y));   //Perform Multiplication
   System.out.println("Division="+(x/y));         //Perform Division
   System.out.println("Modulo Division="+(x%y));  //Perform Modulo Division
   System.out.println("Increment="+(x++));        //Perform Increment operation
   System.out.println("Decrement="+(x--));        //Perform Decrement operation
  }
 }
```

Output



```
D:\ACADEMICS\JAVAPROGRAM TUTORIAL\OPERATORS>java Arithmetic
Addition=15
Subtraction=5
Multiplication=50
Division=2
Modulo Division=0
Increment=10
Decrement=11
```

1.4.4.3 Java Assignment Operators

❖ Assignment operators are used in Java to assign values to variables. For example,

```
int age;
age = 5;
```

❖ Here, = is the assignment operator. 5 is assigned to the variable age

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

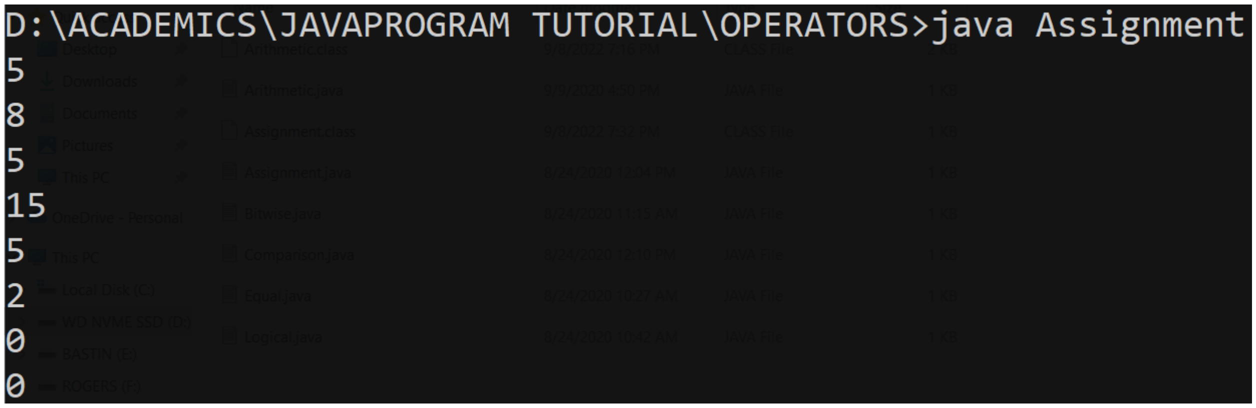
1.4.4.4 Program to demonstrate the various Assignment Operations using JAVA

Program

```
class Assignment {
  public static void main(String[] args)
{
    int x = 5;
    System.out.println(x);
    x+=3;
    System.out.println(x);
    x-=3;
    System.out.println(x);
    x*=3;
    System.out.println(x);
    x/=3;
```

```
            System.out.println(x);
            x%=3;
            System.out.println(x);
            x>>=3;
            System.out.println(x);
            x<<=3;
            System.out.println(x);
        }
    }
```

OUTPUT



```
D:\ACADEMICS\JAVAPROGRAM TUTORIAL\OPERATORS>java Assignment
5
8
5
15
5
2
0
0
```

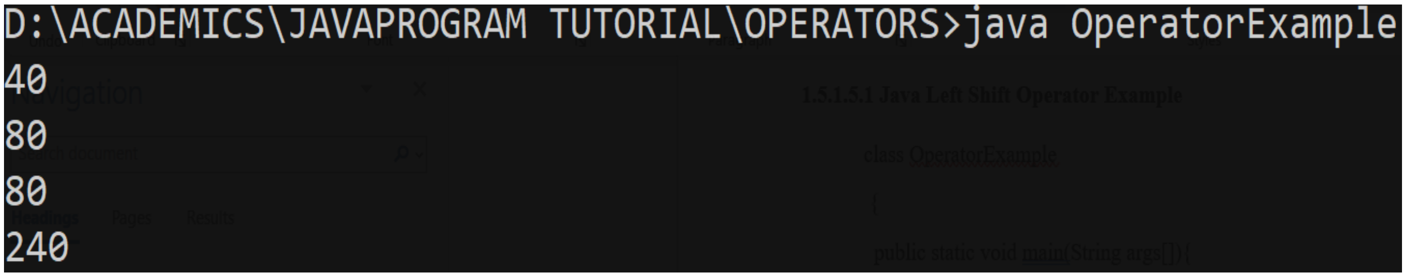
1.5.1.5 Java Left Shift Operator

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

1.5.1.5.1 Java Left Shift Operator Example

```
class OperatorExample
 {
  public static void main(String args[]){
  System.out.println(10<<2);//10*2^2=10*4=40
  System.out.println(10<<3);//10*2^3=10*8=80
  System.out.println(20<<2);//20*2^2=20*4=80
 System.out.println(15<<4);//15*2^4=15*16=240
 }
}
```

OUTPUT



```
D:\ACADEMICS\JAVAPROGRAM TUTORIAL\OPERATORS>java OperatorExample
40
80
80
240
```

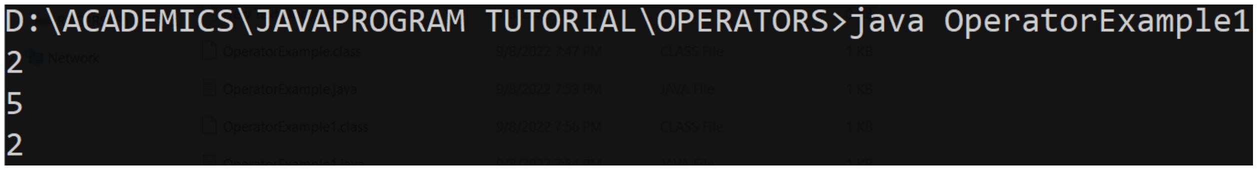
1.4.4.5 Java Right Shift Operator

- 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.

1.5.1.5.1 Java Right Shift Operator Example

```
class OperatorExample1
 {
  public static void main(String args[])
    {
     System.out.println(10>>2);  //10/2^2=10/4=2
     System.out.println(20>>2);  //20/2^2=20/4=5
     System.out.println(20>>3);  //20/2^3=20/8=2
    }
 }
```

Output



```
D:\ACADEMICS\JAVAPROGRAM TUTORIAL\OPERATORS>java OperatorExample1
2
5
2
```

1.4.5 Java Relational Operators

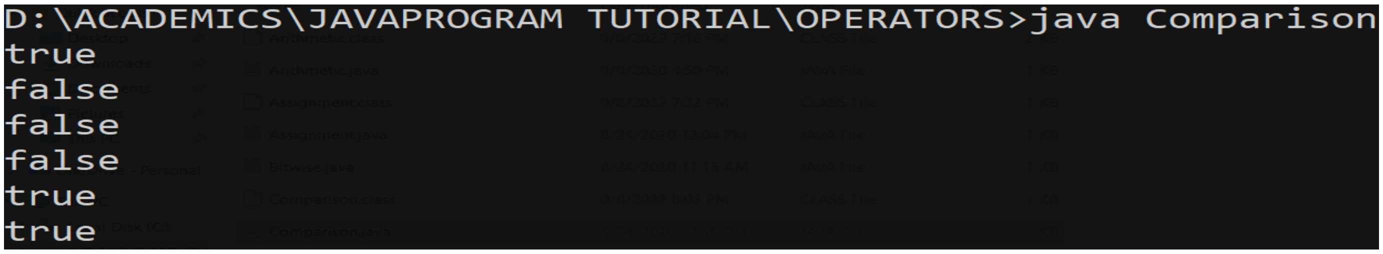
- Relational operators are used to check the relationship between two operands. For example, a < b; Here, < operator is the relational operator. It checks if a is less than b or not. It returns either true or false.

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

1.4.5.1 Program to demonstrate the various Relational Operators using JAVA

```
class Comparison
 {
 public static void main(String[] args)
  {
   int x = 5;
   int y = 5;
   System.out.println(x == y); // returns false because 5 is not equal to 3
   System.out.println(x != y); // returns true because 5 is not equal to 3
   System.out.println(x > y); // returns true because 5 is greater than 3
   System.out.println(x < y); // returns false because 5 is not less than 3
   System.out.println(x >= y); // returns true because 5 is greater, or equal, to 3
   System.out.println(x <= y); // returns false because 5 is neither less than or equal to 3
 }
}
```

Output



```
D:\ACADEMICS\JAVAPROGRAM TUTORIAL\OPERATORS>java Comparison
true
false
false
false
true
true
```

1.4.6 Java Logical Operators

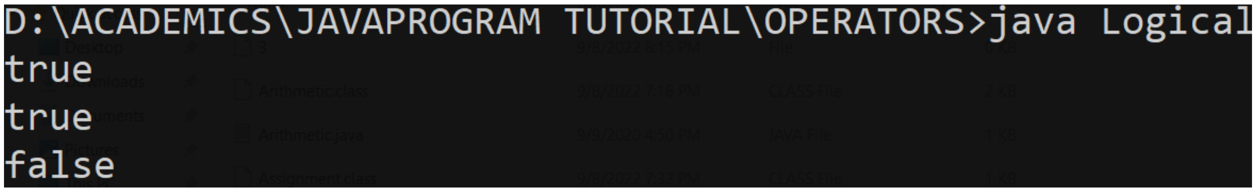
❖ Logical operators are used to determine the logic between variables or values:

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

1.4.6.1 Program to demonstrate the various logical Operators using JAVA

```
class Logical
 {
  public static void main(String[] args)
   {
    int x = 5;
    System.out.println(x > 3 && x < 10);
    System.out.println(x > 3 || x < 4);
    System.out.println(!(x > 3 && x < 10));
    }
 }
```

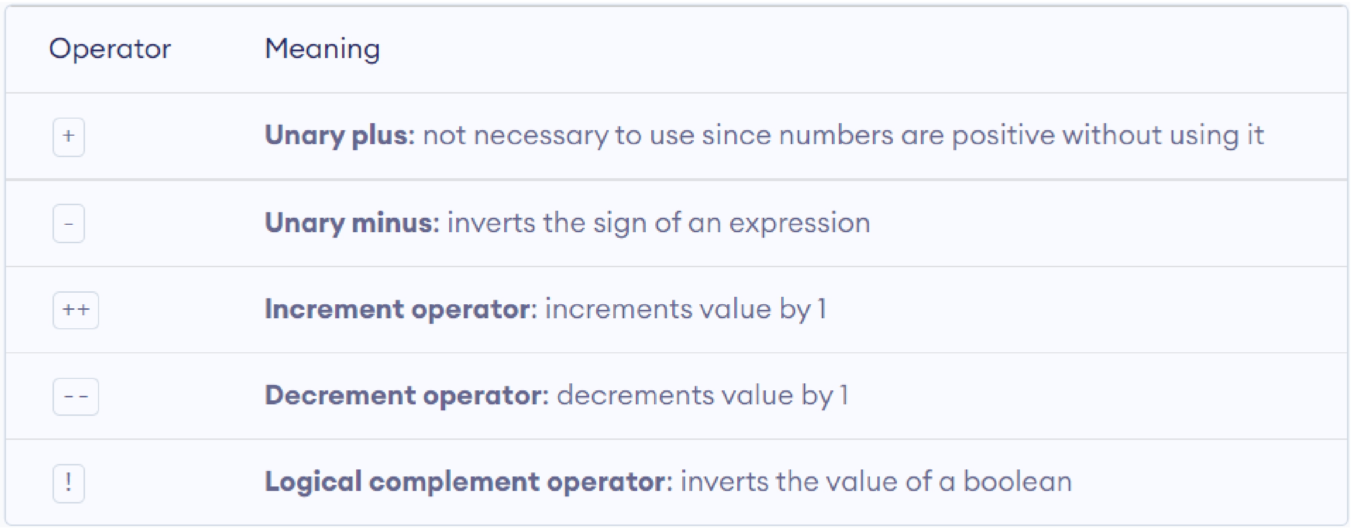
Output



```
D:\ACADEMICS\JAVAPROGRAM TUTORIAL\OPERATORS>java Logical
true
true
false
```

1.4.7 Java Unary Operators

➢ Unary operators are used with only one operand. For example, ++ is a unary operator that increases the value of a variable by 1. That is, ++5 will return 6.

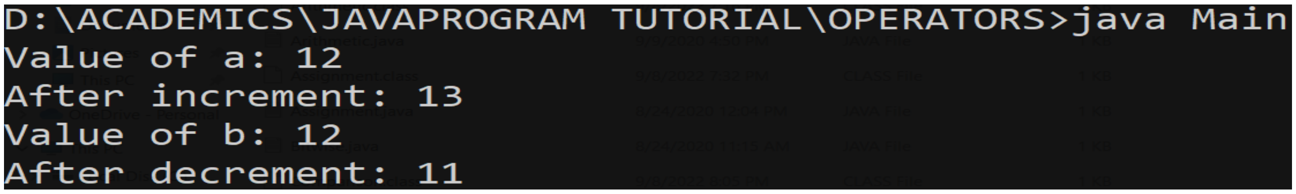


Operator	Meaning
`+`	**Unary plus:** not necessary to use since numbers are positive without using it
`-`	**Unary minus:** inverts the sign of an expression
`++`	**Increment operator:** increments value by 1
`--`	**Decrement operator:** decrements value by 1
`!`	**Logical complement operator:** inverts the value of a boolean

1.4.7.1 Example program to illustrate the use of Unary Operator

```
class Main
 {
 public static void main(String[] args)
  {
  // declare variables
  int a = 12, b = 12;
  int result1, result2;
  // original value
  System.out.println("Value of a: " + a);
  // increment operator
  result1 = ++a;
  System.out.println("After increment: " + result1);
  System.out.println("Value of b: " + b);
  // decrement operator
  result2 = --b;
  System.out.println("After decrement: " + result2);
  }
}
```

Output


```
D:\ACADEMICS\JAVAPROGRAM TUTORIAL\OPERATORS>java Main
Value of a:  12
After increment:  13
Value of b:  12
After decrement:  11
```

1.5 CONTROL STATEMENTS

- A programming language uses control statements to control the flow of execution of a program based on certain conditions. It is one of the fundamental features of Java, which provides a smooth flow of program.
- Java provides three types of control flow statements.
  1. Decision Making statements
     - if statements
     - if else
     - else if
     - switch statement
  2. Loop statements
     - do while loop
     - while loop
     - for loop
     - for-each loop
  3. Jump statements
     - break statement
     - continue statement

1.5.1 Decision Making Statements

1.6.1.1 Simple if Statement

- The Java if statement tests the condition. It executes the *if block* if condition is true.

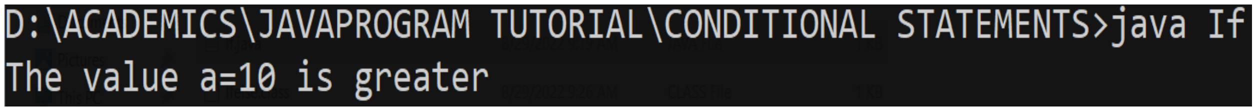
1.5.1.1.1 Syntax:

```
if(condition)
{
//code to be executed
}
```

1.5.1.1.2 Example Program

```
class If
 {
 public static void main(String arg[])
  {
   int a=10,b=5;
   if(a>b)
     {
     System.out.println("The value a=10 is greater" );
     }
   }
 }
```

Output



```
D:\ACADEMICS\JAVAPROGRAM TUTORIAL\CONDITIONAL STATEMENTS>java If
The value a=10 is greater
```

1.6.1.2 If-else Statement

- The Java if-else statement also tests the condition. It executes the if block if condition is true otherwise else block is executed.

1.6.1.2.1 Syntax

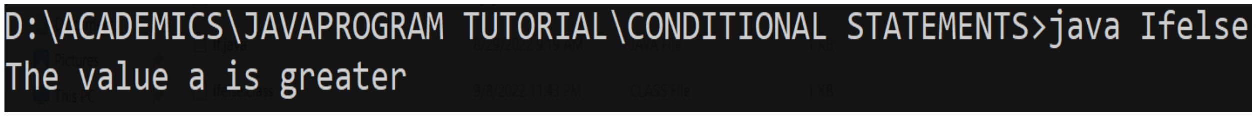
```
if(condition)
{
//code if condition is true
}
else
{
```

```
//code if condition is false
}
```

1.6.1.2.2 Program using if-else statement in JAVA

```
class Ifelse
 {
   public static void main(String ab[])
     {
      int a=16,b=10;
      if(a>b)
     {
      System.out.println("The value a is greater");
     }
   else
     System.out.println("The value b is greater");
    }
  }
```

Output



```
D:\ACADEMICS\JAVAPROGRAM TUTORIAL\CONDITIONAL STATEMENTS>java Ifelse
The value a is greater
```

1.6.1.3 Else if Statement

- The if-else-if ladder statement executes one condition from multiple statements.

1.6.1.3.1 Syntax:

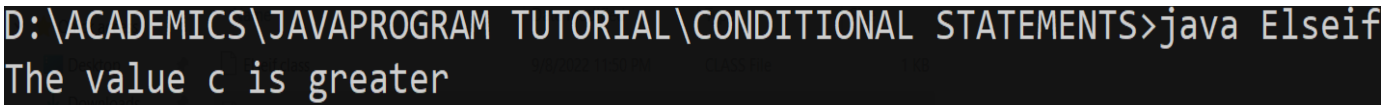
```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
```

```
...
else{
//code to be executed if all the conditions are false
}
```

1.6.1.3.2 Example program using else-if statement

```
class Elseif
 {
 public static void main(String arg[])
  {
   int a=6,b=5,c=7;
   if(a>b && a>c)
     {
      System.out.println("The value a is greater");
     }
   else if(b>a && b>c)
     {
     System.out.println("The value b is greater");
     }
   else
      {
     System.out.println("The value c is greater");
      }
   }
 }
```

OUTPUT



```
D:\ACADEMICS\JAVAPROGRAM TUTORIAL\CONDITIONAL STATEMENTS>java Elseif
The value c is greater
```

1.6.1.4 Nested if statement

- The nested if statement represents the if block within another if block. Here, the inner if block condition executes only when outer if block condition is true.

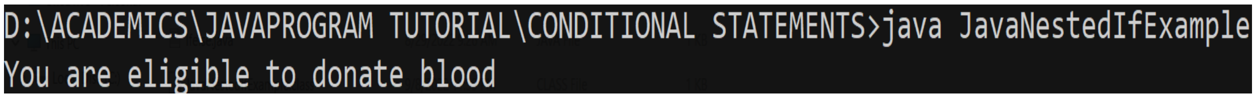
1.6.1.4.1 Syntax

```
if(condition)
{
    //code to be executed
        if(condition)
  {
          //code to be executed
   }
}
```

1.6.1.4.2 Java Program to demonstrate the use of Nested If Statement.

```
class JavaNestedIfExample
  {
    public static void main(String[] args)
      {
     //Creating two variables for age and weight
       int age=20;
       int weight=80;
     //applying condition on age and weight
       if(age>=18)
        {
          if(weight>50)
            {
            System.out.println("You are eligible to donate blood");
            }
        }
      }
  }
```

OUTPUT



```
D:\ACADEMICS\JAVAPROGRAM TUTORIAL\CONDITIONAL STATEMENTS>java JavaNestedIfExample
You are eligible to donate blood
```

1.6.1.5 JAVA switch statement

- The Java switch statement executes one statement from multiple conditions. The switch statement tests the equality of a variable against multiple values.

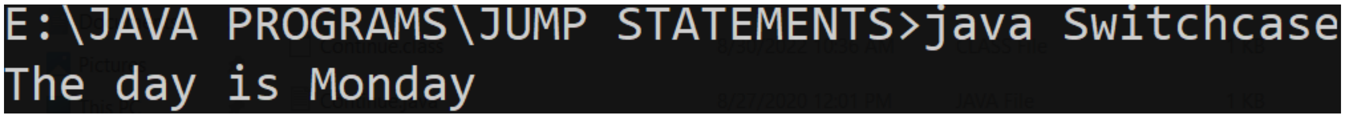
1.6.1.5.1 Syntax

```
switch(expression){
case value1:
 //code to be executed;
 break;  //optional
case value2:
 //code to be executed;
break;  //optional
......
default:
  code to be executed if all cases are not matched;
}
```

1.6.1.5.2 JAVA Program using Switch

```
class Switchcase {
    public static void main(String[] args) {
        int week = 2;
        String day;
        // switch statement to check day
        switch (week) {
            case 1:
                day = "Sunday";
                break;
            case 2:
                day = "Monday";
                break;
            case 3:
                day = "Tuesday";
                break;
            // match the value of week
            case 4:
                day = "Wednesday";
                break;
case 5:
    day = "Thursday";
    break;
case 6:
    day = "Friday";
    break;
case 7:
    day = "Saturday";
    break;
default:
    day = "Invalid day";
    break;
}
System.out.println("The day is " + day);
}
}
```

OUTPUT



```
E:\JAVA PROGRAMS\JUMP STATEMENTS>java Switchcase
The day is Monday
```

1.6.2 JAVA Looping Statements

- Loops are used to repeat a block of code. For example, if you want to show a message 100 times, then rather than typing the same code 100 times, you can use a loop.
- JAVA Looping statement types
  - For loop
  - While loop
  - Do-While loop
  - for-each loop

1.6.2.1 For Statement

- Java for loop is used to run a block of code for a certain number of times.

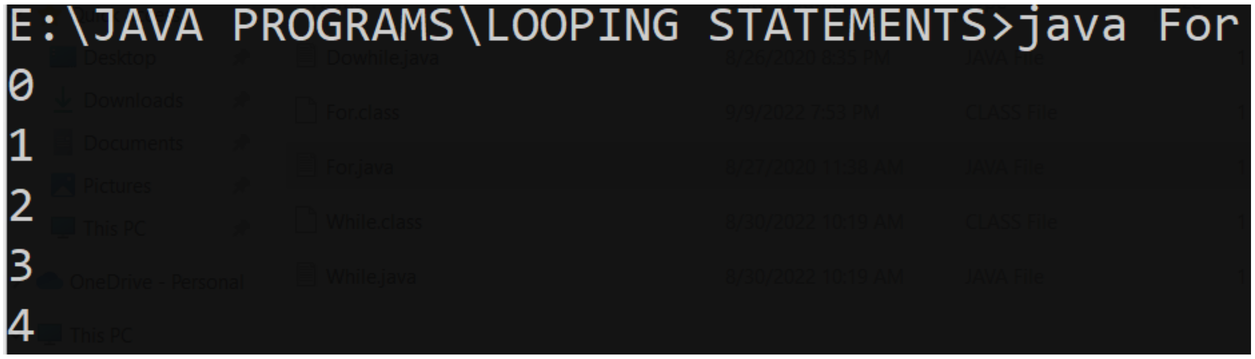
1.6.2.1.1 Syntax

```
for(initialization; condition; increment/decrement)
{
   // body of the loop
}
```

1.6.2.1.2 Simple Program using For Loop

```
class For
 {
  public static void main(String arg[])
    {
     int i;
     for(i=0;i<5;i++)
     System.out.println(i);
    }
  }
```

OUTPUT



```
E:\JAVA PROGRAMS\LOOPING STATEMENTS>java For
0
1
2
3
4
```

1.6.2.2 While Statement

- Java while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition.
- The while loop is considered as a repeating if statement.
- If the number of iterations is not fixed, it is recommended to use the while loop.

1.6.2.2.1 Syntax

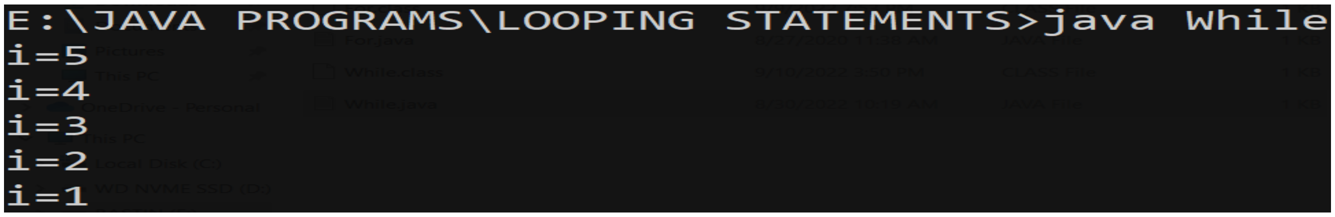
```
while (condition)
{
//code to be executed
increment / decrement statement
}
```

1.6.2.2.2 Example program using While

```
class While
 {
  public static void main(String ab[])
   {
  int i;
  i=5;
  while(i>0)
    {
    System.out.println("i="+i);
```

```
            i--;
           }
         }
        }
```

1.6.2.2.3 OUTPUT



```
E:\JAVA PROGRAMS\LOOPING STATEMENTS>java While
i=5
i=4
i=3
i=2
i=1
```

1.6.2.3 Java do-while Loop

- Java do-while loop is used to execute a block of statements continuously until the given condition is true.
- The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

1.6.2.3.1 Syntax

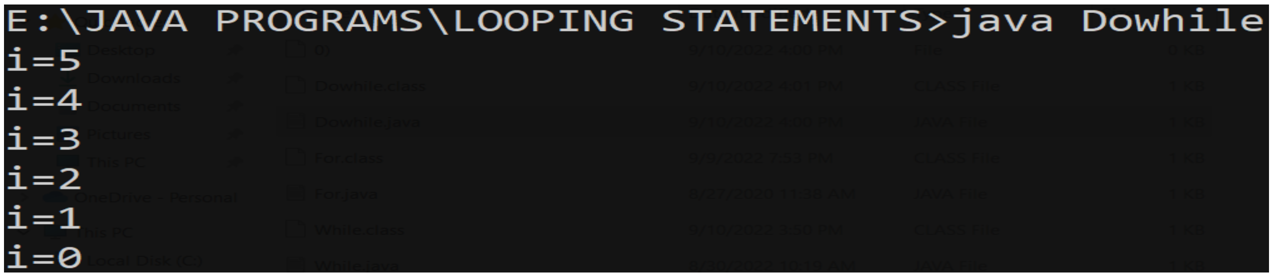
```
do
{
 // code block to be executed
}
while (condition);
```

1.6.2.3.2 Example Program using Do-while loop

```
class Dowhile
   {
    public static void main(String ab[])
   {
    int i;
    i=5;
    do
     {
     System.out.println("i="+i);
```

```
        i--;
        }while(i>=0);
      }
    }
```

1.6.2.3.3 Output



```
E:\JAVA PROGRAMS\LOOPING STATEMENTS>java Dowhile
i=5
i=4
i=3
i=2
i=1
i=0
```

1.6.2.4 Java for-each Loop

- In Java, the for-each loop is used to iterate through elements of arrays and collections
- It is also known as the enhanced for loop.

1.6.2.4.1 The syntax of the Java for-each loop is:

```
for(dataType item : array)
{
   ...
}
```

Here,

array - an array or a collection

item - each item of array/collection is assigned to this variable

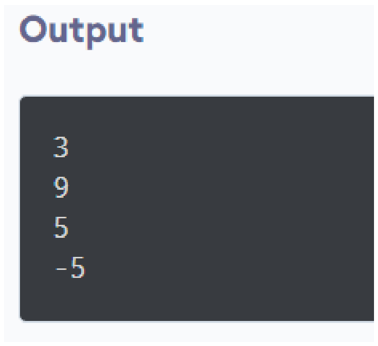
dataType - the data type of the array/collection

1.6.2.4.2 Example Program: Print Array Elements

```
class Main {
  public static void main(String[] args) {
    // create an array
    int[] numbers = {3, 9, 5, -5};
    // for each loop
    for (int number: numbers) {
      System.out.println(number);
    }
  }
```

}

1.6.2.4.3 Output


Output

```
3
9
5
-5
```

1.6.3 Jump Statements

- ➢ Jumping statements are control statements that transfer execution control from one point to another point in the program.
- ➢ There are two Jump statements that are provided in the Java programming language:
  - ✓ Break statement.
  - ✓ Continue statement.

1.6.3.1 Break Statement

- ➢ The break statement in Java terminates the loop immediately, and the control of the program moves to the next statement following the loop.

1.6.3.1.1 Syntax for Break Statement

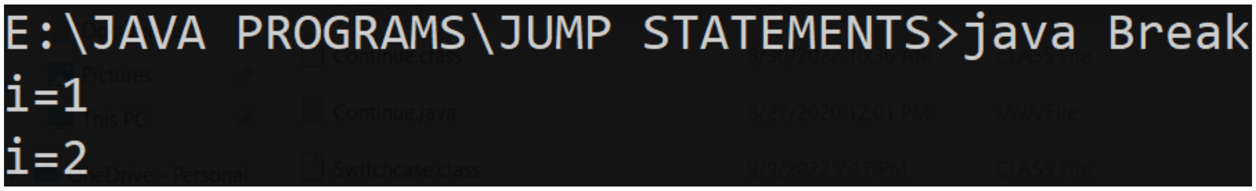
- ➢ break;

1.6.3.1.2 Simple Program using Break Statement

```
class Break
 {
   public static void main(String ar[])
  {
  int i;
```

```
            for(i=1;i<=5;i++)
              {
             if(i==3)
               break;
               System.out.println("i="+i);
               }
            }
          }
```

OUTPUT



```
E:\JAVA PROGRAMS\JUMP STATEMENTS>java Break
i=1
i=2
```

1.6.3.2 Continue Statement

- ➢ The continue statement is used when we want to skip a particular condition and continue the rest execution.

1.6.3.2.1 Syntax

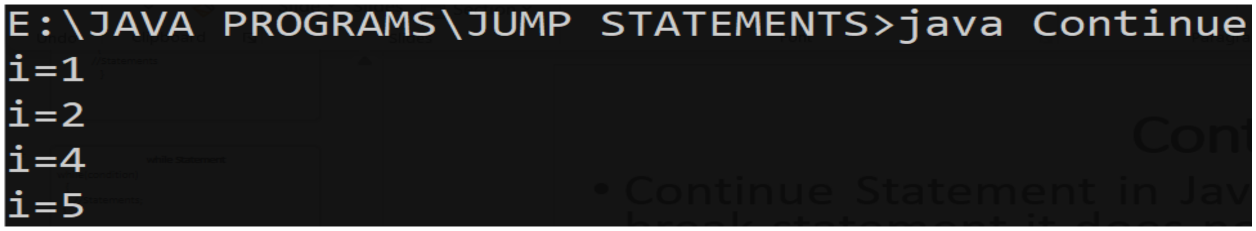
- ➢ continue;

1.6.3.2.2 Simple Program using Continue Statement

```
                class Continue
                   {
                    public static void main(String ar[])
                    {
                     int i;
                     for(i=1;i<=5;i++)
```

```
  {
 if(i==3)
  continue;
  System.out.println("i="+i);
    }
  }
 }
```

OUTPUT



```
E:\JAVA PROGRAMS\JUMP STATEMENTS>java Continue
i=1
i=2
i=4
i=5
```

1.9 PROGRAMMING STRUCTURES IN JAVA

- Java is an object-oriented programming, platform-independent, and secure programming language that makes it popular. Using the Java programming language, we can develop a wide variety of applications.

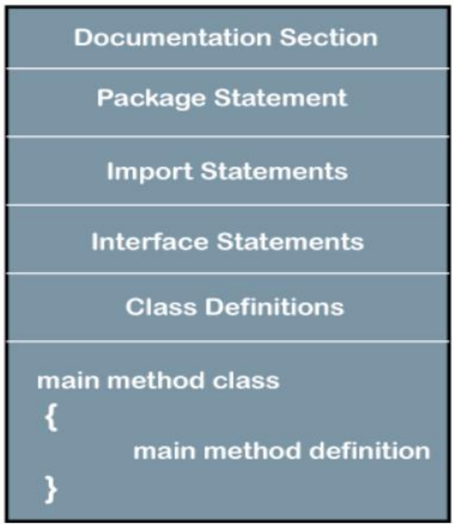


Fig: Structure of a JAVA Program

1.9.1 Documentation Section

- It includes basic information about a Java program. The information includes the author's name, date of creation, version, program name, company name, and description of the program.
- It improves the readability of the program. Whatever we write in the documentation section, the Java compiler ignores the statements during the execution of the program.
- To write the statements in the documentation section, we use comments. The comments may be **single-line, multi-line, and documentation comments**.
- **Single-line Comment:**
  - It starts with a pair of forwarding slash (//). For example: //First Java Program
- **Multi-line Comment:**
  - It starts with a /* and ends with */. We write between these two symbols. For example:
    /*It is an example of
    multiline comment*/
- **Documentation Comment:**
  - It starts with the delimiter (/**) and ends with */. For example:
    /**It is an example of documentation comment*/

1.9.2 Package Declaration

- The package declaration is optional. It is placed just after the documentation section.
- we declare the package name in which the class is placed.
- There can be only one package statement in a Java program.
- It must be defined before any class and interface declaration.
- The keyword **package** is used to declare the package name.
- Syntax
  - package package_name;
- Example:
  - package student; //where student is the package name
  - This statement declares that all the classes and interfaces defined in this source file are a part of the student package.

1.9.3 Import Statement

- The package contains the many predefined classes and interfaces. If we want to use any class of a particular package, we need to import that class.
- The import statement represents the class stored in the other package.
  - import java.util.Scanner; //it imports the Scanner class only
  - import java.util.*; //it imports all the class of the java.util package

1.9.4 Interface Section

- We use the interface keyword to create an interface.
- It contains only constants and method declarations.
- It cannot be instantiated.
- We can use interface in classes by using the implements keyword.
- Example

```
interface car
{
void start();
void stop();
}
```

1.9.5 Class Definition

- Without the class we cannot create any Java program. A Java program may conation more than one class definition.
- We use the class keyword to define the class. The class is a blueprint of a Java program.
- It contains information about user-defined methods, variables, and constants. Every Java program has at least one class that contains the main() method.
- Example

```
class Student //class definition
{
}
```

1.9.6 Class Variables and Constants

- In a Java program, the variables and constants are defined just after the class definition.
- It defines the life of the variables.
- Example

```
class Student //class definition
{
String sname;  //variable
int id;
double percentage;
}
```

1.9.7 Main Method Class

- Every Java stand-alone program requires the main method as the starting point of the program. This is an essential part of a Java program.
- There may be many classes in a Java program, and only one class defines the main method.
- When the main method is declared public, it means that it can be used outside of this class as well.
- The word static means that we want to access a method without making its objects. As we call the main method without creating any objects.
- The word void indicates that it does not return any value. The main is declared as void because it does not return any value.
- Main is the method, which is an essential part of any Java program.

1.9.7.1 String[] args

- It is an array where each element is a string, which is named as args. If you run the Java code through a console, you can pass the input parameter.
- The main() takes it as an input.

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

1.10 Defining classes in Java

- Java provides a reserved keyword class to define a class. The keyword must be followed by the class name. Inside the class, we declare methods and variables**.**
- In general, class declaration includes the following in the order as it appears:
  - **Modifiers:** A class can be public or has default access.
  - **class keyword:** The class keyword is used to create a class.
  - **Class name:** The name must begin with an initial letter (capitalized by convention).
  - **Superclass** (if any): The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
  - **Interfaces** (if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
  - **Body:** The class body surrounded by braces, { }

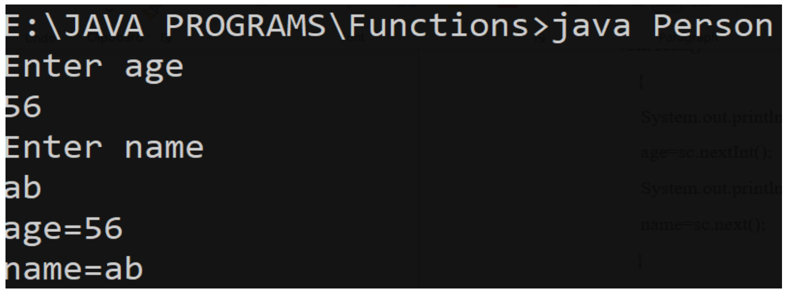
1.10.1 Syntax for Defining a Class

```
class class_name
{
// member variables
// class methods
}
```

1.10.2 Example Program using Classes, Objects and Methods

```
import java.util.Scanner;
class Person
  {
    int age;
    String name;
Scanner sc=new Scanner(System.in);
void read()
   {
    System.out.println("Enter age");
    age=sc.nextInt();
    System.out.println("Enter name");
    name=sc.next();
   }
void display()
   {
    System.out.println("age="+age);
    System.out.println("name="+name);
    }
 public static void main(String args[])
  {
   Person P=new Person();
   P.read();
   P.display();
  }
 }
```

OUTPUT



```
E:\JAVA PROGRAMS\Functions>java Person
Enter age
56
Enter name
ab
age=56
name=ab
```

1.11 JAVA CONSTRUCTORS

- A constructor in Java is a special method that is used to initialize objects. The constructor is called when an object of a class is created.

1.11.1 Types of Constructors

- Types of Constructors
  - Default Constructor
  - Parametrized Constructor

1.11.1.1 Default Constructor

- A constructor is called "Default Constructor" when it doesn't have any parameter.
- Syntax of default constructor:

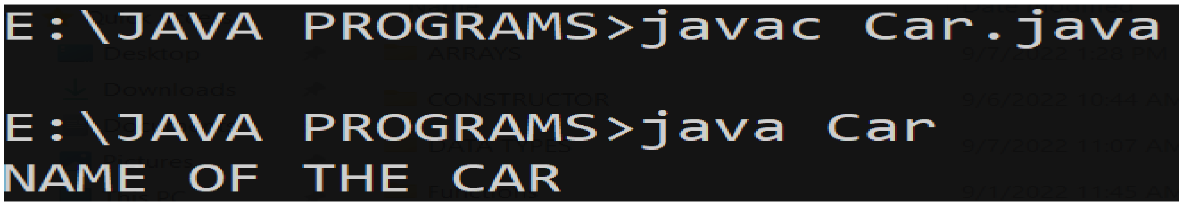
```
Class Classname
  {
  Classname()   //Default Constructor
      {
      Statement 1
      Statement 2
      --------------
       }
  }
```

1.11.1.2 Example of default constructor

```
class Car
  {
   Car()    //creating a default constructor
   {
    System.out.println("NAME OF THE CAR");
    }
    public static void main(String args[])
    {
```

```
                Car c=new Car();   //calling a default constructor
                }
                }
```

OUTPUT



```
E:\JAVA PROGRAMS>javac Car.java

E:\JAVA PROGRAMS>java Car
NAME OF THE CAR
```

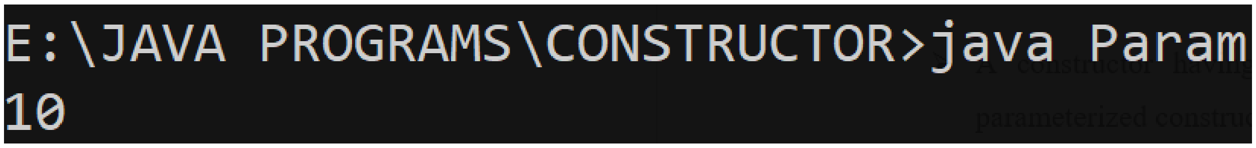
1.11.1.3 PARAMETRIZED CONSTRUCTOR

- A constructor having a specific number of parameters(arguments) is called a parameterized constructor.
- The parameterized constructor is used to provide different values to the objects, you can also provide the same values.

1.11.1.4 Example Program using Parametrized Constructor

```
class Param
 {
  int a;
  Param(int b)
   {
    a=b;
    System.out.println(a);
    }
 public static void main(String args[])
  {
   Param P=new Param(10);
  }
}
```

OUTPUT



```
E:\JAVA PROGRAMS\CONSTRUCTOR>java Param
10
```

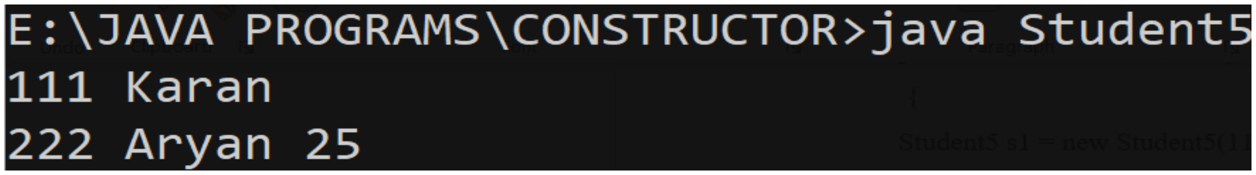
1.11.2 CONSTRUCTOR OVERLOADING

- The constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task.

1.11.2.1 Example Program using Constructor Overloading

```
class Student5
  {
  int id;
  String name;
  int age;
  //creating two arg constructor
  Student5(int i,String n)
 {
  id = i;
  name = n;
  }
  //creating three arg constructor
  Student5(int i,String n,int a)
  {
  id = i;
  name = n;
  age=a;
  }
  void display()
    {
    System.out.println(id+" "+name+" ");
    }
  void display1()
    {
    System.out.println(id+" "+name+" "+age);
    }
  public static void main(String args[])
   {
  Student5 s1 = new Student5(111,"Karan");
  Student5 s2 = new Student5(222,"Aryan",25);
  s1.display();
  s2.display1();
  }
}
```

OUTPUT



```
E:\JAVA PROGRAMS\CONSTRUCTOR>java Student5
111 Karan
222 Aryan 25
```

1.11.3 Properties of Constructor

- Constructors name must be similar to that of the class name inside which it resides.
- Constructors are automatically called when an object is created.
- Constructors cannot be private.
- A constructor can be overloaded.
- Constructors cannot return a value.
- Constructors do not have a return type; not even void.
- An interface cannot have the constructor.
- A constructor cannot be abstract, static, final, native, strictfp, or synchronized
- An abstract class can have the constructor.

1.12 METHODS

- A method is a block of code which only runs when it is called. You can pass data, known as parameters, into a method. Methods are used to perform certain actions, and they are also known as functions.
- It is used to achieve the reusability of code.

1.12.1 Types of Methods in JAVA

- In Java, there are two types of methods:
  - **User-defined Methods**: We can create our own method based on our requirements.
  - **Standard Library Methods**: These are built-in methods in Java that are available to use.

1.12.1.1 User-defined Methods:

- We can create our own method based on our requirements.
- The **syntax** to declare a method is:

```
returnType methodName()
{
  // method body
}
```

- Here,

  returnType - It specifies what type of value a method returns For example if a method has an int return type then it returns an integer value.

  If the method does not return a value, its return type is **void.**

  methodName - It is an identifier that is used to refer to the particular method in a program.

  method body - It includes the programming statements that are used to perform some tasks. The method body is enclosed inside the curly braces { }.

<u>Example</u>

```
int addNumbers()
{
// code
}
```

- ➢ In the above example, the name of the method is **adddNumbers().** The return type is **int**
- ➢ the complete syntax of declaring a method is

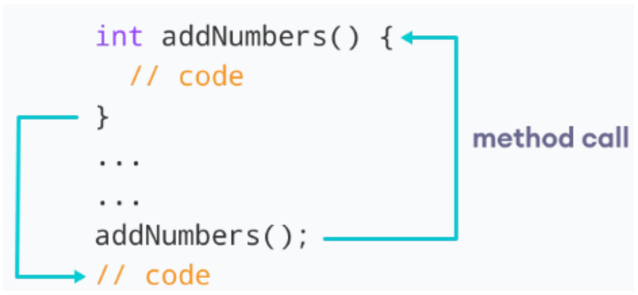
```
modifier static returnType nameOfMethod (parameter1, parameter2, ...)
{
  // method body
}
```

- ➢ Here,
  - **modifier -** It defines access types whether the method is public, private, and so on. To learn more, visit Java Access Specifier.
  - **static** - If we use the static keyword, it can be accessed without creating objects.
    - ✓ For example, the sqrt() method of standard Math class is static. Hence, we can directly call Math.sqrt() without creating an instance of Math class.
  - **parameter1/parameter2 -** These are values passed to a method. We can pass any number of arguments to a method.

1.12.1.2 Calling a Method in Java

- ➢ In the above example, we have declared a method named addNumbers(). Now, to use the method, we need to call it.
- ➢ // calls the method

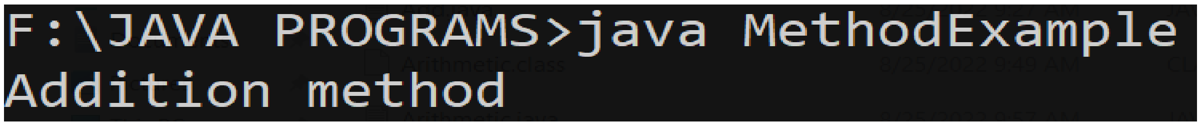
```
addNumbers();
```


```
int addNumbers() {
  // code
}
...
...
addNumbers();
// code
```

1.12.1.3 Method with no parameter and no return type

```
public class MethodExample
{
public void add()
{
System.out.println("Addition method");
 }
public static void main(String[] args) {
MethodExample m = new MethodExample();
m.add();
}
}
```

OUTPUT

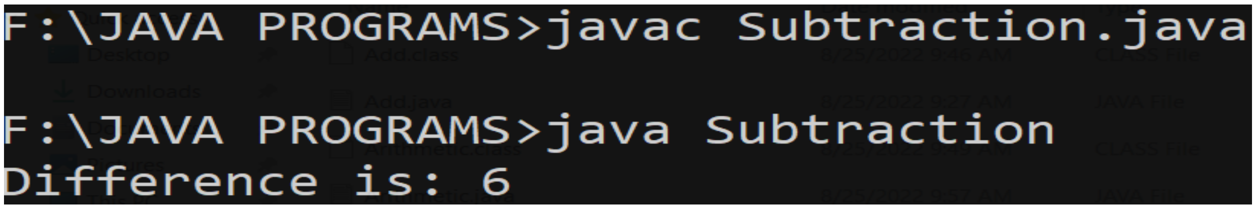

```
F:\JAVA PROGRAMS>java MethodExample
Addition method
```

1.12.1.4 Method with parameters but no return type

```
public class Subtraction {
public void difference(int x, int y)
{
int diff = x - y;
System.out.println("Difference is: " + diff);
```

```
 }
 public static void main(String[] args) {
 Subtraction s = new Subtraction();
 s.difference(10, 4);
 }
}
```

OUTPUT

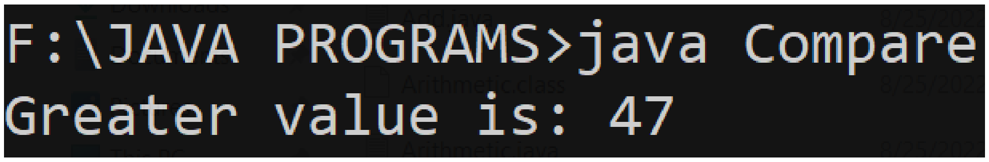


```
F:\JAVA PROGRAMS>javac Subtraction.java

F:\JAVA PROGRAMS>java Subtraction
Difference is: 6
```

1.12.1.5 Method with parameter and return type

```
public class Compare
{
 int max;
 public int greaterNumber(int x, int y) {
 if(x > y)
      max = x;
    else
      max = y;
    return max;
  }
  public static void main(String[] args) {
    Compare c = new Compare();
    int value = c.greaterNumber(47, 29);
    System.out.println("Greater value is: " + value);
  }
}
```

OUTPUT



```
F:\JAVA PROGRAMS>java Compare
Greater value is: 47
```

- ➢ In this example, the method greaterNumber has both parameters and return value. Hence when calling the method, we should pass arguments as well as have a variable to assign the return value from the method.
- ➢ In this case, x and y values have 47 and 29 respectively and return the variable max from the method. While invoking the method, we have the variable value that receives the output of this method.

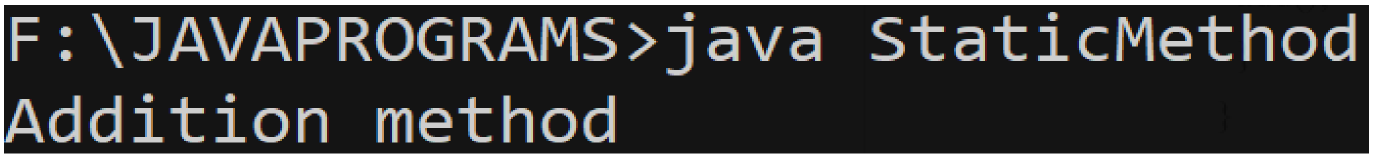
1.12.1.6 Static Java Method

- ➢ Static methods are the methods in Java that can be called without creating an object of class. They are referenced by the class name itself or reference to the Object of that class.

1.12.1.6.1 Example Program using Static Java Method

```
public class StaticMethod
{
 public static void add()
{
 System.out.println("Addition method");
 }
 public static void main(String[] args) {
 add();
 }
}
```

OUTPUT



```
F:\JAVAPROGRAMS>java StaticMethod
Addition method
```

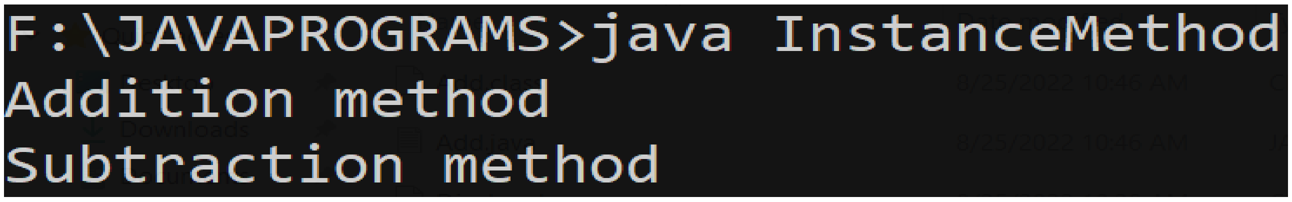
1.12.1.7 Instance Methods

- Methods which are non-static and that belongs to a class is called an instance method. These methods require an object or class instance to make a call to the function.

1.12.1.7.1 Example Program using Instance Method

```
public class MethodExample
 {
  public static void add()     //Static method
  {
    System.out.println("Addition method");
  }
  public void subtract()     //Instance method
  {
    System.out.println("Subtraction method");
  }
  public static void main(String[] args)
  {
    MethodExample m = new MethodExample();      //Invoking a static method
    add();
    m.subtract();      //Invoking an instance method
  }
 }
```

OUTPUT



```
F:\JAVAPROGRAMS>java InstanceMethod
Addition method
Subtraction method
```

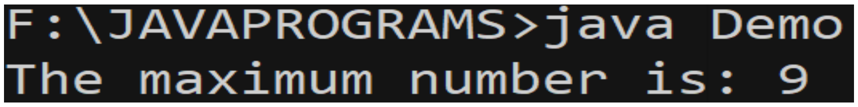
1.12.2 Predefined Method

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

1.12.2.1 Example Program for Predefined method in JAVA

```
public class Demo
{
public static void main(String[] args)
{
// using the max() method of Math class
System.out.print("The maximum number is: " + Math.max(9,7));
}
}
```

OUTPUT



```
F:\JAVAPROGRAMS>java Demo
The maximum number is: 9
```

- In the above example, we have used three predefined methods main(), print(), and max(). We have used these methods directly without declaration because they are predefined.
- The print() method is a method of PrintStream class that prints the result on the console.
- The max() method is a method of the Math class that returns the greater of two numbers.

1.13 Access specifiers

- Access specifiers in Java helps to restrict the scope of a class, constructor, variable, method or data member.
- Access specifiers can be specified separately for a class, constructors, fields, and methods.
- They are also referred as Java Access Modifiers

1.13.1 Types of Access Modifier

- Default Access Modifier
- Private Access Modifier
- Public Access Modifier
- Protected Access Modifier

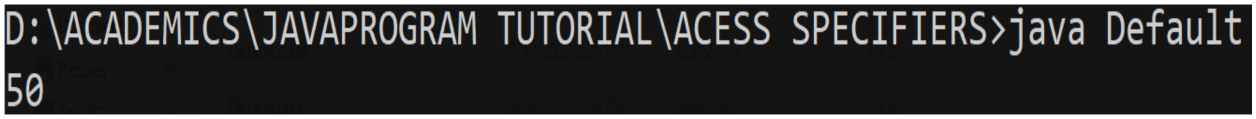
1.13.1.1 Default Access Modifier

- When no access modifier is specified for a particular class, method or a data member, it is said to be having the default access modifier.

1.13.1.1.1 Example Program using Default Access

```
class DefaultEx
{
    int x=50; // default data
}
class Default
{
    public static void main(String[] args)
    {
            DefaultEx  a1=new DefaultEx();
            System.out.println(a1.x); // default data x is accessible outside the class
    }
}
```

OUTPUT



```
D:\ACADEMICS\JAVAPROGRAM TUTORIAL\ACESS SPECIFIERS>java Default
50
```

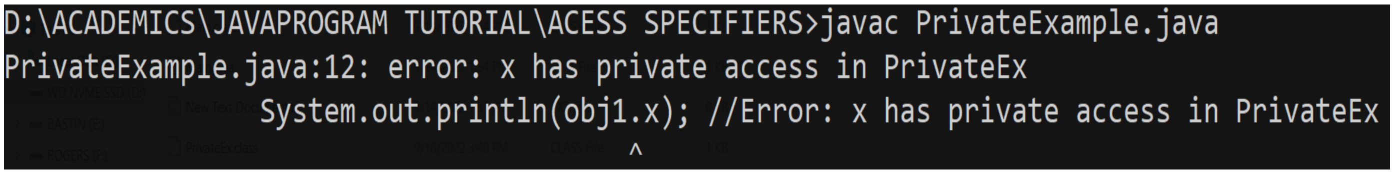
1.13.1.2 Private Access Modifier

- The methods or data members that are declared as private are only accessible within the class in which they are declared.

1.13.1.2.1 Example Program using Private Access Modifier

```
class PrivateEx
{
        private int x=5; // private data
        public int y=10; // public data
}
public class PrivateExample
{
        public static void main(String[] args)
        {
        PrivateEx obj1=new PrivateEx();
        System.out.println(obj1.y); // public data y is accessible by a non-member
        System.out.println(obj1.x); //Error: x has private access in PrivateEx
        }
}
```

OUTPUT



```
D:\ACADEMICS\JAVAPROGRAM TUTORIAL\ACESS SPECIFIERS>javac PrivateExample.java
PrivateExample.java:12: error: x has private access in PrivateEx
                System.out.println(obj1.x); //Error: x has private access in PrivateEx
                                       ^
```

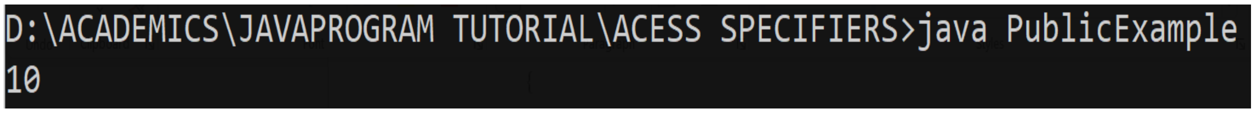
1.13.1.3 Public Access Modifier

- The public access modifier is specified using the keyword public.
- Classes, methods or data members which are declared as public are accessible anywhere throughout the program. There is no restriction on the scope of public data members.

1.13.1.3.1 Example Program using Public Access Modifier

```
class PublicEx
{
      public int no=10;
     public void dis()
        {
         System.out.println(no);
        }
}
public class PublicExample
{
      public static void main(String[] args)
      {
                  PublicEx obj=new PublicEx();
                  obj.dis();
      }
}
```

OUTPUT



```
D:\ACADEMICS\JAVAPROGRAM TUTORIAL\ACESS SPECIFIERS>java PublicExample
10
```

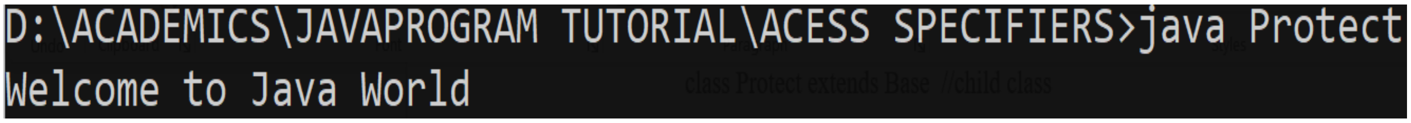
1.13.1.4 Protected Access Modifier

- The protected access modifier is specified using the keyword protected.
- Protected members can be accessed only in the child or derived classes.

1.13.1.3.1 Example Program using Protected Access Modifier

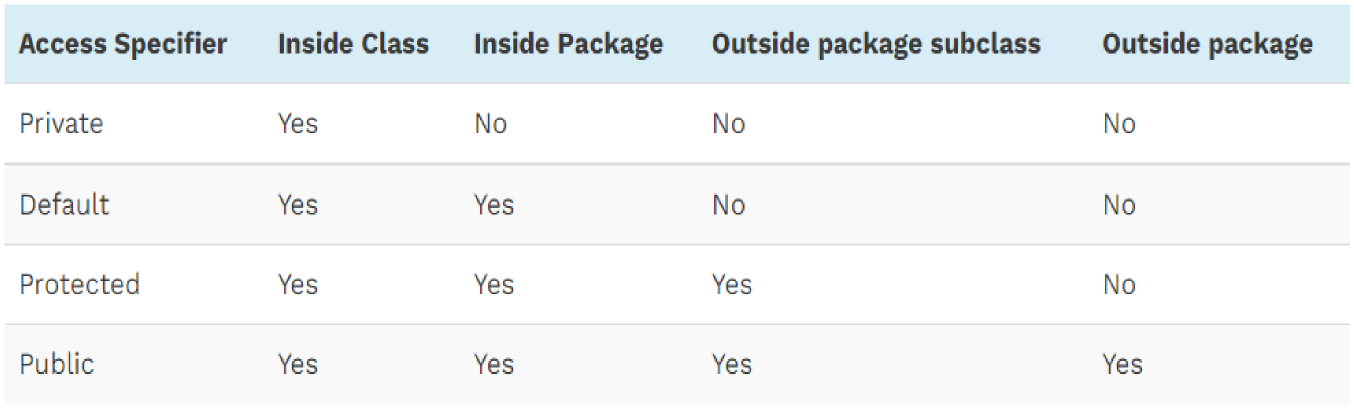
```
class Base    //parent class
{
        protected void show()
        {
                System.out.println("Welcome to Java World");
        }
}
class Protect extends Base  //child class
{
        public static void main(String[] args)
        {
                Protect obj=new Protect();
                obj.show();    //fncall using object
        }
}
```

OUTPUT



```
D:\ACADEMICS\JAVAPROGRAM TUTORIAL\ACESS SPECIFIERS>java Protect
Welcome to Java World
```

1.13.2 Access Rights of Different Access modifiers



Access Specifier	Inside Class	Inside Package	Outside package subclass	Outside package
Private	Yes	No	No	No
Default	Yes	Yes	No	No
Protected	Yes	Yes	Yes	No
Public	Yes	Yes	Yes	Yes

1.14 STATIC MEMBERS

- When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.
- Variables and methods declared using keyword static are called static members of a class.

1.14.1 Static Method

- Static method in Java is a method which belongs to the class and not to the object.
- A static method can access only static data. It is a method which belongs to the class and not to the object(instance).
- A static method can access only static data. It cannot access non-static data (instance variables).
- A static method can be accessed directly by the class name and doesn't need any object
- A static method cannot refer to "this" or "super" keywords in anyway.

1.14.1.1 Syntax

<class-name>.<method-name>

1.14.2 Static variables

- When a variable is declared as static, then a single copy of the variable is created and shared among all objects at the class level.
- Static variables are, essentially, global variables. All instances of the class share the same static variable.
- A static variable can be accessed directly by the class name and doesn't need any object

1.14.2.1 Syntax:

<class-name>.<variable-name>

1.14.3 Static Block

- A static block is a set of instructions that will be executed only once when a class is loaded into memory. A static block is also called a static initialization block.

1.14.3.1 Syntax

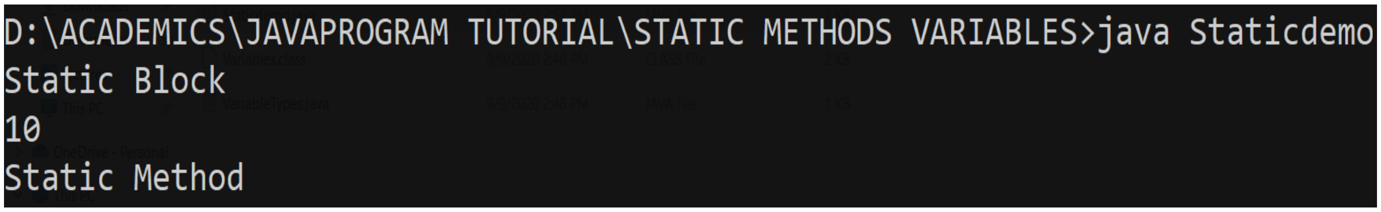
```
class Test
{
 static
{
 //Code goes here
 }
}
```

1.14.4 Example Program to illustrate the use of Static Variables, Methods and Static Blocks

```
class Staticdemo

{

    static int a=10;  //STATIC VARIABLE

    static void display()  //STATIC METHOD

         {

        System.out.println("Static Method");

         }

    static  //STATIC BLOCK

        {

        System.out.println("Static Block");

        }

      public static void main(String args[])

      {

       Staticdemo obj=new Staticdemo();

       System.out.println(obj.a);

       display();
```

```
        }
    }
```

Output



```
D:\ACADEMICS\JAVAPROGRAM TUTORIAL\STATIC METHODS VARIABLES>java Staticdemo
Static Block
10
Static Method
```

1.15 JAVA DOC COMMENTS

- Javadoc is a tool that generates Java code documentation in the HTML format from Java source code. The documentation is formed from Javadoc comments that are usually placed above classes, methods, or fields.
- Following is a simple example where the lines inside /*….*/ are Java multi-line comments. Similarly, the line which proceeds // is Java single-line comment.

Example 1

```
/**
* The HelloWorld program implements an application that
* simply displays "Hello World!" to the standard output.
* @author  Bastin
* @version 1.0
* @since   2014-03-31
*/
public class HelloWorld
 {
  public static void main(String[] args)
     {
    // Prints Hello, World! on standard output.
    System.out.println("Hello World!");
     }
}
```

- You can include required HTML tags inside the description part. For instance, the following example makes use of <h1>....</h1> for heading and <p> has been used for creating paragraph break.

Example 2

```
/**
* <h1>Hello, World!</h1>
* The HelloWorld program implements an application that
* simply displays "Hello World!" to the standard output.
* <p>
* Giving proper comments in your program makes it more
* user friendly and it is assumed as a high quality code.
* @author  Bastin
* @version 1.0
* @since   2014-03-31
*/
public class HelloWorld {
  public static void main(String[] args) {
    // Prints Hello, World! on standard output.
    System.out.println("Hello World!");
  }
}
```

Output : Hello World

1.15.1 The javadoc Tags

- The javadoc tool recognizes the following tags

Tag	Description	Syntax
@author	Adds the author of a class.	@author name-text
{@code}	Displays text in code font without interpreting the text as HTML markup or nested javadoc tags.	{@code text}
{@docRoot}	Represents the relative path to the generated document's root directory from any generated page.	{@docRoot}
@deprecated	Adds a comment indicating that this API should no longer be used.	@deprecated deprecatedtext
@exception	Adds a **Throws** subheading to the generated documentation, with the classname and description text.	@exception class-name description
{@inheritDoc}	Inherits a comment from the **nearest** inheritable class or implementable interface.	Inherits a comment from the immediate surperclass.
{@link}	Inserts an in-line link with the visible text label that points to the documentation for the specified package, class, or member name of a referenced class.	{@link package.class#member label}
{@linkplain}	Identical to {@link}, except the link's label is displayed in plain text than code font.	{@linkplain package.class#member label}
@param	Adds a parameter with the specified parameter-name followed by the specified description to the "Parameters" section.	@param parameter-name description
@return	Adds a "Returns" section with the description text.	@return description
@see	Adds a "See Also" heading with a link or text entry that points to reference.	@see reference
@serial	Used in the doc comment for a default serializable field.	@serial field-description | include | exclude
@serialData	Documents the data written by the writeObject( ) or writeExternal( ) methods.	@serialData data-description

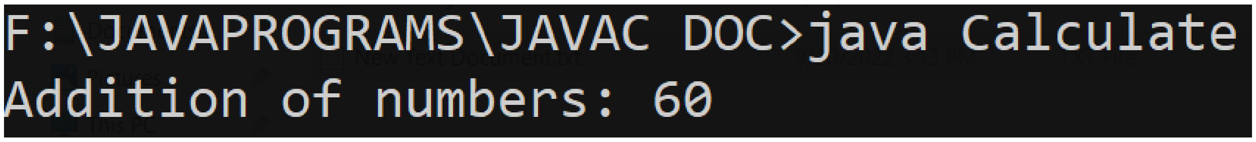
@serialField	Documents an ObjectStreamField component.	@serialField field-name field-type field-description
@since	Adds a "Since" heading with the specified since-text to the generated documentation.	@since release
@throws	The @throws and @exception tags are synonyms.	@throws class-name description
{@value}	When {@value} is used in the doc comment of a static field, it displays the value of that constant.	{@value package.class#field}
@version	Adds a "Version" subheading with the specified version-text to the generated docs when the -version option is used.	@version version-text

Example 3

```
import java.io.*;

/**

 * <h2> Calculation of numbers </h2>

 * This program implements an application

 * to perform operation such as addition of numbers

 * and print the result

 * <p>

 * <b>Note:</b> Comments make the code readable and

 * easy to understand.

 *

 * @author Anurati

 * @version 16.0

 * @since 2021-07-06
```

```
*/
public class Calculate{
  /**
   * This method calculates the summation of two integers.
   * @param input1 This is the first parameter to sum() method
   * @param input2 This is the second parameter to the sum() method.
   * @return int This returns the addition of input1 and input2
   */
   public int sum(int input1, int input2){
     return input1 + input2;
  }
  /**
  * This is the main method uses of sum() method.
  * @param args Unused
  * @see IOException
  */
  public static void main(String[] args) {
     Calculate obj = new Calculate();
     int result = obj.sum(40, 20);
     System.out.println("Addition of numbers: " + result);
  }
}
```

OUTPUT



```
F:\JAVAPROGRAMS\JAVAC DOC>java Calculate
Addition of numbers: 60
```

INHERITANCE, PACKAGES, AND INTERFACES

Overloading Methods – Objects as Parameters – Returning Objects – Static, Nested and Inner Classes. Inheritance: Basics– Types of Inheritance -Super keyword -Method Overriding – Dynamic Method Dispatch –Abstract Classes – final with Inheritance. Packages and Interfaces: Packages – Packages and Member Access –Importing Packages – Interfaces.

2.1 OVERLOADING METHODS

2.1.1 What is Method Overloading?

- Method overloading in java is a feature that allows a class to have more than one method with the same name, but with different parameters.

2.1.2 Method Overloading Types

- Overloading by changing the number of parameters.
- Method Overloading by changing the data type of parameters

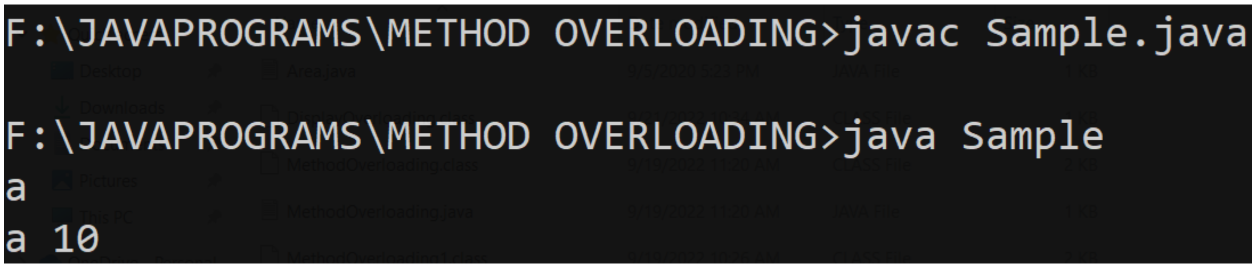
2.1.2.1 Overloading by changing the number of parameters.

- This example shows how method overloading is done by having different number of parameters. In this example, we have two methods with the same name but their parameters count is different.
- First disp() method has one parameter (char) while the second method disp() has two parameters (char, int).
- **Example Program**

```
class DisplayOverloading
{
   public void disp(char c)
   {
       System.out.println(c);
   }
```

```
         public void disp(char c, int num)
         {
             System.out.println(c + " "+num);
         }
     }
     class Sample
     {
        public static void main(String args[])
        {
            DisplayOverloading obj = new DisplayOverloading();
            obj.disp('a');
            obj.disp('a',10);
        }
     }
```

OUTPUT



```
F:\JAVAPROGRAMS\METHOD OVERLOADING>javac Sample.java

F:\JAVAPROGRAMS\METHOD OVERLOADING>java Sample
a
a 10
```

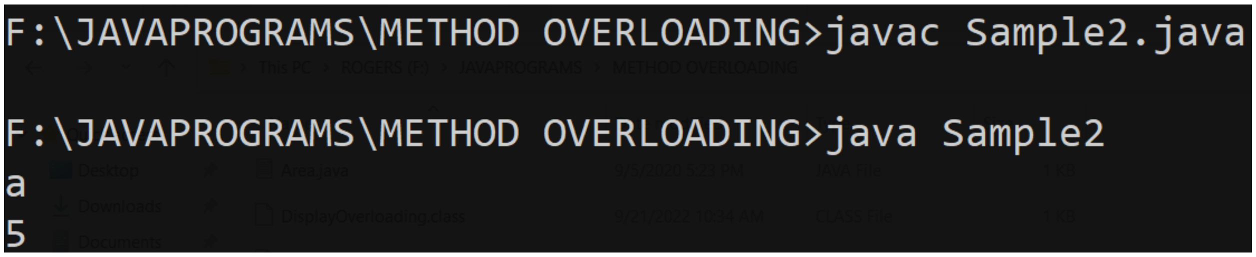
2.1.2.2 Method Overloading by changing the data type of parameters

- In this example, method disp() is overloaded based on the data type of parameters .
- We have two methods with the name disp() and number of parameters is same but the type of parameters is different.
- The first method has one char parameter while the second method has one int parameter.

- **Example Program**

```
class DisplayOverloading2
{
   public void disp(char c)
   {
      System.out.println(c);
   }
   public void disp(int c)
   {
     System.out.println(c );
   }
}
class Sample2
{
   public static void main(String args[])
   {
      DisplayOverloading2 obj = new DisplayOverloading2();
      obj.disp('a');
      obj.disp(5);
   }
}
```

OUTPUT

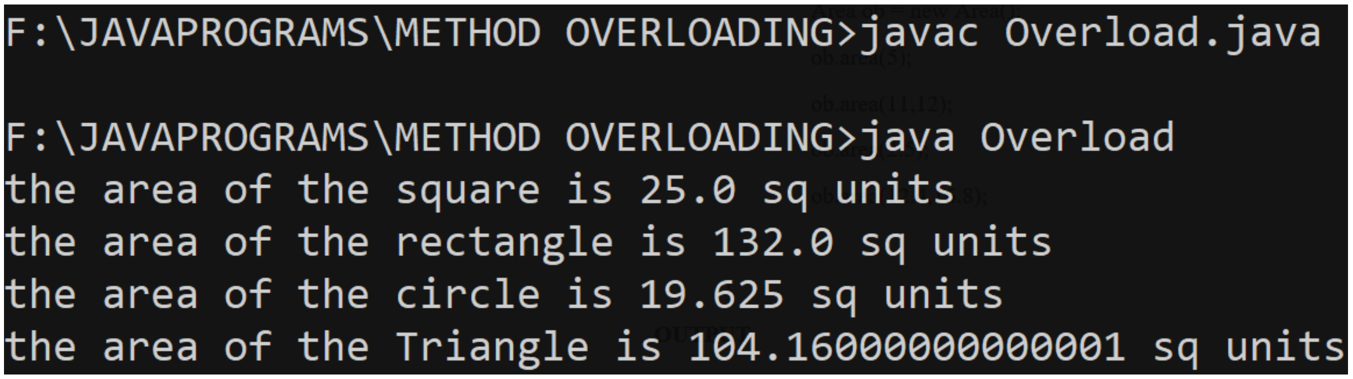


```
F:\JAVAPROGRAMS\METHOD OVERLOADING>javac Sample2.java

F:\JAVAPROGRAMS\METHOD OVERLOADING>java Sample2
a
5
```

2.1.3 Example Program to calculate area of Square, Circle, Rectangle and Triangle using Method Overloading

```
class Area
{
    void area(float x)
    {
        System.out.println("the area of the square is "+Math.pow(x, 2)+" sq units");
    }
    void area(float x, float y)
    {
        System.out.println("the area of the rectangle is "+x*y+" sq units");
    }
    void area(double x)
    {
        double z = 3.14 * x * x;
        System.out.println("the area of the circle is "+z+" sq units");
    }
    void area(double x,double y)
     {
      double z=((x*y)/2);
      System.out.println("the area of the Triangle is "+z+" sq units");
     }
}
class Overload
{
     public static void main(String args[])
            {
                Area ob = new Area();
                ob.area(5);
                ob.area(11,12);
                ob.area(2.5);
                ob.area(12.4,16.8);
        }
    }
```

OUTPUT



```
F:\JAVAPROGRAMS\METHOD OVERLOADING>javac Overload.java

F:\JAVAPROGRAMS\METHOD OVERLOADING>java Overload
the area of the square is 25.0 sq units
the area of the rectangle is 132.0 sq units
the area of the circle is 19.625 sq units
the area of the Triangle is 104.16000000000001 sq units
```

2.2 OBJECTS AS PARAMETERS

- ➢ Object as an argument is use to establish communication between two or more objects of same class as well as different class, i.e, user can easily process data of two same or different objects within function.

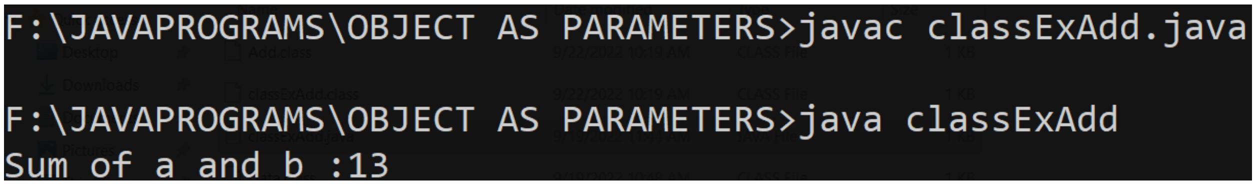
2.2.1 Passing Object as Parameter in Function

- ➢ While creating a variable of class type, we only create a reference to an object.
- ➢ When we pass this reference to a function, the parameters that receive it will refer to the same object as that referred to by the argument.

Example Program

```
class Add
{
        int a;
        int b;
        Add (int x,int y)// parametrized constructor
        {
                a=x;
                b=y;
        }
        void sum(Add A1) // object  'A1' passed as parameter in function 'sum'
        {
                int sum1=A1.a+A1.b;
                System.out.println("Sum of a and b :"+sum1);
        }
}

public class classExAdd
{
        public static void main(String arg[])
        {
                Add A=new Add(5,8);
                /* Calls  the parametrized constructor
                with set of parameters*/
                A.sum(A);
        }
}
```

OUTPUT



```
F:\JAVAPROGRAMS\OBJECT AS PARAMETERS>javac classExAdd.java

F:\JAVAPROGRAMS\OBJECT AS PARAMETERS>java classExAdd
Sum of a and b :13
```

2.2.2 Passing Object as Parameter in Constructor

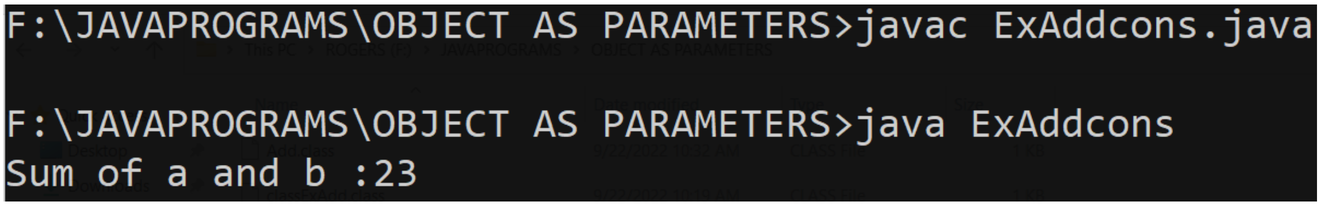
- One of the most common uses of objects as parameters involves constructors. A constructor creates a new object initially the same as passed object.
- It is also used to initialize private members.

Example Program

```
class Add
{
        private int a,b;
        Add(Add A)
        {
                a=A.a;
                b=A.b;
        }
        Add(int x,int y)
        {
                a=x;
                b=y;
        }
        void sum()
        {
                int sum1=a+b;
                System.out.println("Sum of a and b :"+sum1);
        }
}
```

```
        class ExAddcons
        {
                public static void main(String arg[])
                {
                        Add A=new Add(15,8);
                        Add A1=new Add(A);
                        A1.sum();
                }
        }
```

OUTPUT



```
F:\JAVAPROGRAMS\OBJECT AS PARAMETERS>javac ExAddcons.java

F:\JAVAPROGRAMS\OBJECT AS PARAMETERS>java ExAddcons
Sum of a and b :23
```

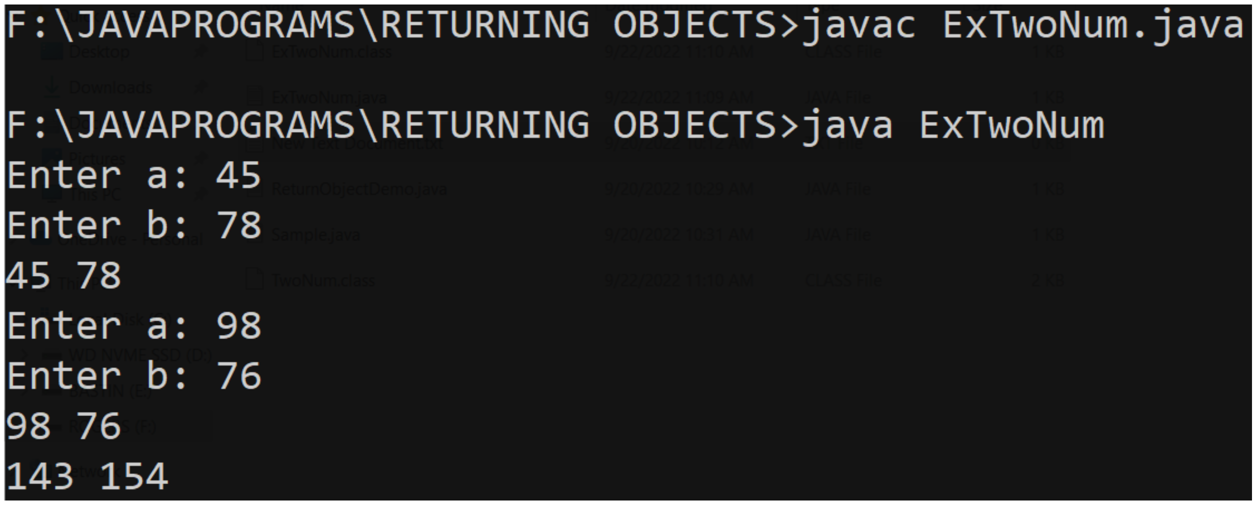
2.3 Returning the Object from Function

- In java, a function can return any type of data, including class type objects.
- For ex: In the program given below, the add() function return an object which contain sum of values of two different Numbers(objects).

Example Program

```
import java.util.Scanner;
class TwoNum
{
        private int a,b;
        Scanner kb=new Scanner(System.in);
        void getValues()                    // getValues() take values of a,b for every no.
        {
                System.out.print("Enter a: ");
                a=kb.nextInt();
                System.out.print("Enter b: ");
                b=kb.nextInt();
        }
        void putValues()                    // putValues() show values for every no.
        {
                System.out.println(a+" "+b);
        }
        TwoNum add(TwoNum B)        /*class type function add() takeobject 'B' as parameter*/
        {
                TwoNum D=new TwoNum();//object D act as instance variable
                D.a=a+B.a;
                D.b=b+B.b;
                return (D);//returning object D
        }
 }
class ExTwoNum
{
        public static void main(String arg[])
        {
                TwoNum A=new TwoNum();
                A.getValues();
                A.putValues();
                TwoNum B=new TwoNum();
                B.getValues();
                B.putValues();
                TwoNum C;
                /*object A calls add() passing object B
                as parameter and result are return at C*/
                C=A.add(B);
                C.putValues();
        }
}
```

OUTPUT



```
F:\JAVAPROGRAMS\RETURNING OBJECTS>javac ExTwoNum.java

F:\JAVAPROGRAMS\RETURNING OBJECTS>java ExTwoNum
Enter a: 45
Enter b: 78
45 78
Enter a: 98
Enter b: 76
98 76
143 154
```

2.4 STATIC, NESTED AND INNER CLASSES

- Java inner class or nested class is a class that is declared inside the class or interface.
- We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.
- Additionally, it can access all the members of the outer class, including private data members and methods.

Syntax of Inner class

```
class Java_Outer_class
```

```
{
 //code
 class Java_Inner_class{
  //code
 }
}
```

Advantage of Java inner classes

- ➢ Nested classes represent a particular type of relationship that is it can access all the members (data members and methods) of the outer class, including private.
- ➢ Nested classes are used to develop more readable and maintainable code because it logically group classes and interfaces in one place only.
- ➢ Code Optimization: It requires less code to write.

Need of Java Inner class

- ➢ Sometimes users need to program a class in such a way so that no other class can access it. Therefore, it would be better if you include it within other classes.

Difference between nested class and inner class in Java

- ➢ An inner class is a part of a nested class. Non-static nested classes are known as inner classes.

2.4.1 Types of Nested classes

- ➢ There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.
  - **Non-static nested class (inner class)**
    - ✓ Member inner class
    - ✓ Anonymous inner class
    - ✓ Local inner class

- **Static nested class**

2.4.1.1 Non-Static nested class (inner class)

- A non-static nested class is a class within another class. It has access to members of the enclosing class (outer class). It is commonly known as inner class.
- Since the inner class exists within the outer class, you must instantiate the outer class first, in order to instantiate the inner class.

2.4.1.1.1 Member inner class

- A non-static class that is created inside a class but outside a method is called member inner class. It is also known as a regular inner class. It can be declared with access modifiers like public, default, private, and protected.

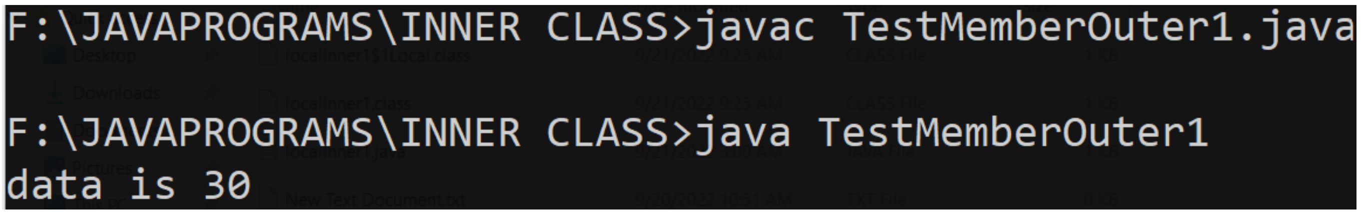
  Syntax:

```
class Outer
{
 //code
 class Inner
{
  //code
 }
}
```

Example Program for Member Inner Class

```
class TestMemberOuter1  //Outer Class
{
        private int data=30;  //Private Variable
        class Inner          //Inner Class
        {
                void msg()
                {
                        System.out.println("data is "+data);
                }
        }
        public static void main(String args[])
        {
                TestMemberOuter1 obj=new TestMemberOuter1();
                TestMemberOuter1.Inner in=obj.new Inner();
                in.msg();
        }
}
```

OUTPUT



```
F:\JAVAPROGRAMS\INNER CLASS>javac TestMemberOuter1.java

F:\JAVAPROGRAMS\INNER CLASS>java TestMemberOuter1
data is 30
```

2.4.1.1.2 Anonymous inner class

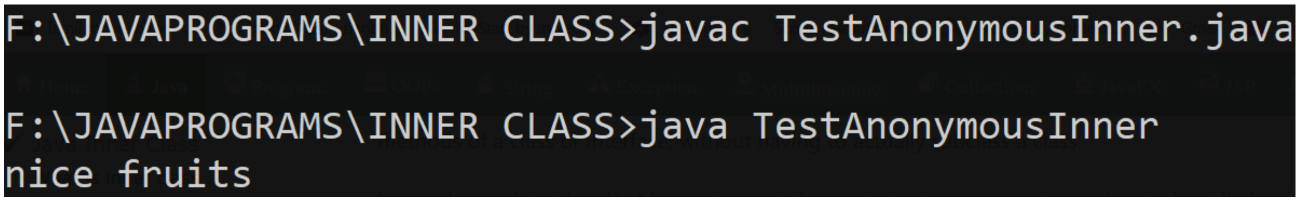
- ➢ Java anonymous inner class is an inner class without a name and for which only a single object is created.
- ➢ It should be used if you have to override a method of class or interface. Java Anonymous inner class can be created in two ways:
    - Class (may be abstract or concrete).
    - Interface

2.4.1.1.2.1 Java anonymous inner class example using class

```
        abstract class Person
        {
```

```
abstract void eat();
}

class TestAnonymousInner
 {
 public static void main(String args[])
  {
  Person p=new Person()
  {
  void eat()
   {
   System.out.println("nice fruits");
   }
   };
  p.eat();
  }
}
```

OUTPUT



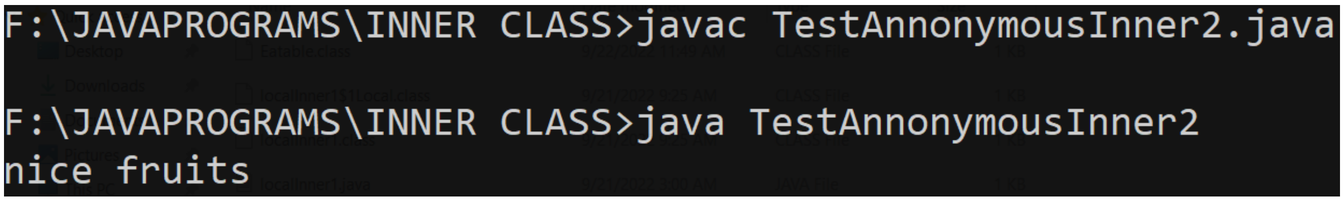
```
F:\JAVAPROGRAMS\INNER CLASS>javac TestAnonymousInner.java

F:\JAVAPROGRAMS\INNER CLASS>java TestAnonymousInner
nice fruits
```

- A class is created, but its name is decided by the compiler, which extends the Person class and provides the implementation of the eat() method.
- An object of the Anonymous class is created that is referred to by 'p,' a reference variable of Person type.

2.4.1.1.2.2 Java anonymous inner class example using Interface

```
interface Eatable
{
 void eat();
}
 class TestAnnonymousInner1
 {
 public static void main(String args[])
{
 Eatable e=new Eatable()
{
  public void eat(){System.out.println("nice fruits");}
 };
 e.eat();
 }
}
```

OUTPUT



```
F:\JAVAPROGRAMS\INNER CLASS>javac TestAnnonymousInner2.java

F:\JAVAPROGRAMS\INNER CLASS>java TestAnnonymousInner2
nice fruits
```

- It performs two main tasks behind this code:

- A class is created, but its name is decided by the compiler, which implements the Eatable interface and provides the implementation of the eat() method.
- An object of the Anonymous class is created that is referred to by 'p', a reference variable of the Eatable type.

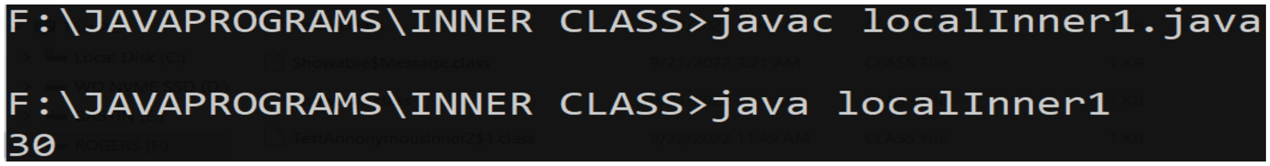
2.4.1.1.3 Java Local inner class

- A class created inside a method, is called local inner class in java. Local Inner Classes are the inner classes that are defined inside a block. Generally, this block is a method body.
- Local Inner classes are not a member of any enclosing classes.They belong to the block they are defined within, due to which local inner classes cannot have any access modifiers associated with them.
- They can be marked as final or abstract.
- These classes have access to the fields of the class enclosing it.

2.4.1.1.3.1 Example Program using Java Local inner class

```
public class localInner1
  {
   private int data=30;//instance variable
    void display()
         {
          class Local
          {
          void msg()
          {
           System.out.println(data);
           }
         }
         Local l=new Local();
         l.msg();
         }
        public static void main(String args[])
         {
         localInner1 obj=new localInner1();
         obj.display();
        }
       }
```

OUTPUT



```
F:\JAVAPROGRAMS\INNER CLASS>javac localInner1.java

F:\JAVAPROGRAMS\INNER CLASS>java localInner1
30
```

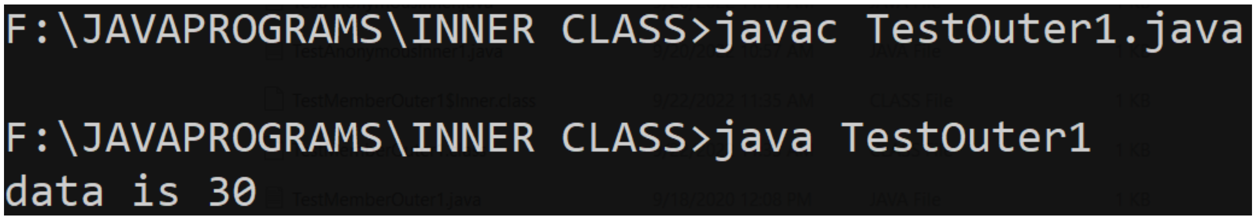
2.4.2 Java static nested class

- A static class is a class that is created inside a class, is called a static nested class in Java.
- It cannot access non-static data members and methods. It can be accessed by outer class name.
- It can access static data members of the outer class, including private.

Example Program

```
class TestOuter1
 {
 static int data=30;
 static class Inner
 {
  void msg()
   {
    System.out.println("data is "+data);}
    }
  public static void main(String args[])
   {
   TestOuter1.Inner obj=new TestOuter1.Inner();
 obj.msg();
 }
 }
```

OUTPUT



```
F:\JAVAPROGRAMS\INNER CLASS>javac TestOuter1.java

F:\JAVAPROGRAMS\INNER CLASS>java TestOuter1
data is 30
```

2.4.3 Java Nested Interface

- An interface, i.e., declared within another interface or class, is known as a nested interface.
- The nested interfaces are used to group related interfaces so that they can be easy to maintain.
- The nested interface must be referred to by the outer interface or class. It can't be accessed directly.
    - The nested interface must be public if it is declared inside the interface, but it can have any access modifier if declared within the class.

Syntax of nested interface which is declared within the interface

```
interface interface_name
{
...
interface nested_interface_name
{
  ...
 }
 }
```

Syntax of nested interface which is declared within the class

```
class class_name
```

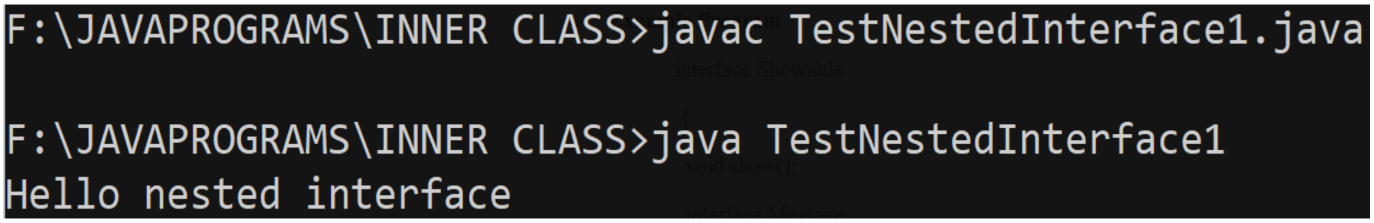
```
{
 ...
 interface nested_interface_name
{
  ...
 }
}
```

Example Program

```
interface Showable
 {
  void show();
  interface Message
  {
   void msg();
  }
}
class TestNestedInterface1 implements Showable. Message
 {
 public void msg()
  {
  System.out.println("Hello nested interface");
  }
 public static void main(String args[])
  {
  Showable. Message message=new TestNestedInterface1();
  message.msg();
 }
```

}

OUTPUT



```
F:\JAVAPROGRAMS\INNER CLASS>javac TestNestedInterface1.java

F:\JAVAPROGRAMS\INNER CLASS>java TestNestedInterface1
Hello nested interface
```

2.5 INHERITANCE:

2.5.1 BASICS

- ➢ Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).
- ➢ The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.
- ➢ When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
- ➢ Inheritance represents the IS-A relationship which is also known as a parent-child relationship.

2.5.1.1 Why use inheritance in java

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

2.5.1.2 Terms used in Inheritance

- ➢ **Class:**
  - ✓ A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- ➢ **Sub Class/Child Class**:
  - ✓ Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

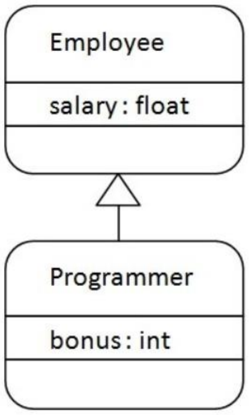
- **Super Class/Parent Class**:
  - ✓ Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability**:
  - ✓ As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

2.5.1.3 The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
   //methods and fields
}
```

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

2.5.1.4 Java Inheritance Example

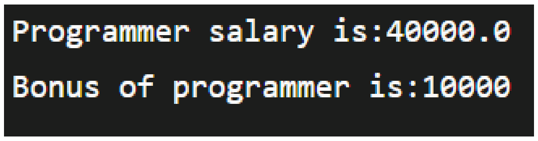


- As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is Programmer IS-A Employee. It means that Programmer is a type of Employee.

Example Program

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

OUTPUT



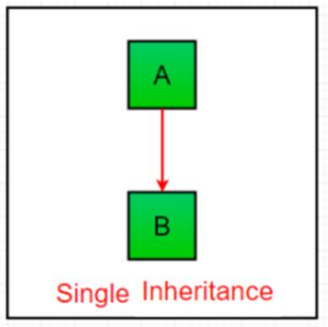
```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

2.5.2 Types of Inheritance

- **Single inheritance**
- **Multilevel inheritance**
- **Hierarchical inheritance**

2.5.2.1 Single inheritance

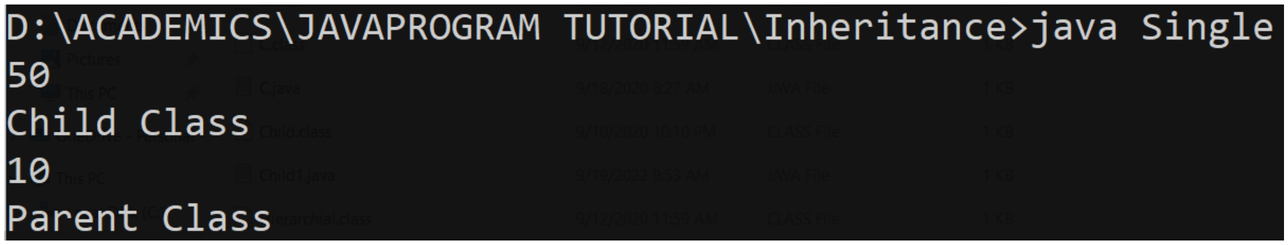
- Single inheritance enables a derived class to inherit properties and behavior from a single parent class.
- In Single Inheritance there will be one parent class and one child class



2.5.2.1.1 Example Program using Single Inheritance

```
class A  //Parent Class
  {
   int a=10;
   void display()  //Function Definition
     {
     System.out.println("Parent Class");
     }
  }
class B extends A   //Child Class
   {
    int b=50;
    void show()
     {
     System.out.println("Child Class");
     }
   }
class Single  //Main Class
 {
  public static void main(String args[])
    {
     B OBJ=new B();
     System.out.println(OBJ.b);
     OBJ.show();
     System.out.println(OBJ.a);
     OBJ.display();
    }
   }
```

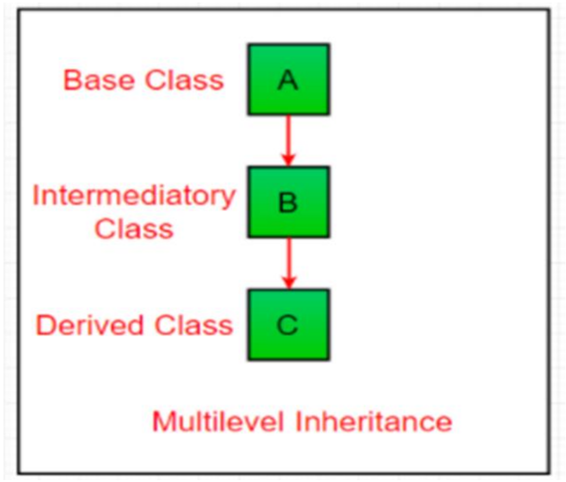
OUTPUT



```
D:\ACADEMICS\JAVAPROGRAM TUTORIAL\Inheritance>java Single
50
Child Class
10
Parent Class
```

2.5.2.2 Multilevel inheritance

- In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class.
- For example, class C extends class B, and class B extends class A.

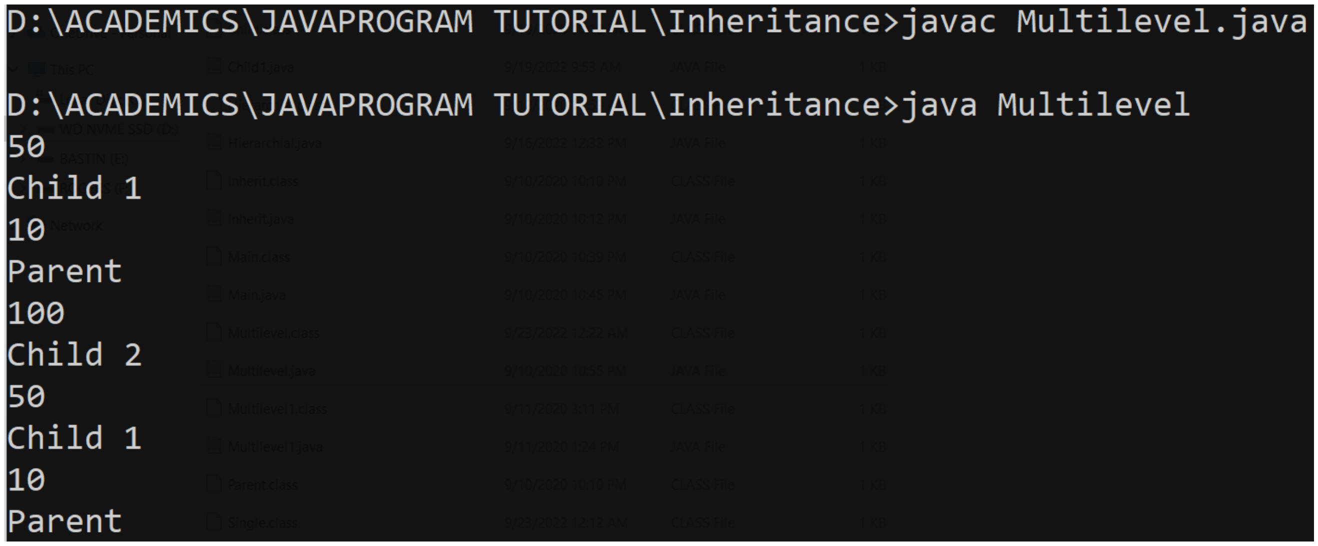


2.5.2.2.1 Example Program using Multilevel Inheritance

```
class A
 {
  int a=10;
  void display()
    {
    System.out.println("Parent");
    }
 }
class B extends A
  {
   int b=50;
   void show()
     {
     System.out.println("Child 1");
     }
  }
class C extends B
```

```
  {
   int c=100;
    void show1()
      {
      System.out.println("Child 2");
      }
   }
class Multilevel
  {
  public static void main(String args[])
     {
      B obj1=new B();
      System.out.println(obj1.b);
      obj1.show();
      System.out.println(obj1.a);
      obj1.display();
      C obj2=new C();
      System.out.println(obj2.c);
      obj2.show1();
      System.out.println(obj2.b);
      obj2.show();
      System.out.println(obj2.a);
      obj2.display();
     }
   }
```

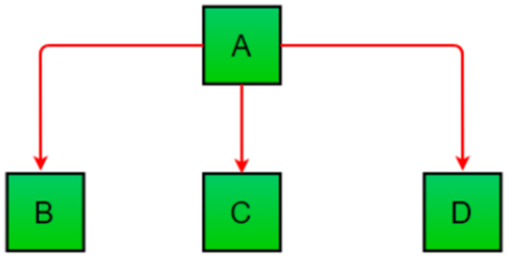
OUTPUT



```
D:\ACADEMICS\JAVAPROGRAM TUTORIAL\Inheritance>javac Multilevel.java

D:\ACADEMICS\JAVAPROGRAM TUTORIAL\Inheritance>java Multilevel
50
Child 1
10
Parent
100
Child 2
50
Child 1
10
Parent
```

2.5.2.3 Hierarchical Inheritance

- "Hierarchical inheritance" occurs when multiple child classes inherit the methods and properties of the same parent class. This simply means we have only one superclass (base class) and multiple sub-classes (Child Class) in hierarchical inheritance in Java.
- In below image, the class A serves as a base class for the derived class B,C and D.



Hierarchical Inheritance

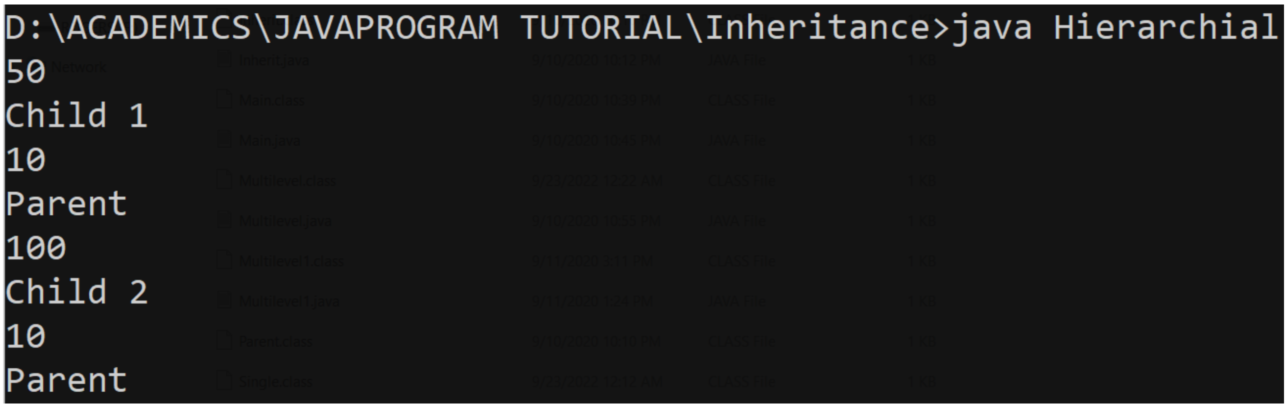
2.5.2.3.1 Example Program using Hierarchical Inheritance

```
class A  //Parent Class
 {
  int a=10;
  void display()
```

```
      {
      System.out.println("Parent");
      }
   }
class B extends A
   {
    int b=50;
    void show()
      {
      System.out.println("Child 1");
      }
   }
class C extends A
  {
   int c=100;
    void show1()
      {
      System.out.println("Child 2");
      }
   }
class Hierarchial  // Main Class
  {
   public static void main(String args[])
      {
       B obj1=new B();
       System.out.println(obj1.b);
       obj1.show();
```

```
        System.out.println(obj1.a);
        obj1.display();
        C obj2=new C();
        System.out.println(obj2.c);
        obj2.show1();
        System.out.println(obj2.a);
        obj2.display();
       }
      }
```

OUTPUT



```
D:\ACADEMICS\JAVAPROGRAM TUTORIAL\Inheritance>java Hierarchial
50
Child 1
10
Parent
100
Child 2
10
Parent
```

2.6 SUPER KEYWORD

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

2.6.1 Usage of Java super Keyword

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

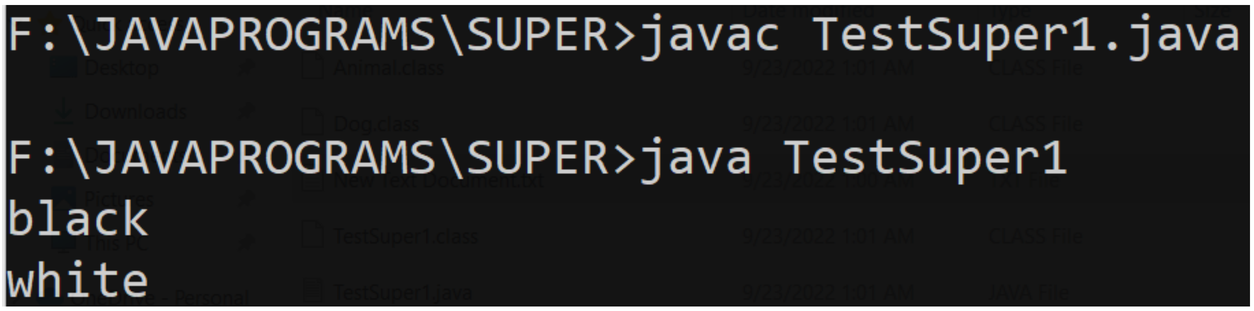
2.6.1.1 super is used to refer immediate parent class instance variable.

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

2.6.1.1.1 Example Program

```
class Animal

{

String color="white";

}

class Dog extends Animal

{

String color="black";

void printColor()

{

System.out.println(color); //prints color of Dog class

System.out.println(super.color); //prints color of Animal class

}

}

class TestSuper1{

public static void main(String args[])

{

Dog d=new Dog();
d.printColor();

}}
```

OUTPUT



```
F:\JAVAPROGRAMS\SUPER>javac TestSuper1.java

F:\JAVAPROGRAMS\SUPER>java TestSuper1
black
white
```

- In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

2.6.1.2 super can be used to invoke immediate parent class method.

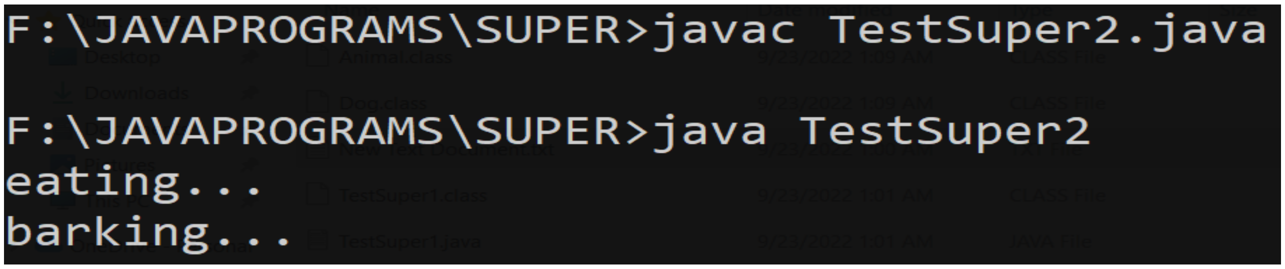
- The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

2.6.1.2.1 Example Program

```
class Animal
{
void eat()
{
System.out.println("eating...");}
}
class Dog extends Animal
{
void eat()
{
```

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

void bark()
{
System.out.println("barking...");
}
void work()
{
super.eat();
bark();
}
}
class TestSuper2
{
public static void main(String args[])
{
Dog d=new Dog();
d.work();
}
}
```

OUTPUT



```
F:\JAVAPROGRAMS\SUPER>javac TestSuper2.java

F:\JAVAPROGRAMS\SUPER>java TestSuper2
eating...
barking...
```

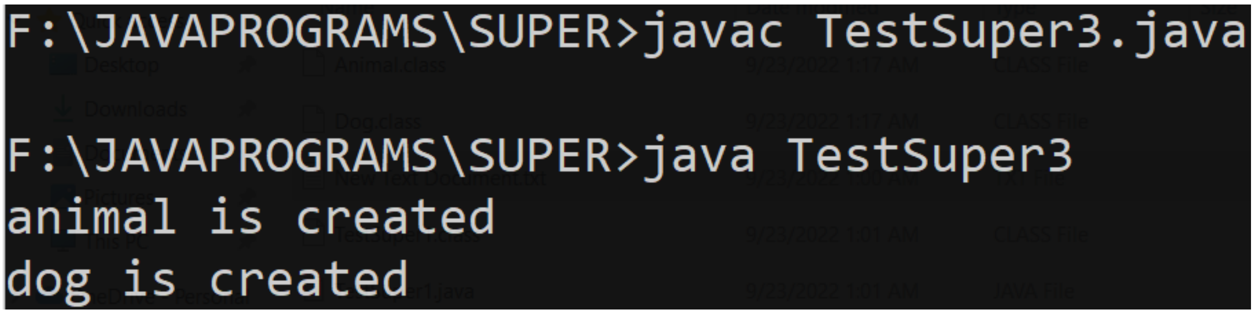
- In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local. To call the parent class method, we need to use super keyword.

2.6.1.3 super is used to invoke parent class constructor.

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

}

OUTPUT

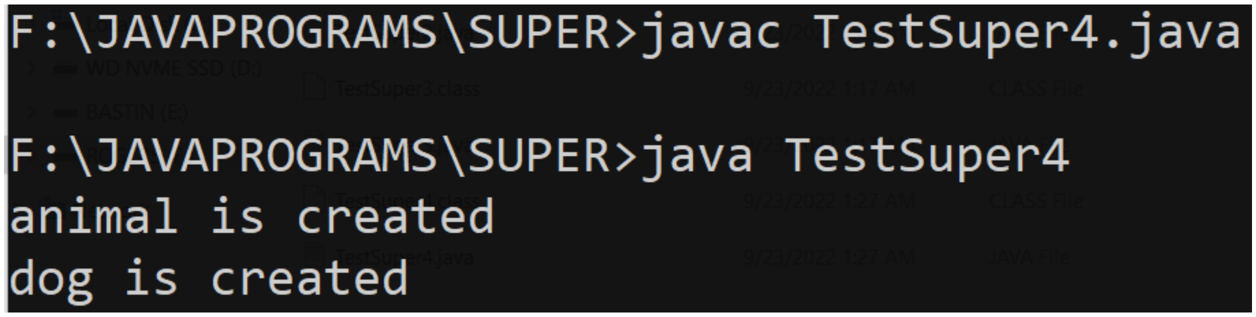


```
F:\JAVAPROGRAMS\SUPER>javac TestSuper3.java

F:\JAVAPROGRAMS\SUPER>java TestSuper3
animal is created
dog is created
```

- In the above example the super keyword used inside the child class constructor Dog will invoke the parent class constructor, i.e., Animal and the statement written inside the parent class constructor will be displayed.

2.6.1.3.1 Example of super keyword where super() is provided by the compiler implicitly.

```
class Animal
{
Animal()
{
System.out.println("animal is created");}
}
class Dog extends Animal
{
Dog()
{
System.out.println("dog is created");
}
}


class TestSuper4

{

public static void main(String args[])

{

Dog d=new Dog();

}

}
```



```
F:\JAVAPROGRAMS\SUPER>javac TestSuper4.java

F:\JAVAPROGRAMS\SUPER>java TestSuper4
animal is created
dog is created
```

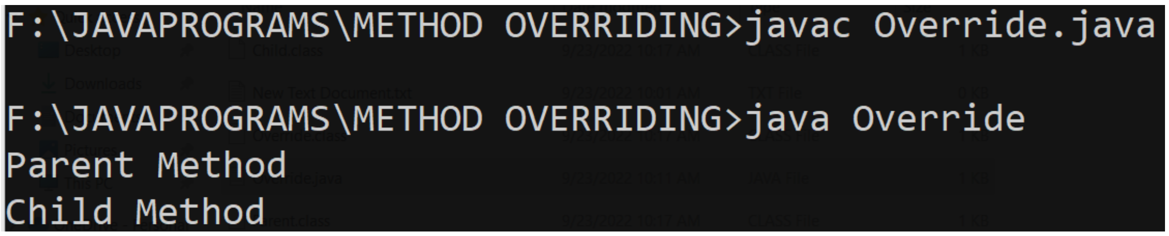
2.7 METHOD OVERRIDING

- In Java, method overriding occurs when a subclass (child class) has the same method as the parent class.
- Here the child class (sub class) will override the method written in super class (parent class)
  - The Method Overriding can be implemented using Inheritance concept
  - Both the method in parent class and the child class should have the same return type
  - Return type, scope and parameters should be same
  - Static Methods cannot be overridden.
  - If the method is declared as a final in the super class, we cannot override that method in child class.
  - It is called as Run time polymorphism
  - Method overriding is an example for Dynamic binding

2.7.1 Example Program for Method Overriding

```
class Parent
 {
  void display()
    {
     System.out.println("Parent Method");
    }
 }
class Child extends Parent
  {
    void display()
    {
     super.display();
     System.out.println("Child Method");
    }
  }
class Override
  {
    public static void main(String args[])
      {
        Child obj=new Child();
        obj.display();
      }
  }
```

OUTPUT



```
F:\JAVAPROGRAMS\METHOD OVERRIDING>javac Override.java

F:\JAVAPROGRAMS\METHOD OVERRIDING>java Override
Parent Method
Child Method
```

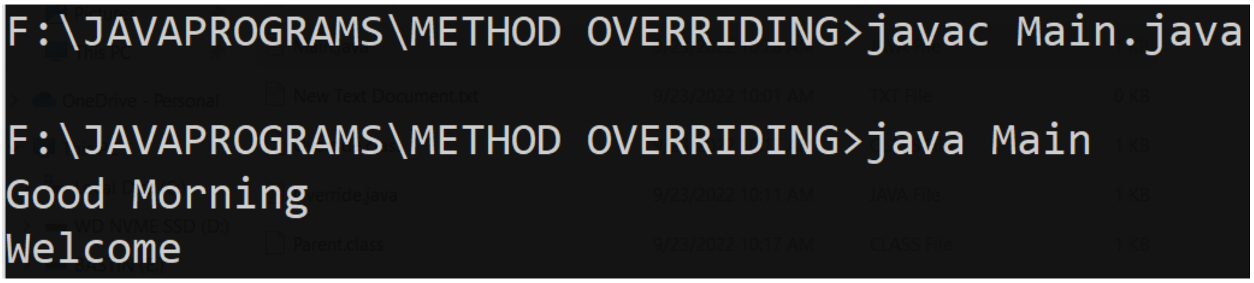
2.7.2 Access Specifiers in Method Overriding

- ➢ The same method declared in the superclass and its subclasses can have different access specifiers. However, there is a restriction.
  - We can only use those access specifiers in subclasses that provide larger access than the access specifier of the superclass. For example,
  - Suppose, a method in the superclass is declared protected. Then, the same method myClass() in the subclass can be either public or protected, but not private.

Example Program

```
class Father
{
  protected void displayInfo()
  {
    System.out.println("Good Morning");
  }
}
class Son extends Father
{
  public void displayInfo()
  {
    super.displayInfo();
    System.out.println("Welcome");
  }
}
class Main
{
  public static void main(String[] args)
   {
    Son d1 = new Son();
    d1.displayInfo();
  }
}
```

OUTPUT



```
F:\JAVAPROGRAMS\METHOD OVERRIDING>javac Main.java

F:\JAVAPROGRAMS\METHOD OVERRIDING>java Main
Good Morning
Welcome
```

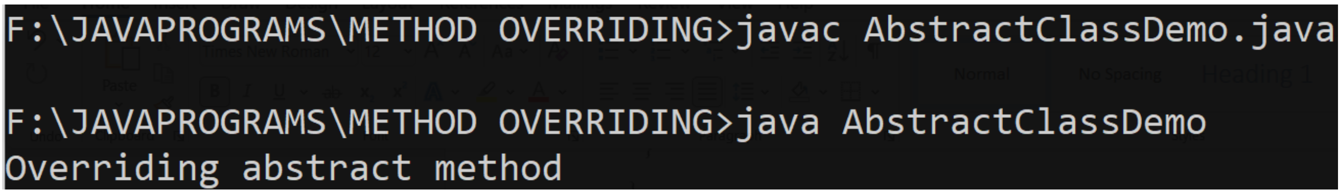
2.7.3 Overriding Abstract Method in Java

- In Java, it is compulsory to override abstract methods of the parent class in its child class because the derived class extends the abstract methods of the base class.
- If we do not override the abstract methods in the subclasses then there will be a compilation error. Therefore, it is necessary for a subclass to override the abstract methods of its base class.

Example Program

```
abstract class Parent
{
  //abstract method
  abstract public void display2();
}
class Child extends Parent
{
  // Must Override this method while extending Parent class
  public void display2()
  {
    System.out.println("Overriding abstract method");
  }
}
public class AbstractClassDemo
{
  public static void main(String[] args)
  {
    Child obj = new Child();
    obj.display2();
  }
}
```

OUTPUT



```
F:\JAVAPROGRAMS\METHOD OVERRIDING>javac AbstractClassDemo.java

F:\JAVAPROGRAMS\METHOD OVERRIDING>java AbstractClassDemo
Overriding abstract method
```

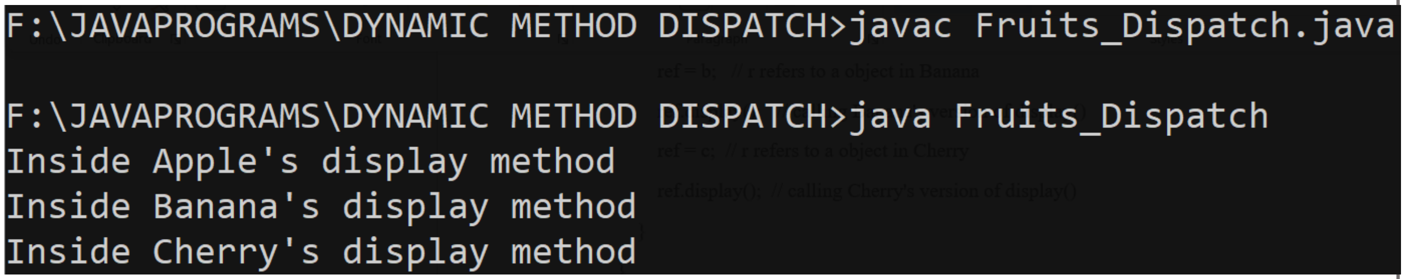
2.8 DYNAMIC METHOD DISPATCH

- In Java, Dynamic method dispatch is a technique in which object refers to superclass but at runtime, the object is constructed for subclass. In other words, it is a technique in which a superclass reference variable refers to a subclass object.

2.8.1 Example Program using Dynamic Method Dispatch

```
class Apple
{
   void display()
   {
      System.out.println("Inside Apple's display method");
   }
}
class Banana extends Apple
{
   void display()   // overriding display()
   {
      System.out.println("Inside Banana's display method");
   }
}
class Cherry extends Apple
{
   void display()   // overriding display()
   {
      System.out.println("Inside Cherry's display method");
   }
}

class Fruits_Dispatch
{
    public static void main(String args[])
    {
        Apple a  = new Apple();   // object of Apple
        Banana b = new Banana();  // object of Banana
        Cherry c = new Cherry();  // object of Cherry
        Apple ref;    // taking a reference of Apple
         ref = a;    // r refers to a object in Apple
         ref.display();   // calling Apple's version of display()
        ref = b;    // r refers to a object in Banana
        ref.display();   // calling Banana's version of display()
        ref = c;   // r refers to a object in Cherry
        ref.display();  // calling Cherry's version of display()
    }
}
```

OUTPUT



```
F:\JAVAPROGRAMS\DYNAMIC METHOD DISPATCH>javac Fruits_Dispatch.java

F:\JAVAPROGRAMS\DYNAMIC METHOD DISPATCH>java Fruits_Dispatch
Inside Apple's display method
Inside Banana's display method
Inside Cherry's display method
```

2.9 ABSTRACT CLASSES

- Abstraction is a process of hiding the implementation details and showing only functionality to the user.
- Another way, it shows only important things to the user and hides the internal details, for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.
- If any class consist of atleast one Abstract method then the class is called as Abstract class. It is declared with abstract keyword. It can have both abstract and non-abstract methods (method with body). Object cannot be created or instantiated for an Abstract Class.
- Example abstract class
  abstract class A{}

2.9.1 Abstract method

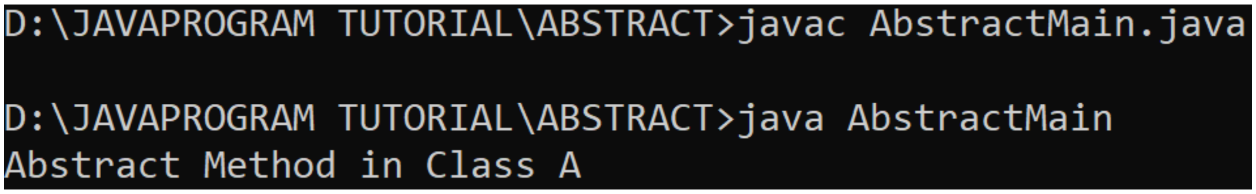
- If any class consists of only the declaration of the method and hiding the implementation part then the method is called as Abstract Method. Implementation of Abstract Method will be written in Derived Class.
- Example abstract method
  - abstract void printStatus(); //no body and abstract

2.9.2 Example of abstract class that has abstract method

```
abstract class A
 {
 abstract void display();
 }
class B extends A
 {
   void display()
    {
     System.out.println("Abstract Method in Class A");
    }
 }
class AbstractMain
 {
 public static void main(String args[])
```

```
    {
     B obj=new B();
     obj.display();
    }
  }
```

OUTPUT



```
D:\JAVAPROGRAM TUTORIAL\ABSTRACT>javac AbstractMain.java

D:\JAVAPROGRAM TUTORIAL\ABSTRACT>java AbstractMain
Abstract Method in Class A
```

2.10 USING FINAL KEYWORD WITH INHERITANCE IN JAVA

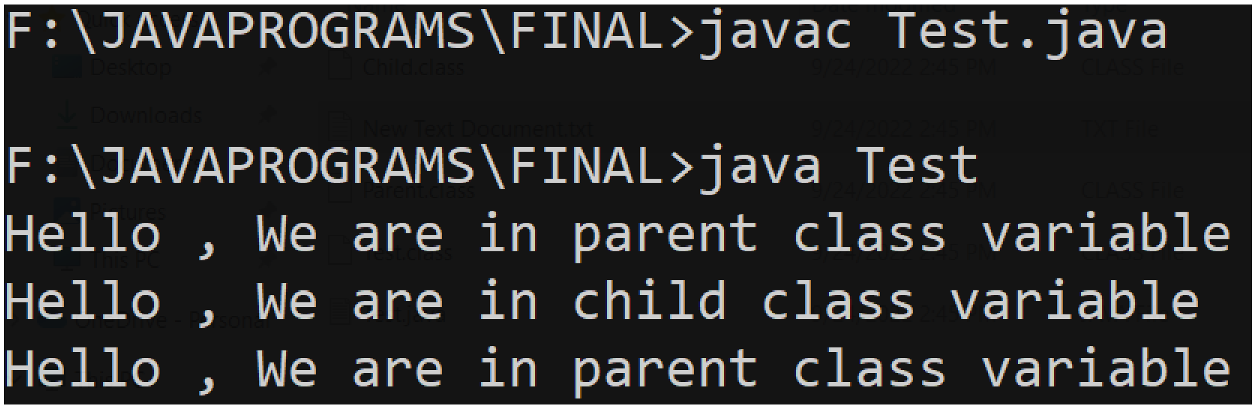
- final is a keyword in java used for restricting some functionalities. We can declare variables, methods, and classes with the final keyword.
- We can use final keywords for variables, methods, and class.
- If we use the final keyword for the inheritance that is if we declare any method with the final keyword in the base class so the implementation of the final method will be the same as in derived class.
- We can declare the final method in any subclass for which we want that if any other class extends this subclass.

2.10.1 Case 1: Declare final variable with inheritance

```
class Parent

  {

   /* Creation of final variable pa of string type i.e

   the value of this variable is fixed throughout all

   the derived classes or not overidden*/

   final String pa = "Hello , We are in parent class variable";

   }
```

```
 class Child extends Parent
{
 /* Creation of variable ch of string type i.e
 the value of this variable is not fixed throughout all
 the derived classes or overidden*/
 String ch = "Hello , We are in child class variable";
 }
 class Test
 {
 public static void main(String[] args)
  {
     // Creation of Parent class object
     Parent p = new Parent();
     // Calling a variable pa by parent object
     System.out.println(p.pa);
     // Creation of Child class object
     Child c = new Child();
     // Calling a variable ch by Child object
     System.out.println(c.ch);
     // Calling a variable pa by Child object
     System.out.println(c.pa);
  }
}
```

OUTPUT

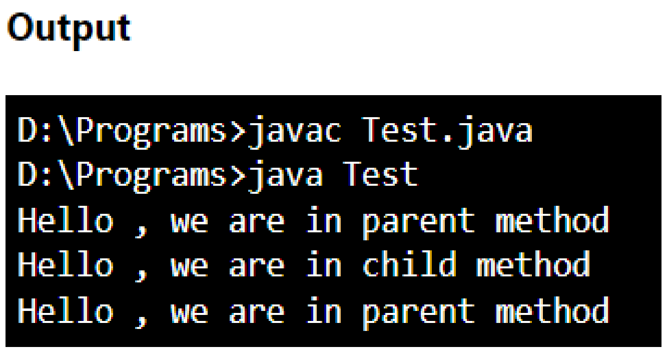


```
F:\JAVAPROGRAMS\FINAL>javac Test.java

F:\JAVAPROGRAMS\FINAL>java Test
Hello , We are in parent class variable
Hello , We are in child class variable
Hello , We are in parent class variable
```

2.10.2 Case 2: Declare final methods with inheritance

```
// Declaring Parent class
class Parent {
   /* Creation of final method parent of void type i.e
   the implementation of this method is fixed throughout
   all the derived classes or not overidden*/
   final void parent() {
      System.out.println("Hello , we are in parent method");
   }
}
// Declaring Child class by extending Parent class
class Child extends Parent {
   /* Creation of final method child of void type i.e
   the implementation of this method is not fixed throughout
   all the derived classes or not overidden*/
   void child()
   {
      System.out.println("Hello , we are in child method");
   }
}
class Test {
   public static void main(String[] args) {
      // Creation of Parent class object
      Parent p = new Parent();
      // Calling a method parent() by parent object
      p.parent();
      // Creation of Child class object
      Child c = new Child();
      // Calling a method child() by Child class object
      c.child();
      // Calling a method parent() by child object
      c.parent();
   }
}
```

OUTPUT


Output

```
D:\Programs>javac Test.java
D:\Programs>java Test
Hello , we are in parent method
Hello , we are in child method
Hello , we are in parent method
```

2.10.3 Java final class

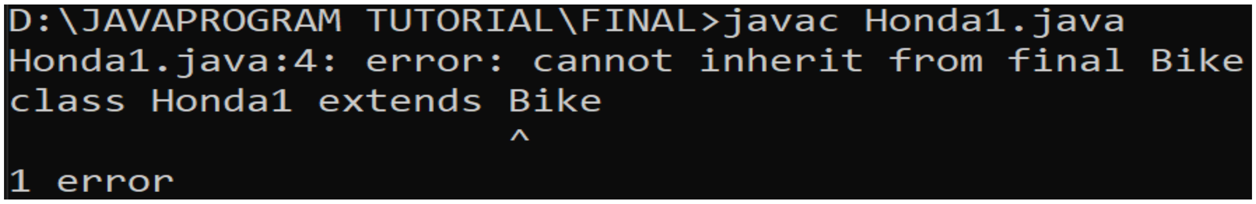
➢ If we make any class as final, we cannot extend it.

Example

```
final class Bike
{
}
class Honda1 extends Bike
```

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

OUTPUT



```
D:\JAVAPROGRAM TUTORIAL\FINAL>javac Honda1.java
Honda1.java:4: error: cannot inherit from final Bike
class Honda1 extends Bike
                     ^
1 error
```

2.11 INTERFACES.

2.11.1 Interfaces-Declaration and Implementation

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

2.11.1.1 Interface Declaration

- Interface is declared by using interface keyword. It provides total abstraction; means all the methods in interface are declared with empty body and are public and all fields are public, static and final by default.
- A class that implement interface must implement all the methods declared in the interface.

Syntax:

```
interface <interface_name>

{

   // declare constant fields

   // declare methods that abstract

   // by default.

}
```

2.8.2 Interface Implementation

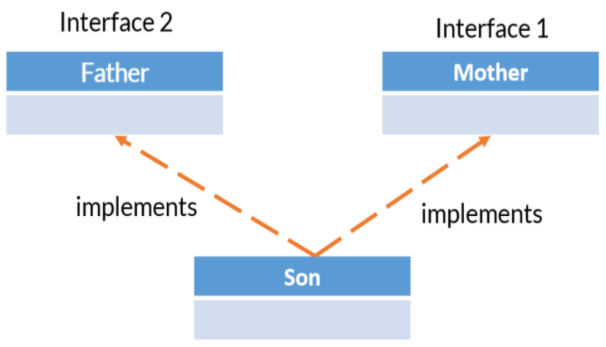
```
interface printable
{
    void print();
}
class A6 implements printable
{
    public void print()
    {
            System.out.println("Hello");
    }
    public static void main(String args[])
    {
            A6 obj = new A6();
            obj.print();
    }
}
```

Output

```
Hello
```

2.11.2 Achieving Multiple Inheritance using Interface

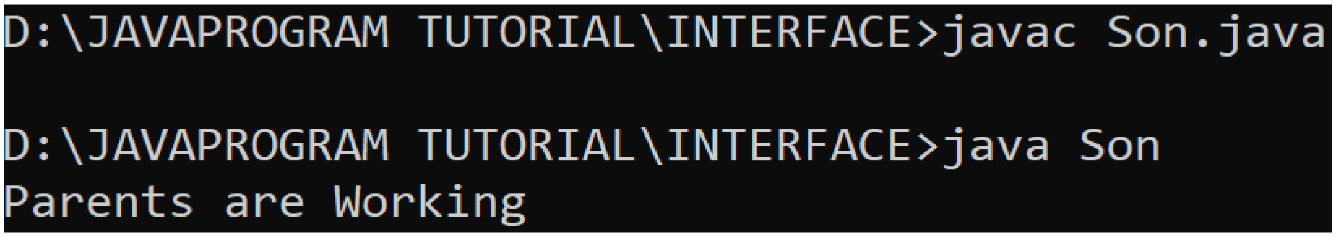
- Multiple Inheritance is a feature of object-oriented concept, where a class can inherit properties of **more than one parent class**.



Example Program

```
interface Father
{
     public abstract void work(); //Method Declaration
}
interface Mother
{
     public abstract void work(); //Method Declaration
}
public class Son implements Father,Mother
   {
    public void work()
      {
      System.out.println("Parents are Working");
      }
   public static void main(String args[])
   {
       Son S = new Son();
       S.work();
   }
}
```

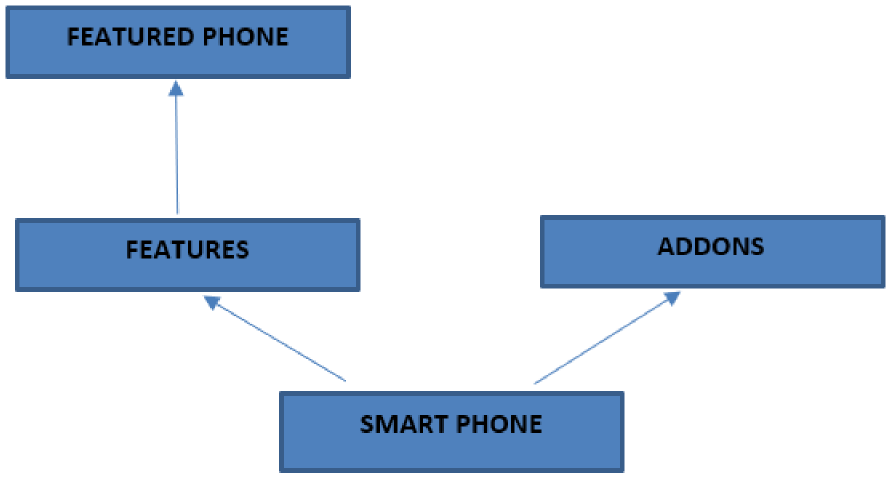
OUTPUT



```
D:\JAVAPROGRAM TUTORIAL\INTERFACE>javac Son.java

D:\JAVAPROGRAM TUTORIAL\INTERFACE>java Son
Parents are Working
```

2.11.3 HYBRID INHERITANCE

- Hybrid inheritance is a combination of **Single** and **Multiple inheritance.**
- By using **interfaces,** we can achieve **hybrid inheritance** in Java



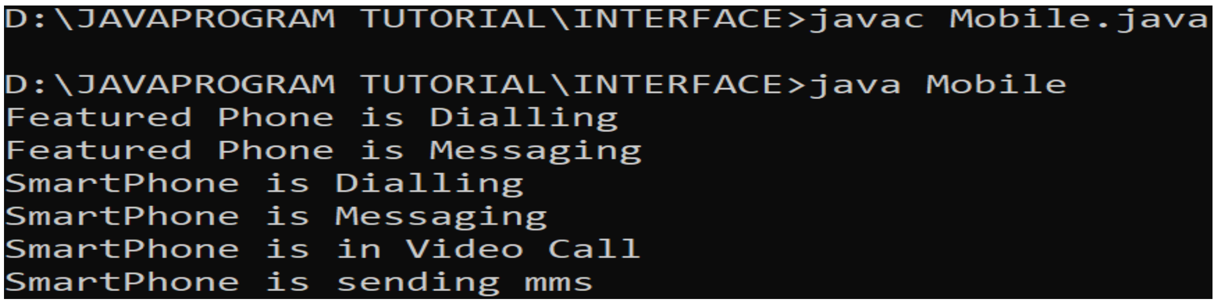
Example Program

```
interface Features
 {
  abstract void dialling();
  abstract void messaging();
 }
interface Addons
 {
  abstract void vcalling();
  abstract void mms();
 }
class Featuredphone implements Features
```

```
 {
  public void dialling()
    {
     System.out.println("Featured Phone is Dialling");
    }

  public void messaging()
    {
     System.out.println("Featured Phone is Messaging");
    }
 }
class SmartPhone implements Features,Addons
  {
    public void dialling()
    {
     System.out.println("SmartPhone is Dialling");
    }
  public void messaging()
    {
     System.out.println("SmartPhone is Messaging");
    }
  public void vcalling()
    {
     System.out.println("SmartPhone is in Video Call");
    }
  public void mms()
    {
     System.out.println("SmartPhone is sending mms");
    }
  }
class Mobile
```

```
{
 public static void main(String args[])
   {
    Featuredphone fp=new Featuredphone();
    SmartPhone sp=new SmartPhone();
    fp.dialling();
    fp.messaging();
    sp.dialling();
    sp.messaging();
    sp.vcalling();
    sp.mms();
   }
}
```

OUTPUT



```
D:\JAVAPROGRAM TUTORIAL\INTERFACE>javac Mobile.java

D:\JAVAPROGRAM TUTORIAL\INTERFACE>java Mobile
Featured Phone is Dialling
Featured Phone is Messaging
SmartPhone is Dialling
SmartPhone is Messaging
SmartPhone is in Video Call
SmartPhone is sending mms
```

2.12 Packages

A Package can be defined as a grouping of related types (classes, interfaces, enumerations and annotations) providing access protection and namespace management.

Some of the existing packages in Java are −

- java.lang − bundles the fundamental classes
- java.io − classes for input , output functions are bundled in this package

2.12.1 Creating a Package

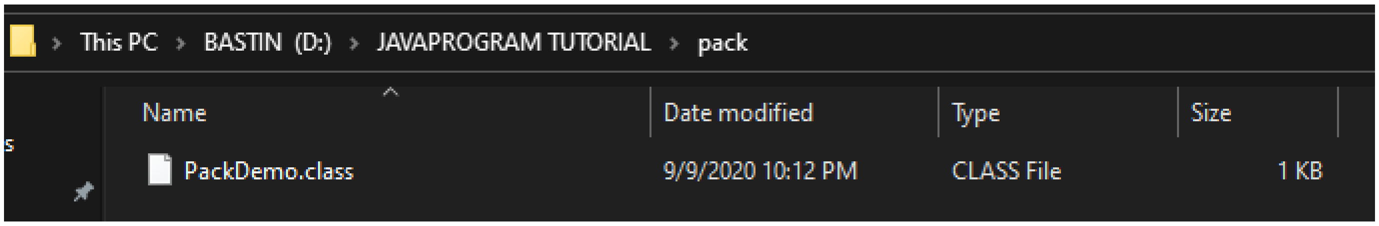
- While creating a package, First choose a name for the package and include a **package** statement along with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.
- The package statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.
- If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.
- To compile the Java programs with package statements, you have to use -d option as shown below.

javac -d Destination_folder file_name.java

Then a folder with the given package name is created in the specified destination, and the compiled class files will be placed in that folder.

Example

```
package pack;
public class PackDemo
{
  public void show()
    {
      System.out.println("Welcome to JAVA");
    }
}
```



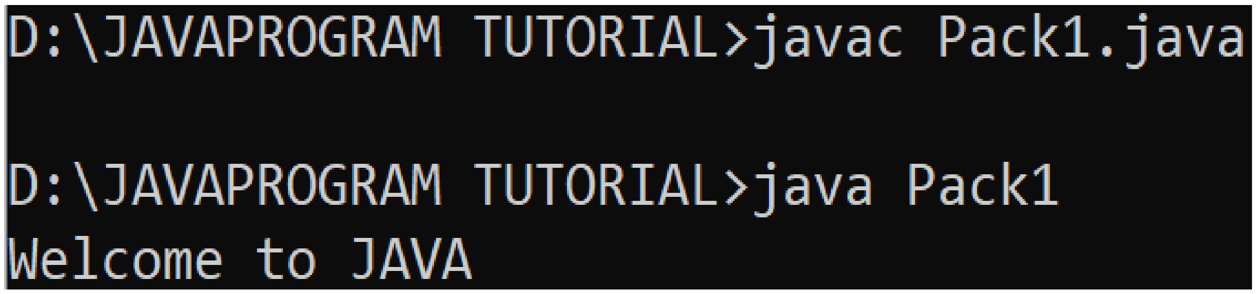
This PC > BASTIN (D:) > JAVAPROGRAM TUTORIAL > pack

Name	Date modified	Type	Size
PackDemo.class	9/9/2020 10:12 PM	CLASS File	1 KB

```
import pack.PackDemo;
class Pack1
  {
```

```
  public static void main(String arg[])
    {
     PackDemo obj=new PackDemo();
     obj.show();
    }
 }
```

OUTPUT



```
D:\JAVAPROGRAM TUTORIAL>javac Pack1.java

D:\JAVAPROGRAM TUTORIAL>java Pack1
Welcome to JAVA
```

2.12.2 IMPORTING PACKAGES

- In java, the ***import*** keyword used to import built-in and user-defined packages. When a package has imported, we can refer to all the classes of that package using their name directly.
- The import statement must be after the package statement, and before any other statement.
- Using an import statement, we may import a specific class or all the classes from a package.

2.12.2.1 Importing specific class

Syntax

import packageName.ClassName;

2.12.2.1.1Example

```
package myPackage;

import java.util.Scanner;
```

```
public class ImportingExample {
    public static void main(String[] args) {
        Scanner read = new Scanner(System.in);
        int i = read.nextInt();
        System.out.println("You have entered a number " + i);
    }
}
```

- In the above code, the class ImportingExample belongs to myPackage package, and it also importing a class called Scanner from java.util package.

2.12.2.2 Importing all the classes

Syntax

```
import packageName.*;
```

Example Program

```
package myPackage;
import java.util.*;
public class ImportingExample {
    public static void main(String[] args) {
        Scanner read = new Scanner(System.in);
        int i = read.nextInt();
        System.out.println("You have entered a number " + i);
        Random rand = new Random();
        int num = rand.nextInt(100);
        System.out.println("Randomly generated number " + num);
        }
}
```

- In the above code, the class ImportingExample belongs to myPackage package, and it also importing all the classes like Scanner, Random, Stack, Vector, ArrayList, HashSet, etc. from the java.util package.

UNIT III - EXCEPTION HANDLING AND MULTITHREADING

Exception Handling basics – Multiple catch Clauses – Nested try Statements – Java's Built-in Exceptions – User defined Exception. Multithreaded Programming: Java Thread Model–Creating a Thread and Multiple Threads – Priorities – Synchronization – Inter Thread Communication- Suspending – Resuming and Stopping Threads –Multithreading. Wrappers – Auto boxing.

3.1. EXCEPTION HANDLING BASICS

The exception handling in java is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

3.1.1 Types of Exception

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception.

The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between checked and unchecked exceptions

1) Checked Exception

The checked exceptions are those exceptions which is checked by the compiler at runtime e.g.IOException, SQLException etc.

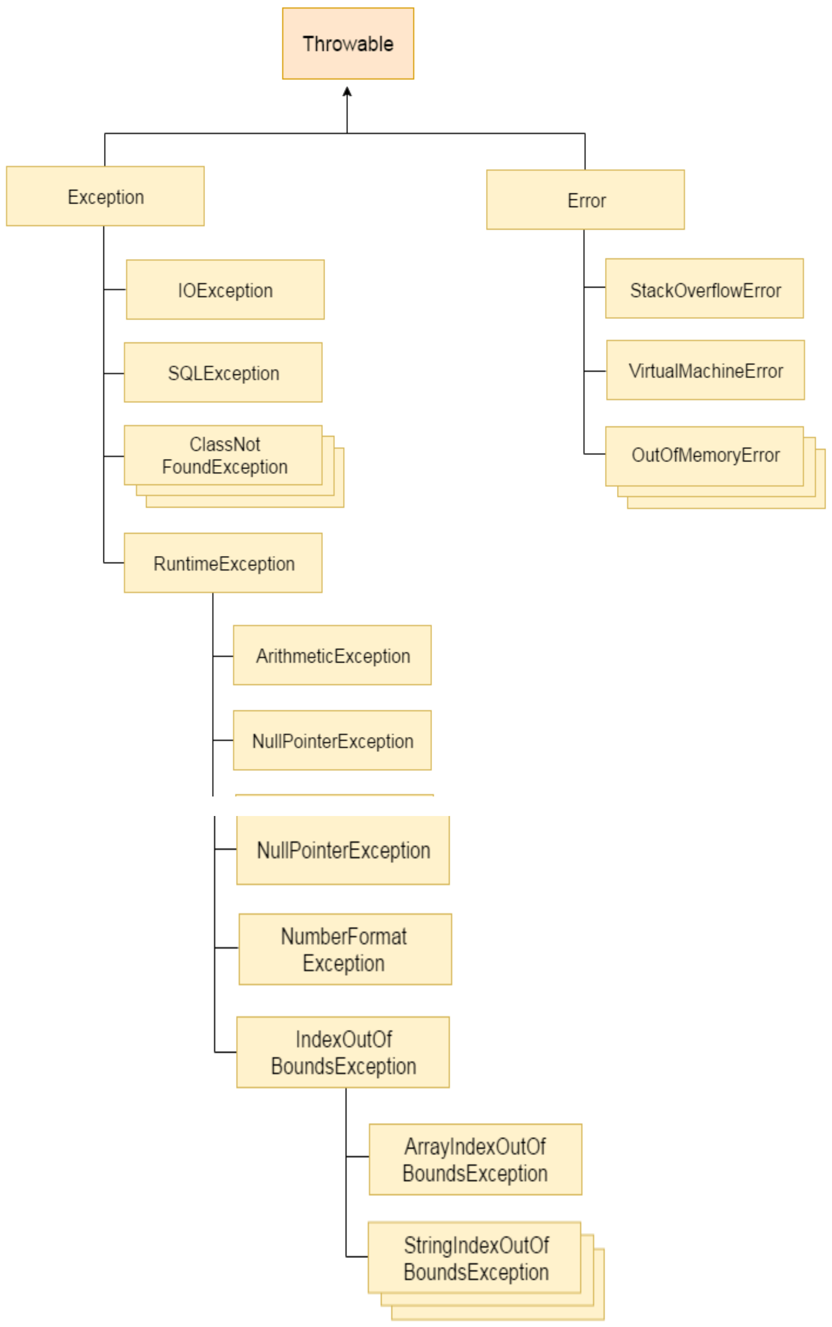
2) Unchecked Exception

The Unchecked exceptions are not checked by the compiler at run time e.g. Arithmetic Exception, NullPointerException, ArrayIndexOutOfBoundsException etc.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

3.1.2. HIERARCHY OF JAVA EXCEPTION CLASSES



3.1.3 THROWING AND CATCHING EXCEPTION

There are 5 keywords used in java exception handling.

1. try
2. catch
3. finally
4. throw
5. throws

Java Try Block

Java try block is used to enclose the code that might throw an exception. It must be used within the method.Java try block must be followed by either catch or finally block.

Syntax

```
try
{
//code that may throw exception
}
catch(Exception_class_Name ref)
{}
```

Catching Exceptions

A method catches an exception using a combination of the try and catchkeywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following –

Syntax

```
try
{
  // Protected code
}
catch (ExceptionName e1)
{
  // Catch block
}
```

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every trya block should be immediately followed either by a catch block or finally block.

Example

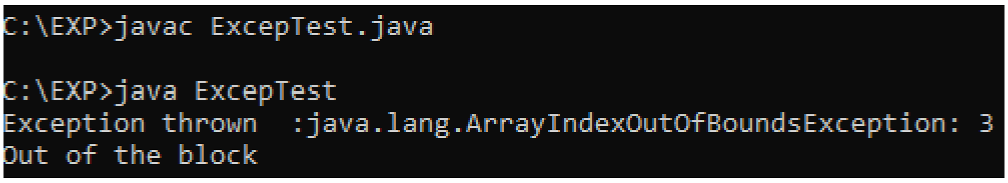
The following is an array declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;
public class ExcepTest
```

```
{
        public static void main(String args[])
        {
                try
                {
                        int a[] = new int[2];
                        System.out.println("Access element three :" + a[3]);
                } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Exception thrown  :" + e);
   }
   System.out.println("Out of the block");
  }
}
```

This will produce the following result −

Output



```
C:\EXP>javac ExcepTest.java

C:\EXP>java ExcepTest
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

3.2 MULTIPLE CATCH BLOCKS

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following −

Syntax

```
try {
  // Protected code
   }
 catch (ExceptionType1 e1)
{
  // Catch block
} catch (ExceptionType2 e2)
 {
  // Catch block
} catch (ExceptionType3 e3)
 {
  // Catch block
}
```

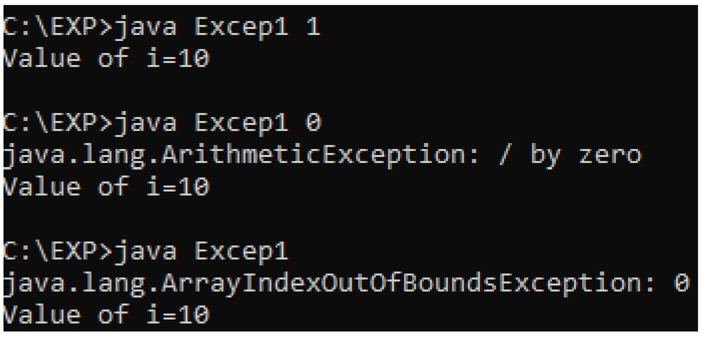
The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Example

Here is code segment showing how to use multiple try/catch statements.

```
class Excep1
  {
 public static void main(String args[])
     {
     int i=10;
try
{
   i=i/Integer.parseInt(args[0]);
}
 catch(ArithmeticException e)
{
  System.out.println(e);
}
catch(Exception e1)
 {
  System.out.println(e1);
}
  System.out.println("Value of i=" +i);
}
}
```

OUTPUT



```
C:\EXP>java Excep1 1
Value of i=10

C:\EXP>java Excep1 0
java.lang.ArithmeticException: / by zero
Value of i=10

C:\EXP>java Excep1
java.lang.ArrayIndexOutOfBoundsException: 0
Value of i=10
```

3.3 Nested Try Statements

- The try block within a try block is called nested try block. Every inner try block must have a corresponding catch blocks.If any inner try block does not does not have a catch block then the outer try block catch statement will be executed.

Syntax

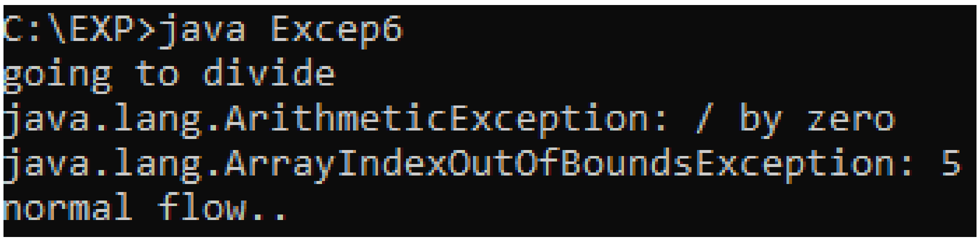
```
try
{
   statement 1;
   statement 2;
   try
   {
      statement 1;
      statement 2;
   }
   catch(Exception e)
   {
   }
}
catch(Exception e)
{
}
```

Program

```
class Excep6
{
 public static void main(String args[])
  {
    try
     {
     System.out.println("going to divide");
     int b =39/0;
     }
  catch(ArithmeticException e)
     {
     System.out.println(e);
     }


    try
```

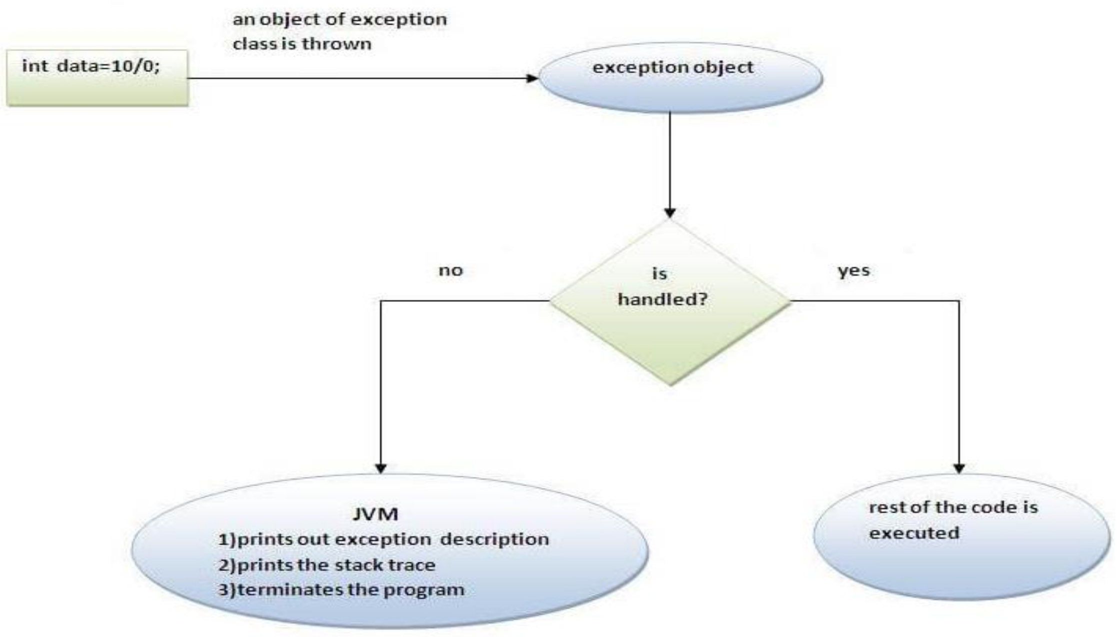
```
   {
  int a[]=new int[5];
  a[5]=4;
   }
 catch(ArrayIndexOutOfBoundsException e)
   {
   System.out.println(e);
   }
 catch(Exception e)
   {
   System.out.println("handeled");
   }
 System.out.println("normal flow..");
 }
}
```

Output



```
C:\EXP>java Excep6
going to divide
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: 5
normal flow..
```

Internal working of java try-catch block


int data=10/0;
an object of exception class is thrown
exception object
is handled?
no
yes
JVM
1)prints out exception description
2)prints the stack trace
3)terminates the program
rest of the code is executed

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

The Finally Block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception. Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax −

Syntax

```
try {
  // Protected code
} catch (ExceptionType1 e1) {
  // Catch block
} catch (ExceptionType2 e2) {
  // Catch block
} catch (ExceptionType3 e3) {
  // Catch block
}finally {
  // The finally block always executes.
}
```

Case 1

Let's see the java finally example where exception doesn't occur.

```
class TestFinallyBlock
{
                public static void main(String args[])
               {
                       try
                       {
                               int data=25/5;
                               System.out.println(data);
                       }
                       catch(NullPointerException e)
```

```
                        {
                                System.out.println(e);
                        }
                        finally
                        {
                                System.out.println("finally block is always executed");
                        }
                        System.out.println("rest of the code...");
                }
}
```

Test it Now

Output:

```
5
finally block is always executed
```

Case 2

Let's see the java finally example where exception occurs and not handled.

```
class TestFinallyBlock1
{
        public static void main(String args[])
        {
                try
                {
                        int data=25/0;
                        System.out.println(data);
                }
                catch(NullPointerException e)
                {
                        System.out.println(e);
                }
                finally
                {
                        System.out.println("finally block is always executed");
                }
                System.out.println("rest of the code...");
        }
}
```

Output:finally block is always executed

```
     Exception in thread main java.lang.ArithmeticException:/ by zero
```

Case 3

Let's see the java finally example where exception occurs and handled.

```
public class TestFinallyBlock2
{
        public static void main(String args[])
        {
                try
                {
                        int data=25/0;
                        System.out.println(data);
```

```
                    }
                    catch(ArithmeticException e)
                    {
                              System.out.println(e);
                    }
                    finally
                    {
                              System.out.println("finally block is always executed");
                    }
                    System.out.println("rest of the code...");
          }
}
```

```
Output: Exception in thread main java.lang.ArithmeticException:/ by zero
      finally block is always executed
```

3.4.THROWING AND CATCHING EXCEPTIONS

The Throw Keyword

The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exception.The throw keyword is mainly used to throw custom exceptions.

Syntax

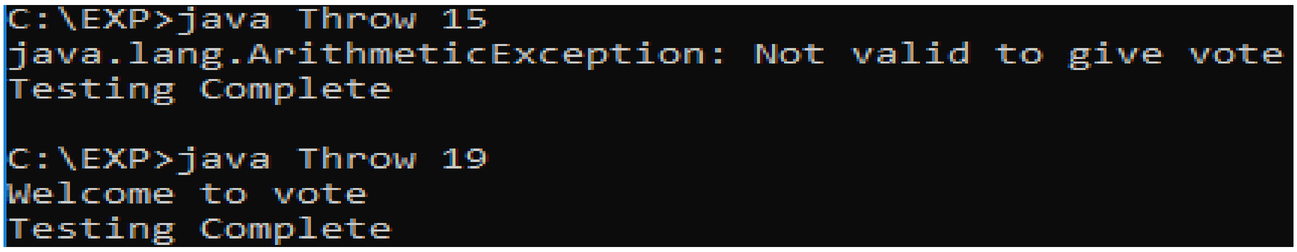
```
throw new ArithmeticException("—");
```

Program

```
class Throw
  {
  static void validage(int age)
    {
    if(age<18)
       {
        throw new ArithmeticException("Not valid to give vote");
       }
    else
      {
       System.out.println("Welcome to vote");
      }
   }
public static void main(String args[])
  {
  try
   {
   validage(Integer.parseInt(args[0]));
   }
  catch(ArithmeticException e)
     {
     System.out.println(e);
     }
```

```
 System.out.println("Testing Complete");
}
}
```

Output



```
C:\EXP>java Throw 15
java.lang.ArithmeticException: Not valid to give vote
Testing Complete

C:\EXP>java Throw 19
Welcome to vote
Testing Complete
```

The Throws Keyword

The Java throws keyword is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

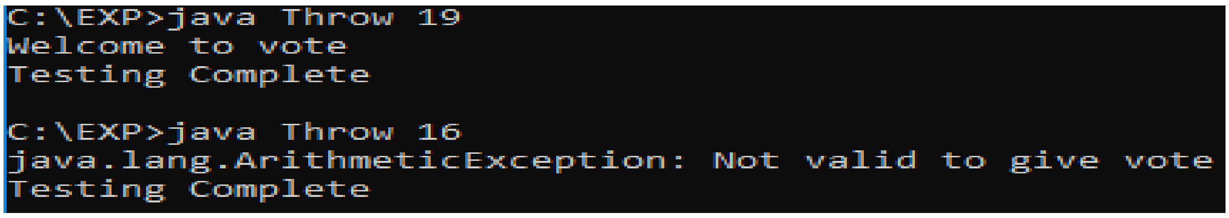
Syntax

```
return_type method_name() throws exception_class_name{
//method code
}
```

Program

```
class myexception extends Exception
 {
 myexception(String s)
   {
   super(s);
   }
}
class Throws
 {
 static void validage(int age) throws myexception
  {
  if(age<18)
    {
     throw new myexception("Not valid to give vote");
    }
  else
   {
    System.out.println("Welcome to vote");
   }
  }
public static void main(String args[])
 {
 try
  {
  validage(Integer.parseInt(args[0]));
  }
 catch(myexception my)
   {
   System.out.println(my);
   }
 System.out.println("Testing Complete");
}
}
```

Output



```
C:\EXP>java Throw 19
Welcome to vote
Testing Complete

C:\EXP>java Throw 16
java.lang.ArithmeticException: Not valid to give vote
Testing Complete
```

3.4 JAVA'S BUILT-IN EXCEPTIONS

SI.NO.	Java UnChecked Exceptions Defined in java.lang.	Java Checked Exceptions Defined in java.lang.
1.	ArithmeticException Arithmetic error, such as divide-by-zero.	ClassNotFoundException Class not found.
2.	ArrayIndexOutOfBoundsException Array index is out-of-bounds.	CloneNotSupportedException

		Attempt to clone an object that does not implement the Cloneable interface.
3.	ArrayStoreException Assignment to an array element of an incompatible type.	IllegalAccessException Access to a class is denied.
4.	ClassCastException Invalid cast.	InstantiationException Attempt to create an object of an abstract class or interface.
5.	IllegalArgumentException Illegal argument used to invoke a method.	InterruptedException One thread has been interrupted by another thread.
6.	IllegalMonitorStateException Illegal monitor operation, such as waiting on an unlocked thread.	NoSuchFieldException A requested field does not exist.
7.	IllegalStateException Environment or application is in incorrect state.	NoSuchMethodException A requested method does not exist.
8.	IllegalThreadStateException Requested operation not compatible with the current thread state.	
9.	IndexOutOfBoundsException Some type of index is out-of-bounds.	
10.	NegativeArraySizeException Array created with a negative size.	
11.	NullPointerException Invalid use of a null reference.	
12.	NumberFormatException Invalid conversion of a string to a numeric format.	
13.	SecurityException Attempt to violate security.	
14.	StringIndexOutOfBounds	

	Attempt to index outside the bounds of a string.	
15.	UnsupportedOperationException An unsupported operation was countered.	

Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

3.4.1 Examples of Built-in Exception:

1. **Arithmetic exception:** It is thrown when an exceptional condition has occurred in an arithmetic operation.

```
// Java program to demonstrate
// ArithmeticException
class ArithmeticException_Demo
{
public static void main(String args[])
   {
      try {
         int a = 30, b = 0;
         int c = a / b; // cannot divide by zero
         System.out.println("Result = " + c);
      }
      catch (ArithmeticException e) {
         System.out.println("Can't divide a number by 0");
      }
   }
}
```

Output:

```
Can't divide a number by 0
```

2. **ArrayIndexOutOfBounds Exception** : It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

```
// Java program to demonstrate
// ArrayIndexOutOfBoundException
class ArrayIndexOutOfBound_Demo
{
public static void main(String args[])
   {
       try
          {
          int a[] = new int[5];
          a[6] = 9; // accessing 7th element in an array of size 5
       }
       catch (ArrayIndexOutOfBoundsException e) {
          System.out.println("Array Index is Out Of Bounds");
       }
   }
}
```

Output:

```
Array Index is Out Of Bounds
```

3. **ClassNotFoundException** : This Exception is raised when we try to access a class whose definition is not found.

```
// Java program to illustrate the
// concept of ClassNotFoundException
class Bishal
{
}
 class Geeks
{
}
 class MyClass
{
public static void main(String[] args)
   {
       Object o = class.forName(args[0]).newInstance();
```

```
            System.out.println("Class created for" + o.getClass().getName());
        }
    }
```

Output:

```
ClassNotFoundException
```

4. **FileNotFoundException** : This Exception is raised when a file is not accessible or does not open.

```
// Java program to demonstrate
// FileNotFoundException
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
class File_notFound_Demo
{
public static void main(String args[])
   {
      try {

         // Following file does not exist
         File file = new File("E:// file.txt");

         FileReader fr = new FileReader(file);
      }
      catch (FileNotFoundException e) {
         System.out.println("File does not exist");
      }
   }
}
```

Output:

```
File does not exist
```

5. **IOException** : It is thrown when an input-output operation failed or interrupted

```
// Java program to illustrate IOException
import java.io.*;
class Geeks
 {
public static void main(String args[])
   {
      FileInputStream f = null;
      f = new FileInputStream("abc.txt");
      int i;
      while ((i = f.read()) != -1) {
```

```
                System.out.print((char)i);
             }
             f.close();
          }
       }
```

Output:

error: unreported exception IOException; must be caught or declared to be thrown

6. **InterruptedException** : It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.

```
// Java Program to illustrate
// InterruptedException
class Geeks {
public static void main(String args[])
   {
      Thread t = new Thread();
      t.sleep(10000);
   }
}
```

Output:

error: unreported exception InterruptedException; must be caught or declared to be thrown

7. **NoSuchMethodException :** t is thrown when accessing a method which is not found.

```
// Java Program to illustrate
// NoSuchMethodException
class Geeks {
public Geeks()
  {
      Class i;
      try {
         i = Class.forName("java.lang.String");
         try {
            Class[] p = new Class[5];
         }
```

```
                catch (SecurityException e) {
                    e.printStackTrace();
                }
                catch (NoSuchMethodException e) {
                    e.printStackTrace();
                }
            }
            catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
        }
     public static void main(String[] args)
        {
            new Geeks();
        }
    }
```

Output:

```
error: exception NoSuchMethodException is never thrown
in body of corresponding try statement
```

8. **NullPointerException** : This exception is raised when referring to the members of a null object. Null represents nothing

```
// Java program to demonstrate NullPointerException
class NullPointer_Demo {
public static void main(String args[])
{
        try {
            String a = null; // null value
            System.out.println(a.charAt(0));
        }
        catch (NullPointerException e) {
            System.out.println("NullPointerException..");
        }
    }
}
```

Output:

```
NullPointerException..
```

9. **NumberFormatException** : This exception is raised when a method could not convert a string into a numeric format.

```
// Java program to demonstrate
// NumberFormatException
class NumberFormat_Demo
{
  public static void main(String args[])
   {
      try {
         // "akki" is not a number
         int num = Integer.parseInt("akki");
          System.out.println(num);
      }
      catch (NumberFormatException e) {
         System.out.println("Number format exception");
      }
   }
}
```

Output:

```
Number format exception
```

10. **StringIndexOutOfBoundsException** : It is thrown by String class methods to indicate that an index is either negative than the size of the string.

```
// Java program to demonstrate
// StringIndexOutOfBoundsException
class StringIndexOutOfBound_Demo {
public static void main(String args[])
 {
     try {
        String a = "This is like chipping "; // length is 22
        char c = a.charAt(24); // accessing 25th element
```

```
            System.out.println(c);
          }
          catch (StringIndexOutOfBoundsException e) {
            System.out.println("StringIndexOutOfBoundsException");
          }
        }
      }
```

Output:

```
StringIndexOutOfBoundsException
```

3.5 USER DEFINED EXCEPTION

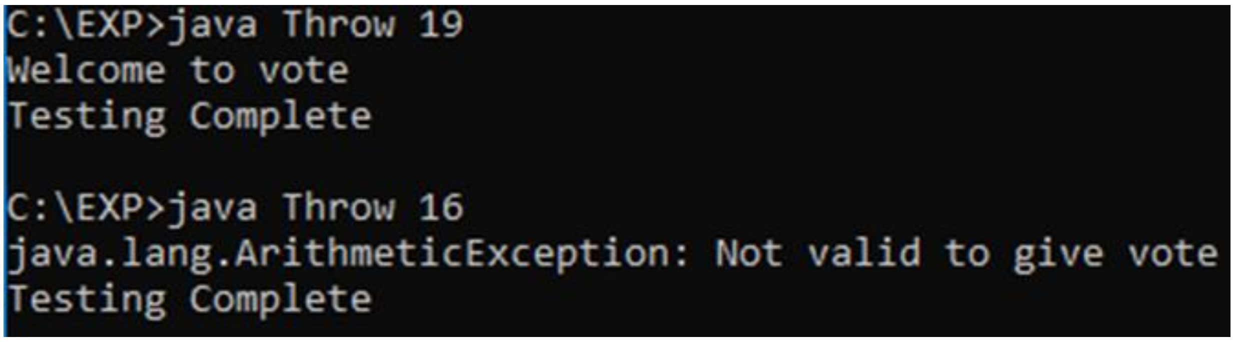
- Java provides us the facility to create our own exceptions which are basically derived classes of Exception. Creating our own Exception is known as a custom exception or user-defined exception.
- Basically, Java custom exceptions are used to customize the exception according to user needs.
- In simple words, we can say that a User-Defined Exception or custom exception is creating your own exception class and throwing that exception using the 'throw' keyword.

3.5.1 Example of java custom exception.

```
class myexception extends Exception
 {
 myexception(String s)
    {
    super(s);
    }
}
class Throws
 {
 static void validage(int age) throws myexception
```

```
    {
    if(age<18)
      {
       throw new myexception("Not valid to give vote");
      }
    else
     {
      System.out.println("Welcome to vote");
     }
   }
public static void main(String args[])
  {
  try
   {
   validage(Integer.parseInt(args[0]));
   }
  catch(myexception my)
    {
    System.out.println(my);
    }
  System.out.println("Testing Complete");
}
}
```

OUTPUT



```
C:\EXP>java Throw 19
Welcome to vote
Testing Complete

C:\EXP>java Throw 16
java.lang.ArithmeticException: Not valid to give vote
Testing Complete
```

3.6 MULTITHREADED PROGRAMMING:

Multithreading

- ➢ Multithreading in java is a process of executing multiple threads simultaneously.
- ➢ A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.
- ➢ However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
- ➢ Java Multithreading is mostly used in games, animation, etc.

Differences between Multithreading and Multitasking

- ➢ Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:
  - Process-based Multitasking (Multiprocessing)
  - Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

- o Each process has an address in memory. In other words, each process allocates a separate memory area.
- o A process is heavyweight.
- o Cost of communication between the process is high.
- o Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

- o Threads share the same address space.

- A thread is lightweight.
- Cost of communication between the thread is low.

Note: At least one process is required for each thread

3.6.1 JAVA THREAD MODEL

Thread Life Cycle

- A thread can be in one of the five states. The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:
  - New
  - Runnable
  - Running
  - Non-Runnable (Blocked)
  - Terminated

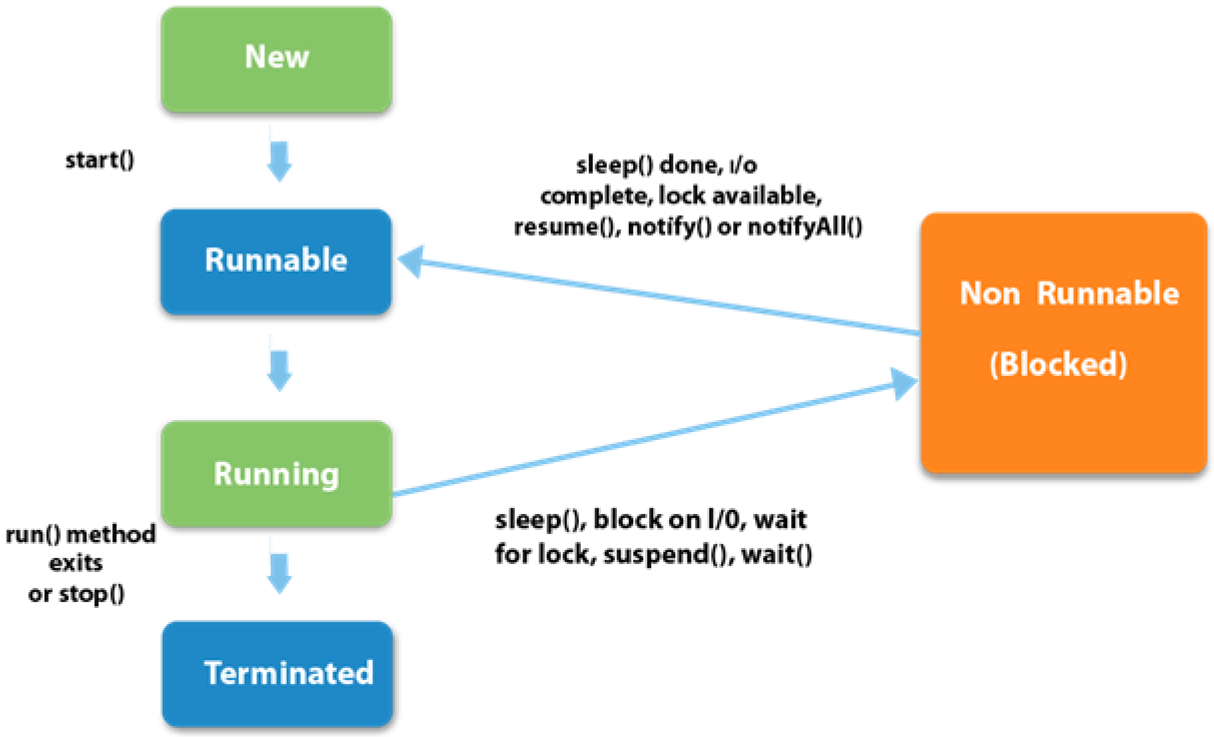

New
start()
Runnable
sleep() done, I/o complete, lock available, resume(), notify() or notifyAll()
Non Runnable (Blocked)
Running
sleep(), block on I/0, wait for lock, suspend(), wait()
run() method exits or stop()
Terminated

Fig 1. Thread Life Cycle

New

- The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

Runnable

- The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

Running

- The thread is in running state if the thread scheduler has selected it.

Non-Runnable (Blocked)

- This is the state when the thread is still alive, but is currently not eligible to run.

Terminated

- A thread is in terminated or dead state when its run() method exits.

3.7 CREATING A THREAD AND MULTIPLE THREADS

- There are two ways to create a thread:
    - By extending Thread class
    - By implementing Runnable interface.

Java Thread Example by extending Thread class

```
class Multi extends Thread{
public void run()
{
System.out.println("thread is running...");
}
public static void main(String args[])
{
Multi t1=new Multi();
t1.start();
 }
}
```

Output:

```
thread is running...
```

Java Thread Example by implementing Runnable interface

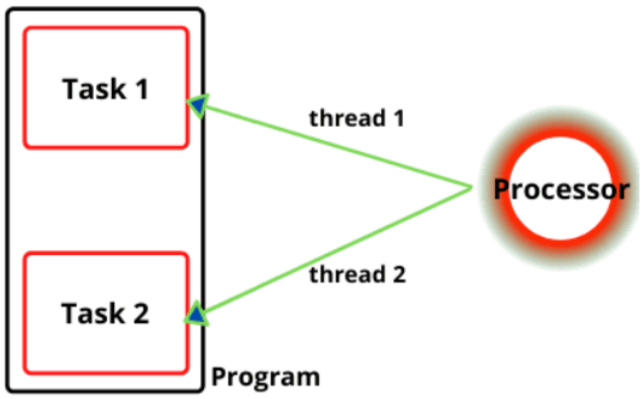
```
class Multi3 implements Runnable
{
public void run()
{
System.out.println("thread is running...");
}
public static void main(String args[])
{
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

Output:

```
thread is running...
```

3.7.2 Creating Multiple Threads in Java

- Creating more than one thread to perform multiple tasks is called multithreading in Java. In multiple threading programming, multiple threads are executing simultaneously that improves the performance of CPU because CPU is not idle if other threads are waiting to get some resources.
- Multiple threads share the same address space in the heap memory. Therefore, It is good to create multiple threads to execute multiple tasks rather than creating multiple processes.



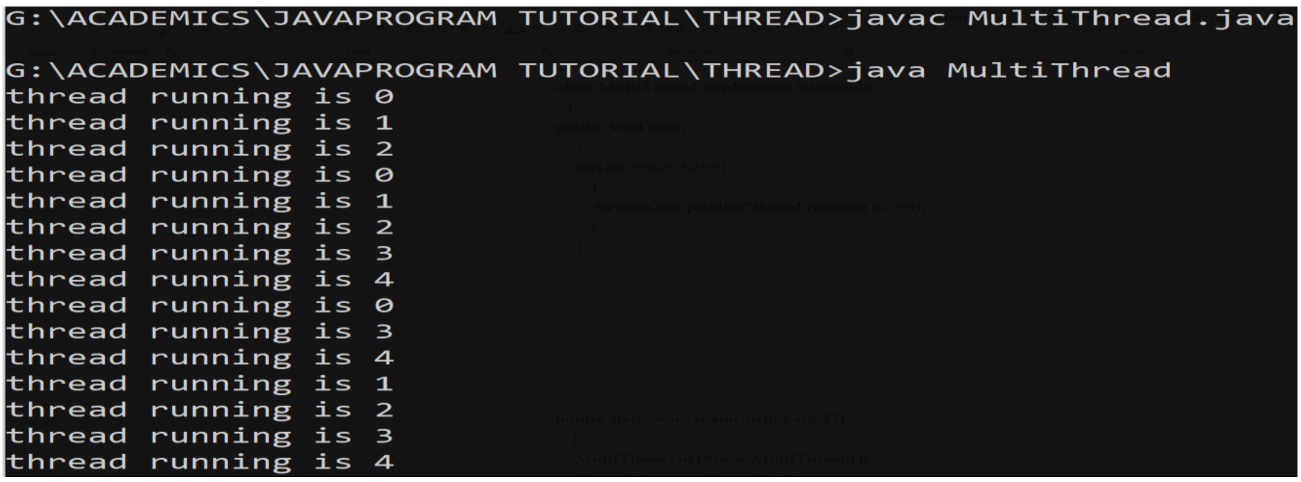
Example Program

// Two threads performing same tasks at a time.

//By implementing Runnable Interface

```
class MultiThread implements Runnable
 {
public void run()
  {
  for(int i=0;i<5;i++)
    {
     System.out.println("thread running is "+i);
    }
  }
public static void main(String args[])
 {
  MultiThread m1=new MultiThread();
  Thread t1 =new Thread(m1);
  Thread t2= new Thread(m1);
  Thread t3= new Thread(m1);
  t1.start();
  t2.start();
  t3.start();
 }
}
```

OUTPUT



```
G:\ACADEMICS\JAVAPROGRAM TUTORIAL\THREAD>javac MultiThread.java

G:\ACADEMICS\JAVAPROGRAM TUTORIAL\THREAD>java MultiThread
thread running is 0
thread running is 1
thread running is 2
thread running is 0
thread running is 1
thread running is 2
thread running is 3
thread running is 4
thread running is 0
thread running is 3
thread running is 4
thread running is 1
thread running is 2
thread running is 3
thread running is 4
```

3.8 PRIORITIES

- Thread priority in Java is a number assigned to a thread that is used by Thread scheduler to decide which thread should be allowed to execute.
- In Java, each thread is assigned a different priority that will decide the order (preference) in which it is scheduled for running.
- Thread priorities are represented by a number from 1 to 10 that specifies the relative priority of one thread to another.
- The default priority of a thread is 5. Thread class in Java also provides several priority constants to define the priority of a thread. These are:

  1. MIN_PRIORITY = 1
  2. NORM_PRIORITY = 5
  3. MAX_PRIORTY = 10
- Thread scheduler selects the thread for execution on the first-come, first-serve basis. That is, the threads having equal priorities share the processor time on the first-come first-serve basis.
- When multiple threads are ready for execution, the highest priority thread is selected and executed by JVM.
- In case when a high priority thread stops, or enters the blocked state, a low priority thread starts executing.
- If any high priority thread enters the runnable state, it will preempt the currently running thread forcing it to move to the runnable state. Note that the highest priority thread always preempts any lower priority thread.

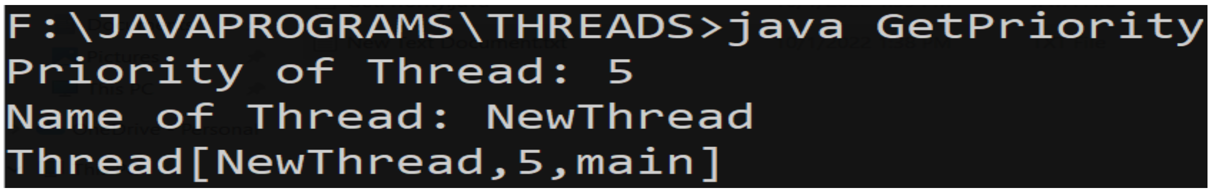
3.8.1 How to get Priority of Current Thread in Java?

- Thread class provides a method named **getPriority()** that is used to determine the priority of a thread. It returns the priority of a thread through which it is called.

3.8.1.1 Let's create a Java program in which we will determine the priority and name of the current thread.

```
public class A implements Runnable
{
public void run()
{
  System.out.println(Thread.currentThread()); // This method is static.
}
public static void main(String[] args)
{
 A a = new A();
 Thread t = new Thread(a, "NewThread");
 System.out.println("Priority of Thread: " +t.getPriority());
 System.out.println("Name of Thread: " +t.getName());
 t.start();
  }
}
```

OUTPUT



```
F:\JAVAPROGRAMS\THREADS>java GetPriority
Priority of Thread: 5
Name of Thread: NewThread
Thread[NewThread,5,main]
```

3.8.2 How to set Priority of Thread in Java?

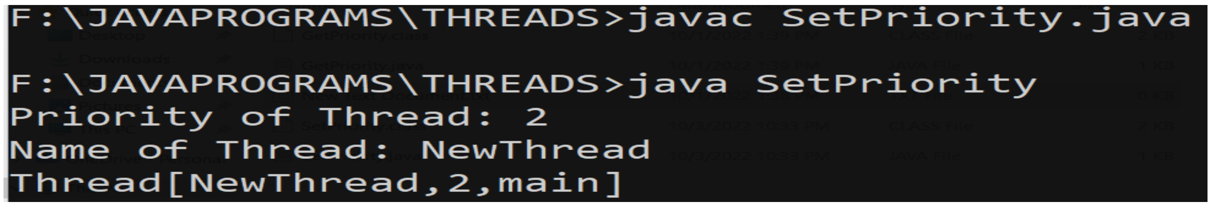
- The **setPriority()** of Thread class is used to set the priority of a thread. This method accepts an integer value as an argument and sets that value as priority of a thread through which it is called. The syntax to set the priority of a thread is as follows:
    - **ThreadName.setPriority(n);**

      where, n is an integer value which ranges from 1 to 10.

3.8.2.1 Example Program

```
public class A implements Runnable
{
public void run()
{
  System.out.println(Thread.currentThread()); // This method is static.
}
public static void main(String[] args)
{
 A a = new A();
 Thread t = new Thread(a, "NewThread");
 t.setPriority(2); // Setting the priority of thread.
 System.out.println("Priority of Thread: " +t.getPriority());
 System.out.println("Name of Thread: " +t.getName());
 t.start();
  }
}
```

OUTPUT

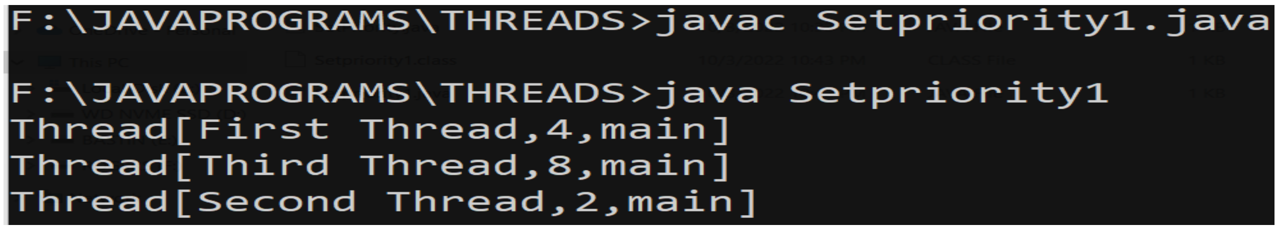


```
F:\JAVAPROGRAMS\THREADS>javac SetPriority.java

F:\JAVAPROGRAMS\THREADS>java SetPriority
Priority of Thread: 2
Name of Thread: NewThread
Thread[NewThread,2,main]
```

3.8.2.2 Example Program using setPriority Method in JAVA

```
public class A implements Runnable
{
public void run()
{
  System.out.println(Thread.currentThread()); // This method is static.
}
public static void main(String[] args)
{
 A a = new A();
 Thread t1 = new Thread(a, "First Thread");
 Thread t2 = new Thread(a, "Second Thread");
 Thread t3 = new Thread(a, "Third Thread");
 t1.setPriority(4); // Setting the priority of first thread.
 t2.setPriority(2); // Setting the priority of second thread.
 t3.setPriority(8); // Setting the priority of third thread.
 t1.start();
 t2.start();
 t3.start();
  }
}
```

OUTPUT



```
F:\JAVAPROGRAMS\THREADS>javac Setpriority1.java

F:\JAVAPROGRAMS\THREADS>java Setpriority1
Thread[First Thread,4,main]
Thread[Third Thread,8,main]
Thread[Second Thread,2,main]
```

3.9 SYNCHRONIZATION

➢ Synchronization in java is the capability to control the access of multiple threads to any shared resource. Java Synchronization is better option where we want to allow only one thread to access the shared resource.

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. static synchronization.
2. Inter-thread communication

3.9.1 Mutual Exclusive

➢ Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

- by synchronized method
- by synchronized block
- by static synchronization

Concept of Lock in Java

➢ Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

3.9.1.1 By synchronized method

- If you declare any method as synchronized, it is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.

- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Example Program for Synchronized method

```
class Table
{
synchronized void printTable(int n)
{//synchronized method
      for(int i=1;i<=5;i++)
      {
         System.out.println(n*i);
         try
        {
            Thread.sleep(400);
       }
         catch(Exception e)
         {
            System.out.println(e);
         }
       }
      }
    }
    class MyThread1 extends Thread
    {
      Table t;
      MyThread1(Table t)
      {
        this.t=t;
      }
      public void run()
      {
```

```
    t.printTable(5);
  }
}
class MyThread2 extends Thread
{
  Table t;
  MyThread2(Table t)
  {
    this.t=t;
  }
  public void run()
  {
    t.printTable(100);
  }
}
public class TestSynchronization2
{
  public static void main(String args[])
  {
    Table obj = new Table();//only one object
    MyThread1 t1=new MyThread1(obj);
    MyThread2 t2=new MyThread2(obj);
    t1.start();
    t2.start();
  }
}
```

Output:

```
 5
10
15
20
25
100
200
300
400
500
```

3.9.1.2 By synchronized block

- Synchronized block can be used to perform synchronization on any specific resource of the method.
- Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.
- If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to remember for Synchronized block

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

```
synchronized (object reference expression) {
  //code block
}
```

3.9.1.3 Example of synchronized block

Let's see the simple example of synchronized block.

Program of synchronized block

```
class Table
{
    void printTable(int n)
    {
            synchronized(this)
            {//synchronized block
                    for(int i=1;i<=5;i++)
                    {
                            System.out.println(n*i);
                            try
                            {
                                    Thread.sleep(400);
                            }
                            catch(Exception e)
                            {
                                    System.out.println(e);
                            }
                    }
            }
    }//end of the method
}
class MyThread1 extends Thread
{
    Table t;
```

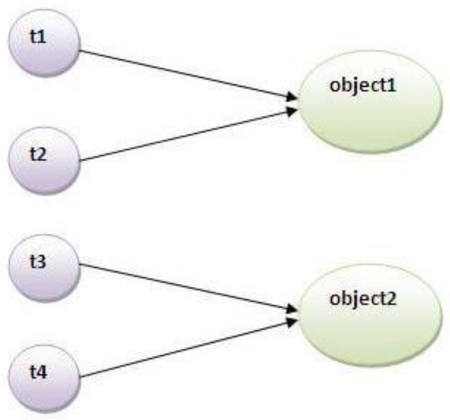
```
        MyThread1(Table t)
        {
                this.t=t;
        }
        public void run()
        {
                t.printTable(5);
        }
    }
    class MyThread2 extends Thread
    {
        Table t;
        MyThread2(Table t)
        {
                this.t=t;
        }
        public void run()
        {
                t.printTable(100);
        }
    }
    public class TestSynchronizedBlock1
    {
        public static void main(String args[])
        {
                Table obj = new Table();//only one object
                MyThread1 t1=new MyThread1(obj);
                MyThread2 t2=new MyThread2(obj);
                t1.start();
                t2.start();
        }
    }
```

Output:

```
 5
10
15
20
25
100
200
300
400
500
```

3.9.1.4 Static synchronization

If you make any static method as synchronized, the lock will be on the class not on object.



Problem without static synchronization

- Suppose there are two objects of a shared class(e.g. Table) named object1 and object2.In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock. But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock.I want no interference between t1 and t3 or t2 and t4.Static synchronization solves this problem.

Example of static synchronization

In this example we are applying synchronized keyword on the static method to perform static synchronization.

```
class Table
{
    synchronized static void printTable(int n)
    {
            for(int i=1;i<=10;i++)
            {
                    System.out.println(n*i);
                    try
                    {
                            Thread.sleep(400);
                    }
                    catch(Exception e)
                    {}
            }
    }
}
```

```
class MyThread1 extends Thread
{
    public void run()
    {
            Table.printTable(1);
    }
}
class MyThread2 extends Thread
{
    public void run()
    {
            Table.printTable(10);
    }
}
class MyThread3 extends Thread
{
    public void run()
    {
            Table.printTable(100);
    }
}
class MyThread4 extends Thread
{
    public void run()
    {
            Table.printTable(1000);
    }
}
public class TestSynchronization4
{
    public static void main(String t[])
    {
            MyThread1 t1=new MyThread1();
```

```
                MyThread2 t2=new MyThread2();
                MyThread3 t3=new MyThread3();
                MyThread4 t4=new MyThread4();
                t1.start();
                t2.start();
                t3.start();
                t4.start();
        }
    }
```

Output:

```
    1
    2
    3
    4
    5
    6
    7
    8
    9
    10
    10
    20
    30
    40
    50
    60
    70
    80
    90
    100
    100
    200
    300
    400
    500
    600
    700
    800
    900
    1000
    1000
    2000
    3000
    4000
```

```
5000
6000
7000
8000
9000
10000
```

3.10 INTER THREAD COMMUNICATION

- Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.
- Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of **Object class**:
  - wait()
  - notify()
  - notifyAll()

1) wait() method

- Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	**Description**
public final void wait()throws InterruptedException	waits until object is notified.
public final void wait(long timeout)throws InterruptedException	waits for the specified amount of time.

2) notify() method

- Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

- **Syntax:**
  - public final void notify()

3) notifyAll() method

- Wakes up all threads that are waiting on this object's monitor.

  Syntax:

  public final void notifyAll()

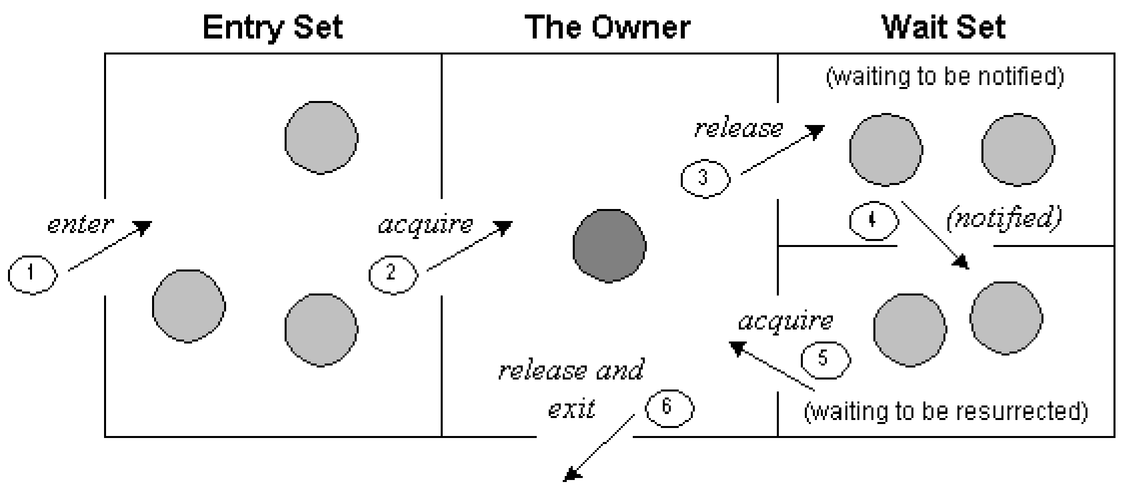


Fig 3. Process of inter-thread communication

The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.
7. wait(), notify() and notifyAll() methods are defined in Object class not Thread class because they are related to lock and object has a lock.

3.10.1 Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

wait()	**sleep()**

wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method
is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.

3.10.2 Example of inter thread communication in java

```
class Customer
{
    int amount=10000;
    synchronized void withdraw(int amount)
    {
            System.out.println("going to withdraw...");
            if(this.amount<amount)
            {
                    System.out.println("Less balance; waiting for deposit...");
                    try
                    {
                            wait();
                    }
                    catch(Exception e)
                    {}
            }
            this.amount-=amount;
            System.out.println("withdraw completed...");
    }
    synchronized void deposit(int amount)
    {
            System.out.println("going to deposit...");
            this.amount+=amount;
            System.out.println("deposit completed... ");
            notify();
```

```
        }
    }
    class Test
    {
        public static void main (String args[])
        {
                final Customer c=new Customer();
                new Thread()
                {
                        public void run()
                        {
                                c.withdraw(15000);
                        }
                }.start();
                new Thread()
                {
                        public void run()
                        {
                                c.deposit(10000);
                        }
                }.start();
        }
    }
```

Output:

```
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed
```

3.11 JAVA THREAD SUSPEND() METHOD

- The suspend() method of thread class puts the thread from running to waiting state.
- This method is used if you want to stop the thread execution and start it again when a certain event occurs.

- This method allows a thread to temporarily cease execution. The suspended thread can be resumed using the resume() method.
- This method does not return any value.

3.11.1 Syntax

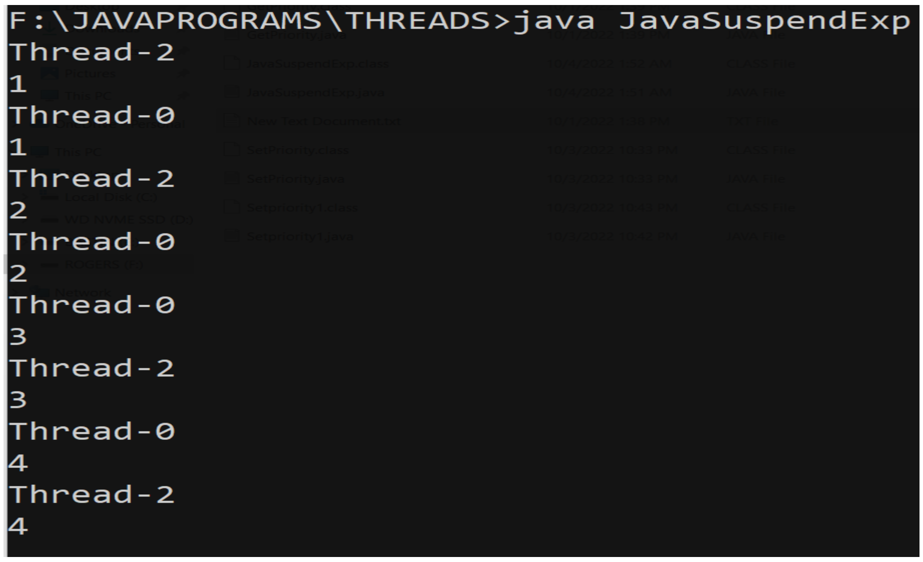
- **public final void suspend()**

3.11.2 Example Program

```
public class JavaSuspendExp extends Thread
{
   public void run()
   {
      for(int i=1; i<5; i++)
      {
         try
         {
            // thread to sleep for 500 milliseconds
             sleep(500);
             System.out.println(Thread.currentThread().getName());
         }
         catch(InterruptedException e)
         {
           System.out.println(e);
          }
         System.out.println(i);
      }
   }
   public static void main(String args[])
   {
      // creating three threads
      JavaSuspendExp t1=new JavaSuspendExp ();
```

```
            JavaSuspendExp t2=new JavaSuspendExp ();
            JavaSuspendExp t3=new JavaSuspendExp ();
            // call run() method
            t1.start();
            t2.start();
            // suspend t2 thread
            t2.suspend();
            // call run() method
            t3.start();
        }
    }
```

OUTPUT



```
F:\JAVAPROGRAMS\THREADS>java JavaSuspendExp
Thread-2
1
Thread-0
1
Thread-2
2
Thread-0
2
Thread-0
3
Thread-2
3
Thread-0
4
Thread-2
4
```

3.12 JAVA THREAD RESUME() METHOD

- The resume() method of thread class is only used with suspend() method. This method is used to resume a thread which was suspended using suspend() method. This method allows the suspended thread to start again.

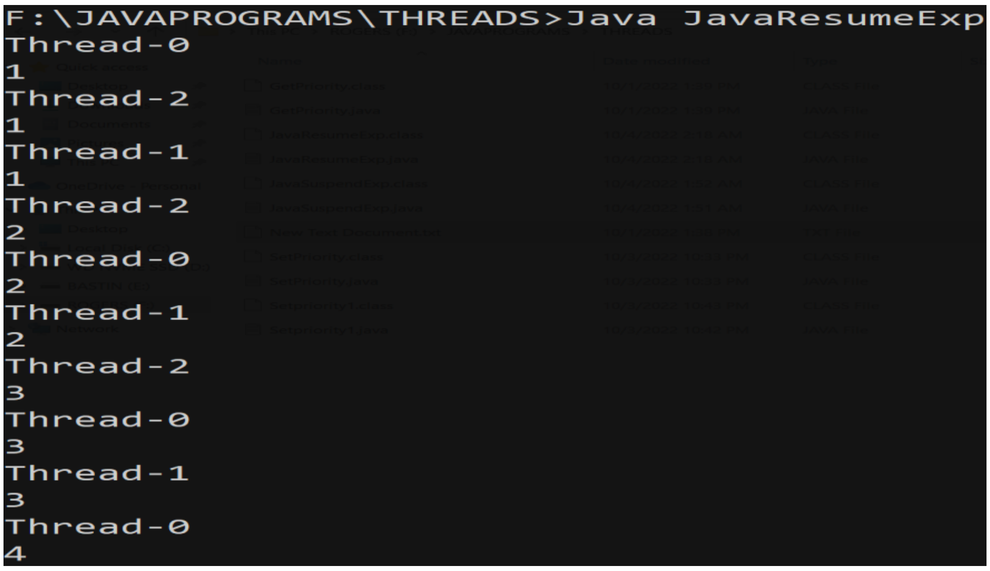
3.12.1 Syntax

- **public final void resume()**

3.12.2 Example Program

```
public class JavaResumeExp extends Thread
{
   public void run()
   {
          for(int i=1; i<5; i++)
          {
             try
             {
                // thread to sleep for 500 milliseconds
                 sleep(500);
                 System.out.println(Thread.currentThread().getName());
             }catch(InterruptedException e)
             {
             System.out.println(e);
             }
             System.out.println(i);
          }
       }
       public static void main(String args[])
       {
          // creating three threads
          JavaResumeExp t1=new JavaResumeExp ();
          JavaResumeExp t2=new JavaResumeExp ();
          JavaResumeExp t3=new JavaResumeExp ();
          // call run() method
          t1.start();
          t2.start();
          t2.suspend(); // suspend t2 thread
          // call run() method
          t3.start();
          t2.resume(); // resume t2 thread
       }
    }
```

OUTPUT



```
F:\JAVAPROGRAMS\THREADS>Java JavaResumeExp
Thread-0
1
Thread-2
1
Thread-1
1
Thread-2
2
Thread-0
2
Thread-1
2
Thread-2
3
Thread-0
3
Thread-1
3
Thread-0
4
```

3.13 JAVA THREAD STOP() METHOD

- The stop() method of thread class terminates the thread execution. Once a thread is stopped, it cannot be restarted by start() method.
- This method does not return any value.

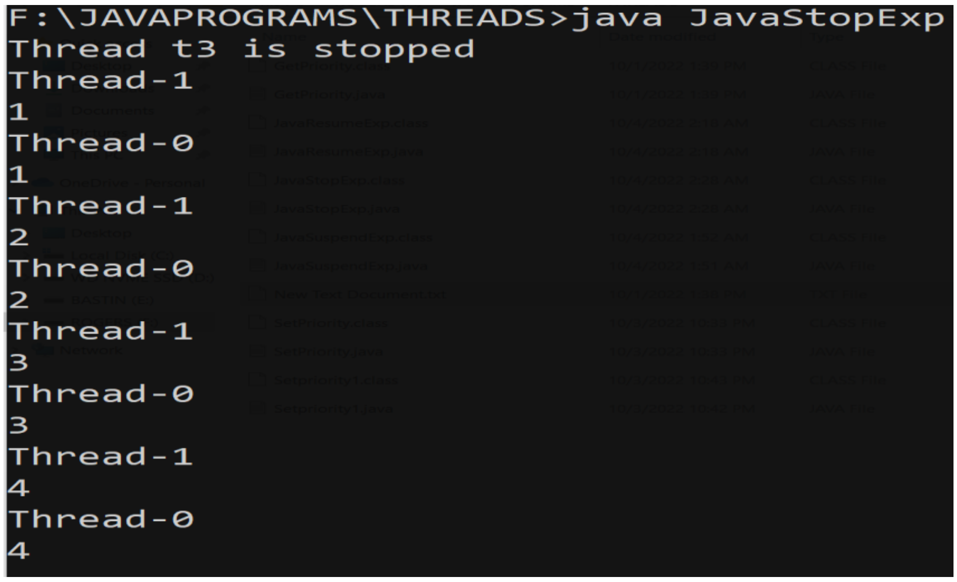
3.13.1 Syntax

- **public final void stop()**
- **public final void stop(Throwable obj)**
    - obj : The Throwable object to be thrown.

3.13.2 Example Program

```
public class JavaStopExp extends Thread
{
    public void run()
    {
        for(int i=1; i<5; i++)
        {
            try
            {
                // thread to sleep for 500 milliseconds
                sleep(500);
                System.out.println(Thread.currentThread().getName());
            }catch(InterruptedException e)
            {
             System.out.println(e);
             }
               System.out.println(i);
            }
        }
        public static void main(String args[])
        {
            // creating three threads
            JavaStopExp t1=new JavaStopExp ();
            JavaStopExp t2=new JavaStopExp ();
            JavaStopExp t3=new JavaStopExp ();
            // call run() method
            t1.start();
            t2.start();
            // stop t3 thread
            t3.stop();
            System.out.println("Thread t3 is stopped");
        }
    }
```

OUTPUT

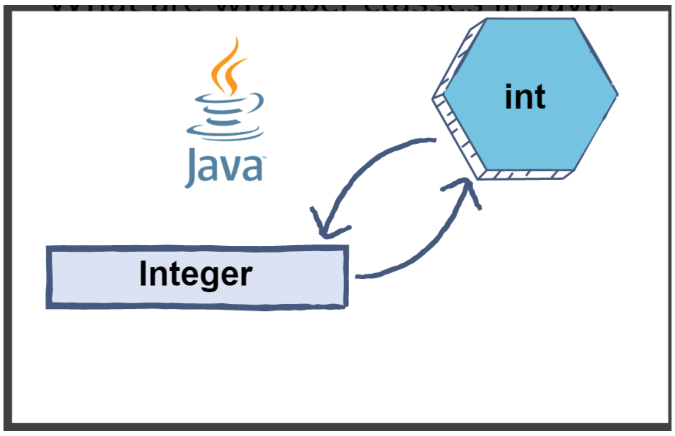


```
F:\JAVAPROGRAMS\THREADS>java JavaStopExp
Thread t3 is stopped
Thread-1
1
Thread-0
1
Thread-1
2
Thread-0
2
Thread-1
3
Thread-0
3
Thread-1
4
Thread-0
4
```

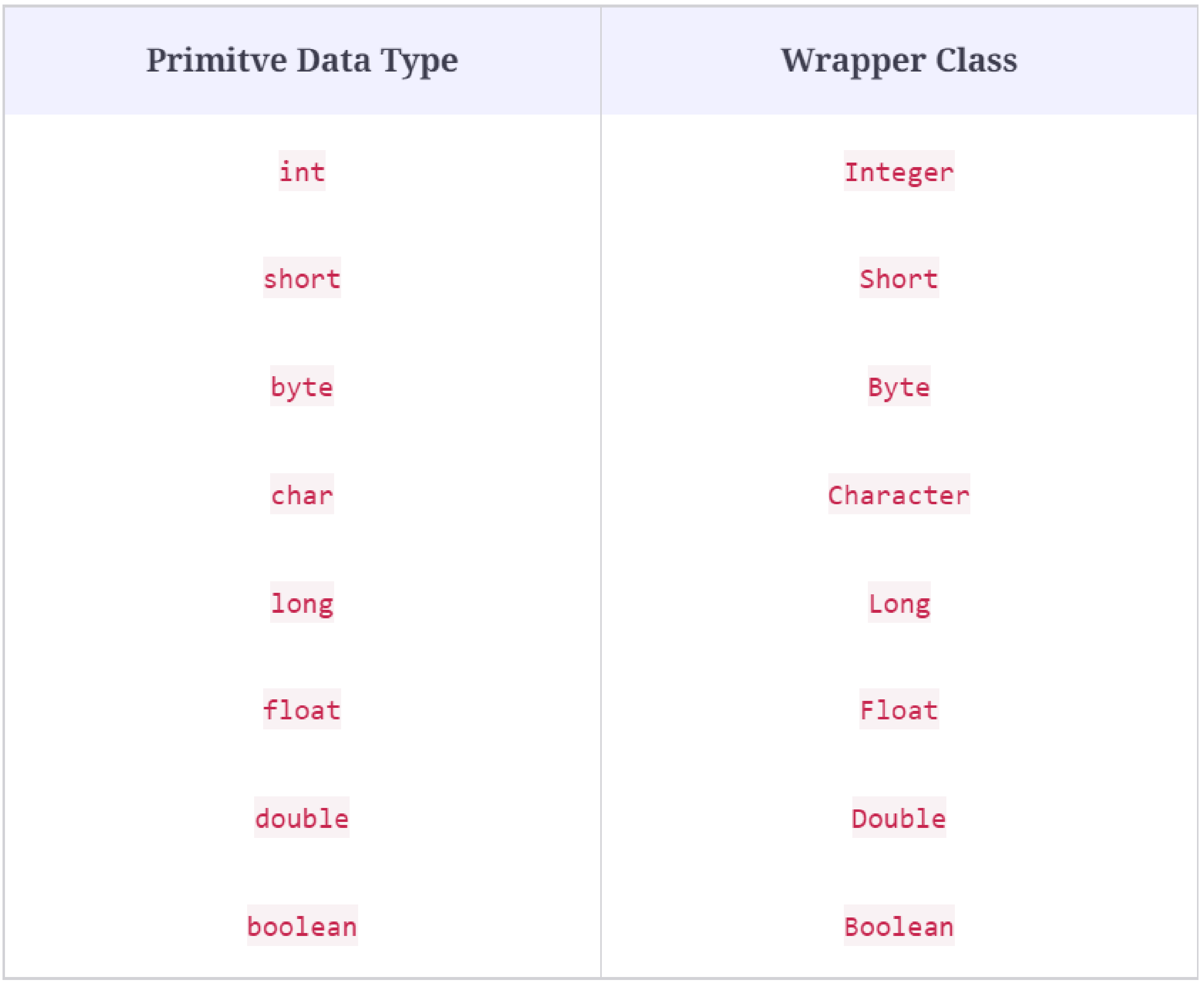
3.14 Multithreading. Wrappers – Auto boxing.

3.14.1 What are wrapper classes in Java?

- A wrapper class in Java converts a primitive data type into class object.
- Java is an object-oriented language that only supports pass by value. Therefore, wrapper class objects allow us to change the original passed value. These wrapper classes help with multithreading and synchronization because, in Java, multithreading only works with objects.



Primitive Data Types and its corresponding Wrapper Classes

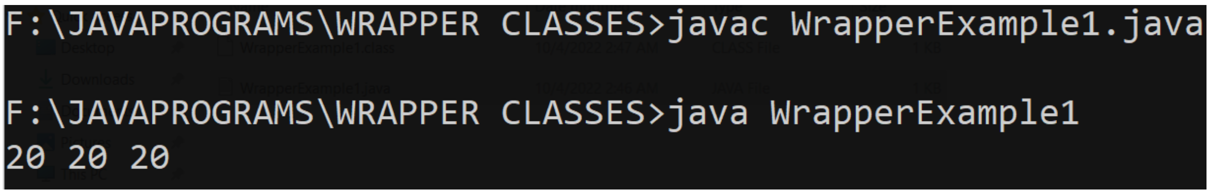


Primitve Data Type	Wrapper Class
`int`	`Integer`
`short`	`Short`
`byte`	`Byte`
`char`	`Character`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`

3.14.1.1 Wrapper class Example: Primitive to Wrapper

```
import java.lang.*;
public class WrapperExample1
{
    public static void main(String args[])
    {
        //Converting int into Integer
        int a=20;
        Integer i=Integer.valueOf(a);//converting int into Integer
        Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
        System.out.println(a+" "+i+" "+j);
    }
}
```

OUTPUT

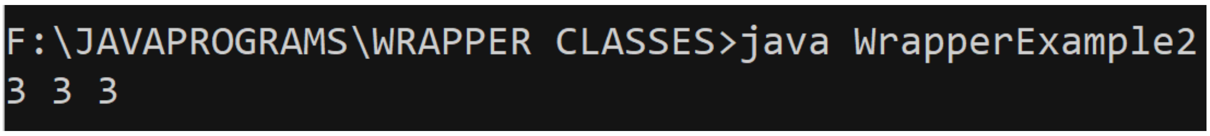


```
F:\JAVAPROGRAMS\WRAPPER CLASSES>javac WrapperExample1.java

F:\JAVAPROGRAMS\WRAPPER CLASSES>java WrapperExample1
20 20 20
```

3.14.1.2 Wrapper class Example: Wrapper to Primitive

```
import java.lang.*;
public class WrapperExample2
{
    public static void main(String args[])
    {
        //Converting Integer to int
        Integer a=new Integer(3);
        int i=a.intValue(); //unboxing i.e converting Integer to int
        int j=a; //auto unboxing, now compiler will write a.intValue() internally
        System.out.println(a+" "+i+" "+j);
    }
}
```

OUTPUT



```
F:\JAVAPROGRAMS\WRAPPER CLASSES>java WrapperExample2
3 3 3
```

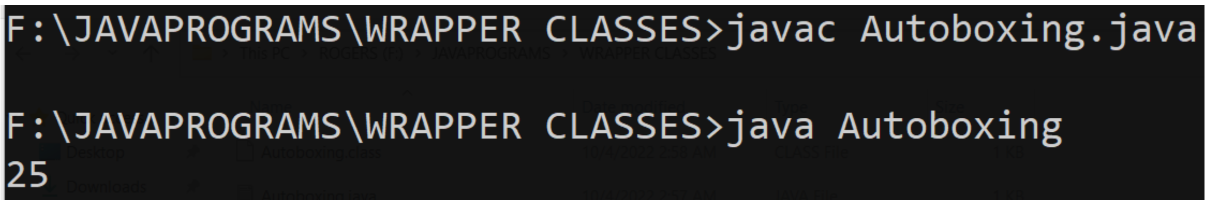
3.14.2 AUTOBOXING

➢ Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double etc.

3.14.2.1 Java program to demonstrate Autoboxing

```
import java.util.ArrayList;
class Autoboxing
{
    public static void main(String[] args)
    {
        char ch = 'a';
        // Autoboxing- primitive to Character object conversion
        Character a = ch;
        ArrayList<Integer> arrayList = new ArrayList<Integer>();
        // Autoboxing because ArrayList stores only objects
        arrayList.add(25);
        // printing the values from object
        System.out.println(arrayList.get(0));
    }
}
```

OUTPUT



```
F:\JAVAPROGRAMS\WRAPPER CLASSES>javac Autoboxing.java

F:\JAVAPROGRAMS\WRAPPER CLASSES>java Autoboxing
25
```

UNIT IV EXCEPTION HANDLING AND I/O

I/O Basics – Reading and Writing Console I/O – Reading and Writing Files. Generics: Generic Programming – Generic classes – Generic Methods – Bounded Types – Restrictions and Limitations. Strings: Basic String class, methods and String Buffer Class.

4.1 INPUT / OUTPUT BASICS

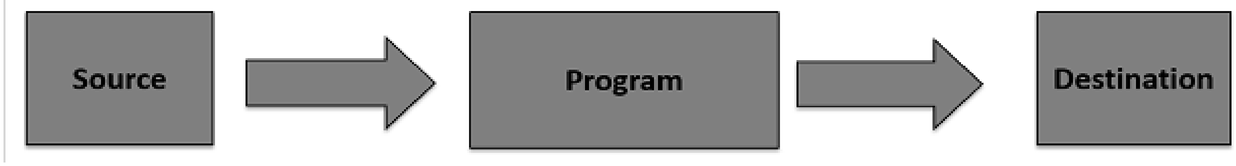
- Java I/O (Input and Output) is used *to process the input* and *produce the output*. Java uses the concept of stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations. We can perform file handling in java by Java I/O API.

4.1.1 STREAM

A stream can be defined as a sequence of data. There are two kinds of Streams −

- InputStream − The InputStream is used to read data from a source.
- OutputStream − The OutputStream is used for writing data to a destination.

In java, 3 streams are created for us automatically. All these streams are attached with console.



1) System.out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream

Let's see the code to print output and error message to the console.

```
System.out.println("simple message");
System.err.println("error message");
```

Let's see the code to get input from console.

```
int i=System.in.read();//returns ASCII code of 1st character
System.out.println((char)i); //will print the character
```

OutputStream vs InputStream

The explanation of OutputStream and InputStream classes are given below:

OutputStream

- Java application uses an output stream to write data to a destination, it may be a file, an array, peripheral device or socket.

InputStream

- Java application uses an input stream to read data from a source, it may be a file, an array, peripheral device or socket.

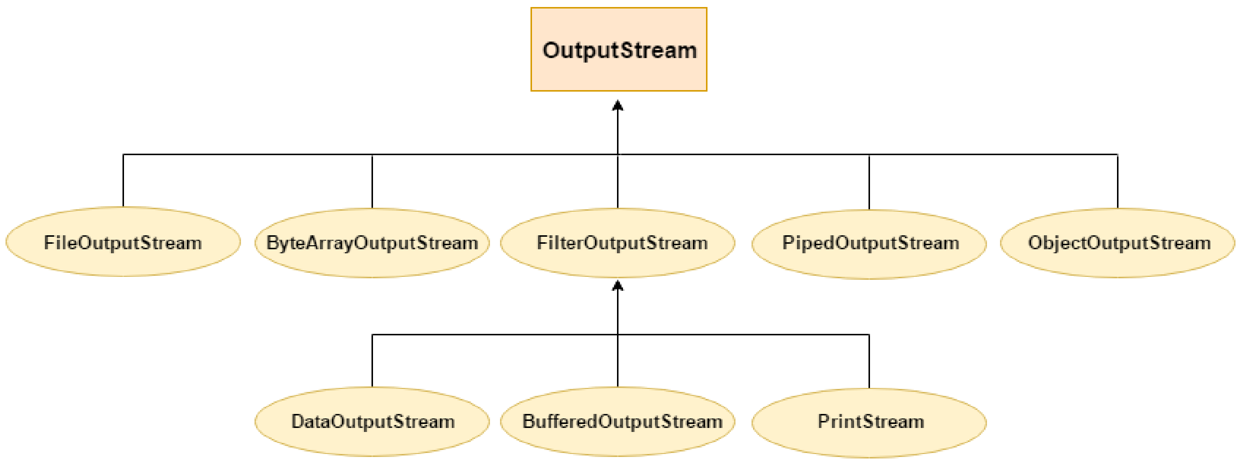
OutputStream class

- OutputStream class is an abstract class. It is the super class of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Useful Methods of OutputStream

Method	Description
1)public void write(int)throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

OutputStream Hierarchy



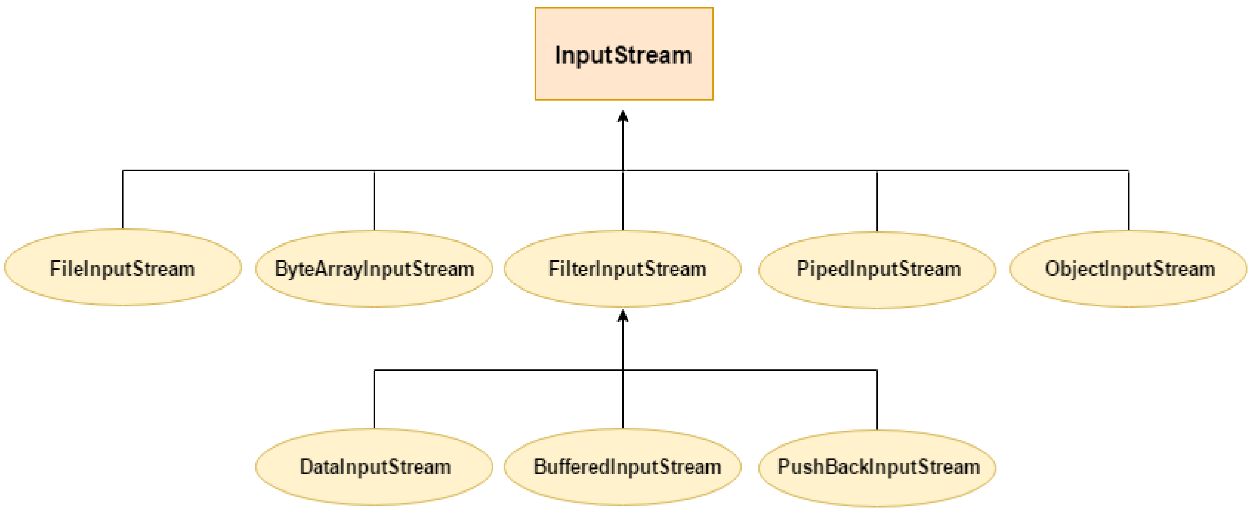
InputStream class

- InputStream class is an abstract class. It is the super class of all classes representing an input stream of bytes.

Useful Methods of InputStream

Method	Description
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of file.
2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

InputStream Hierarchy



4.1.2 Java FileOutputStream Class

- Java FileOutputStream is an output stream used for writing data to a file. If you have to write primitive values into a file, use FileOutputStream class.
- You can write byte-oriented as well as character-oriented data through FileOutputStream class. But, for character-oriented data, it is preferred to use FileWriter than FileOutputStream.

FileOutputStream Class Declaration

Let's see the declaration for Java.io.FileOutputStream class:

```
public class FileOutputStream extends OutputStream
```

FileOutputStream Class Methods

Method	Description
protected void finalize()	It is used to clean up the connection with the file output stream.
void write(byte[] ary)	It is used to write ary.length bytes from the byte array to the file output stream.
void write(byte[] ary, int off, int len)	It is used to write len bytes from the byte array starting at offset off to the file output stream.
void write(int b)	It is used to write the specified byte to the file output stream.
FileChannel getChannel()	It is used to return the file channel object associated with the file output stream.
FileDescriptor getFD()	It is used to return the file descriptor associated with the stream.
void close()	It is used to closes the file output stream.

Java FileOutputStream Example : write byte

```
import java.io.* ;
class File1
{
public static void main(String args[])
{
FileOutputStream fout=null;
byte b1[]={'A','B'};
        try{
          fout=new FileOutputStream("testout.txt");
          fout.write(b1);
          fout.close();
          System.out.println("success...");
         }
catch(Exception e)
{
System.out.println(e);
}
}
}
```

Output:

```
       Success...
The content of a text file testout.txt is set with the data AB.
testout.txt
AB
```

4.1.3 Java FileInputStream Class

- Java FileInputStream class obtains input bytes from a file. It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. You can also read character-stream data. But, for reading streams of characters, it is recommended to use FileReader class.

Java FileInputStream Class Declaration

Let's see the declaration for java.io.FileInputStream class:

```
public class FileInputStream extends InputStream
```

Java FileInputStream Class Methods

Method	Description
int available()	It is used to return the estimated number of bytes that can be read from the input stream.
int read()	It is used to read the byte of data from the input stream.
int read(byte[] b)	It is used to read up to b.length bytes of data from the input stream.

int read(byte[] b, int off, int len)	It is used to read up to len bytes of data from the input stream.
long skip(long x)	It is used to skip over and discards x bytes of data from the input stream.
FileChannel getChannel()	It is used to return the unique FileChannel object associated with the file input stream.
FileDescriptor getFD()	It is used to return the FileDescriptor object.
protected void finalize()	It is used to ensure that the close method is call when there is no more reference to the file input stream.
void close()	It is used to closes the stream.

4.1.4 Java FileInputStream example: read all characters

```
import java.io.* ;
class File2
{
public static void main(String args[])
{
FileInputStream fin=null;
          try
            {
            fin=new FileInputStream("testout.txt");
            int b;
            while((b=fin.read())!=-1)
              {
               System.out.println((char)b);
               }
             fin.close();
            }
        catch(Exception e)
            {
            System.out.println(e);
            }
}
}
```

Output:

Contents in testout.txt will be displayed

4.1.4 BYTE STREAMS AND CHARACTER STREAMS

BYTE STREAMS

Java ByteArrayOutputStream Class

- Java ByteArrayOutputStream class is used to write common data into multiple files. In this stream, the data is written into a byte array which can be written to multiple streams later.
- The ByteArrayOutputStream holds a copy of data and forwards it to multiple streams. The buffer of ByteArrayOutputStream automatically grows according to data.

Java ByteArrayOutputStream Class Declaration

Let's see the declaration for Java.io.ByteArrayOutputStream class:

```
public class ByteArrayOutputStream extends OutputStream
```

Java ByteArrayOutputStream Class Constructors

Constructor	**Description**
ByteArrayOutputStream()	Creates a new byte array output stream with the initial capacity of 32 bytes, though its size increases if necessary.
ByteArrayOutputStream(int size)	Creates a new byte array output stream, with a buffer capacity of the specified size, in bytes.

Java ByteArrayOutputStream class methods

Method	Description
int size()	It is used to returns the current size of a buffer.

byte[] toByteArray()	It is used to create a newly allocated byte array.
String toString()	It is used for converting the content into a string decoding bytes using a platform default character set.
String toString(String charsetName)	It is used for converting the content into a string decoding bytes using a specified charsetName.
void write(int b)	It is used for writing the byte specified to the byte array output stream.
void write(byte[] b, int off, int len	It is used for writing len bytes from specified byte array starting from the offset off to the byte array output stream.
void writeTo(OutputStream out)	It is used for writing the complete content of a byte array output stream to the specified output stream.
void reset()	It is used to reset the count field of a byte array output stream to zero value.
void close()	It is used to close the ByteArrayOutputStream.

Example of Java ByteArrayOutputStream

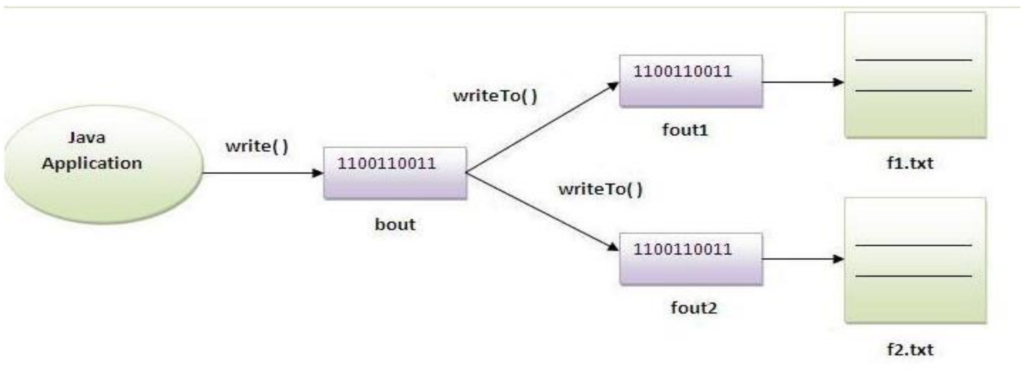
- Let's see a simple example of java ByteArrayOutputStream class to write common data into 2 files: f1.txt and f2.txt.

Example Program

```
import java.io.*;
public class DataStreamExample
{
public static void main(String args[])throws Exception
{
     FileOutputStream fout1=new FileOutputStream("D:\\f1.txt");
    FileOutputStream fout2=new FileOutputStream("D:\\f2.txt");
    ByteArrayOutputStream bout=new ByteArrayOutputStream();
    bout.write(65);
    bout.writeTo(fout1);
    bout.writeTo(fout2);
    bout.flush();
    bout.close();//has no effect
    System.out.println("Success...");
   }
   }
```

Output:

```
Success...
f1.txt:
A
f2.txt:
A
```



4.2 READING AND WRITING CONSOLE I/O

4.2.1 Java Console Class

- The Java Console is a predefined class that is available in io package, it is used to get user input at run time. It provides methods to read texts and passwords. If you read password using Console class, it will not be displayed to the user.
- The java.io.Console class is attached with system console internally.
- Let's see a simple example to read text from console.

```
String text=System.console().readLine();
System.out.println("Text is: "+text);
```

Java Console Class Declaration

Let's see the declaration for Java.io.Console class:

public final class Console extends Object implements Flushable

Java Console Class Methods

Method	Description
Reader reader()	It is used to retrieve the reader object associated with the console
String readLine()	It is used to read a single line of text from the console.
String readLine(String fmt, Object... args)	It provides a formatted prompt then reads the single line of text from the console.
char[] readPassword()	It is used to read password that is not being displayed on the console.
char[] readPassword(String fmt, Object... args)	It provides a formatted prompt then reads the password that is not being displayed on the console.
Console format(String fmt, Object... args)	It is used to write a formatted string to the console output stream.
Console printf(String format, Object... args)	It is used to write a string to the console output stream.
PrintWriter writer()	It is used to retrieve the PrintWriter object associated with the console.
void flush()	It is used to flushes the console.

How to get the object of Console

System class provides a static method console() that returns the singleton instance of Console class.

```
public static Console console()
{}
```

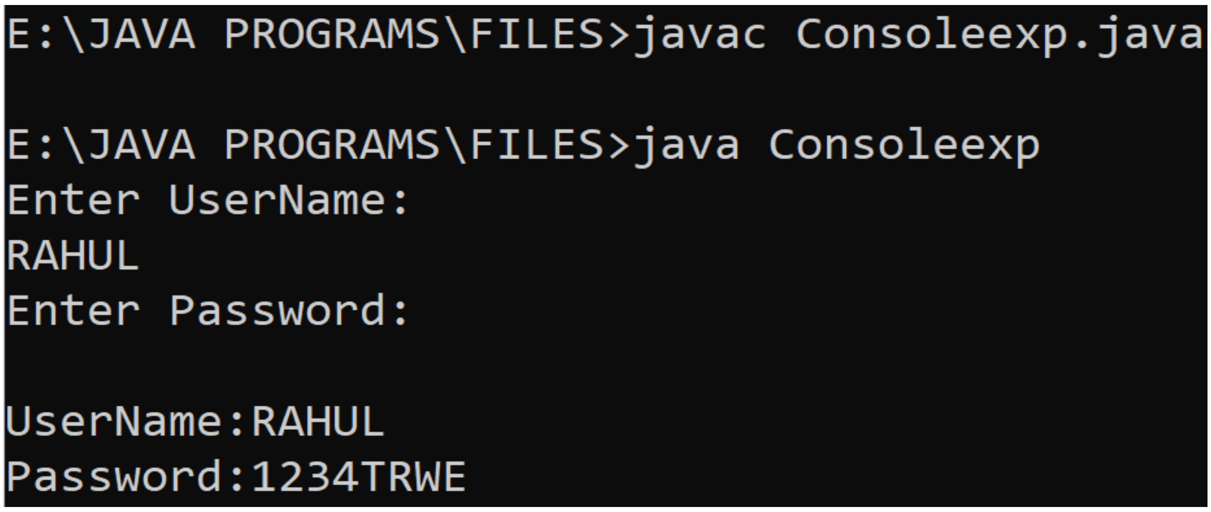
Let's see the code to get the instance of Console class.

```
Console c=System.console();
```

Java Console Example to Read Username and Password

```
import java.io.*;

class Consoleexp

  {

   public static void main(String arg[])

     {

      String str ; char ch[];

      Console C=System.console();

      System.out.println("Enter UserName:");

      str=C.readLine();

      System.out.println("Enter Password:");

      ch=C.readPassword();

      String S=String.valueOf(ch);

      System.out.println("UserName:" +str);

      System.out.println("Password:" +S);

     }

}
```

Output



```
E:\JAVA PROGRAMS\FILES>javac Consoleexp.java

E:\JAVA PROGRAMS\FILES>java Consoleexp
Enter UserName:
RAHUL
Enter Password:

UserName:RAHUL
Password:1234TRWE
```

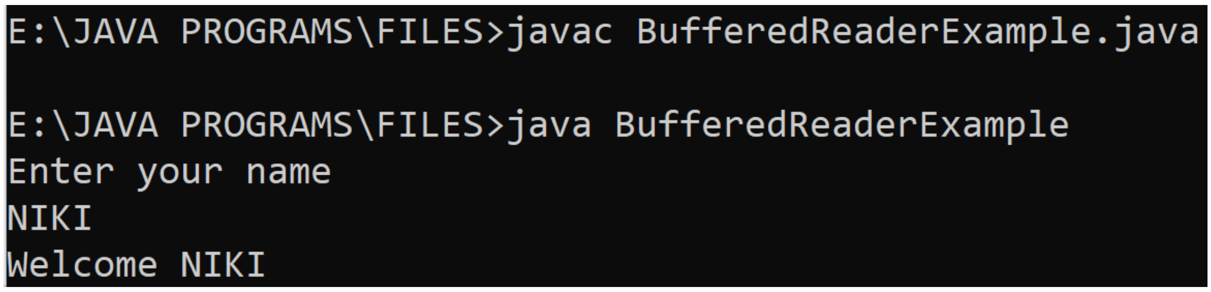
4.2.2 Using Buffered Reader Class

- Java BufferedReader class is used to read the text from a character-based input stream. It can be used to read data line by line by readLine() method.
- InputStreamReader class performs two tasks
  - Read input stream of keyboard.
  - Convert byte streams to character streams.

ExampleProgram: Reading data from console by InputStreamReader and BufferedReader

```
import java.io.*;
class Bufferreader
  {
   public static void main(String arg[]) throws IOException
      {
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter UserName:");
        String name=br.readLine();
        System.out.println("Welcome " +name);
        }
 }
```

Output



```
E:\JAVA PROGRAMS\FILES>javac BufferedReaderExample.java

E:\JAVA PROGRAMS\FILES>java BufferedReaderExample
Enter your name
NIKI
Welcome NIKI
```

4.3 READING AND WRITING FILES

4.3.1 Java Reader

The FileReader class of the java.io package can be used to read data (in characters) from files.

Fields

Modifier and Type	**Field**	**Description**
protected Object	lock	The object used to synchronize operations on this stream.

Constructor

Modifier	**Constructor**	**Description**
Protected	Reader()	It creates a new character-stream reader whose critical sections will synchronize on the reader itself.
Protected	Reader(Object lock)	It creates a new character-stream reader whose critical sections will synchronize on the given object.

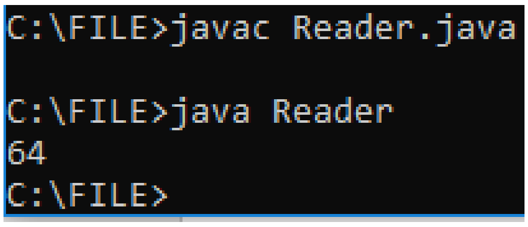
Methods

- read() - reads a single character from the reader
- read(char[] array) - reads the characters from the reader and stores in the specified array
- read(char[] array, int start, int length) - reads the number of characters equal to length from the reader and stores in the specified array starting from the position start

Example

```
import java.io.*;
class Reader
{
public static void main(String[] args)
{
File infile=new File("a2.txt");
FileReader fr=null;
try
{
 fr=new FileReader(infile);
 int ch;
 while ((ch=fr.read())!= -1)
   {
   System.out.print((char)ch);
   }
 }
catch (Exception e)
{
 System.out.println(e);
}
 }
 }
```

OUTPUT



```
C:\FILE>javac Reader.java

C:\FILE>java Reader
64
C:\FILE>
```

4.3.2 JAVA FILEWRITER CLASS

Java FileWriter class is used to write character-oriented data to a file. It is character-oriented class which is used for file handling in java.

Unlike FileOutputStream class, you don't need to convert string into byte array because it provides method to write string directly.

Java FileWriter class declaration

Let's see the declaration for Java.io.FileWriter class:

```
public class FileWriter extends OutputStreamWriter
```

Constructors of FileWriter Class

Constructor	**Description**
FileWriter(String file)	Creates a new file. It gets file name in string.
FileWriter(File file)	Creates a new file. It gets file name in File object.

Methods of FileWriter Class

Method	**Description**
void write(String text)	It is used to write the string into FileWriter.
void write(char c)	It is used to write the char into FileWriter.
void write(char[] c)	It is used to write char array into FileWriter.
void flush()	It is used to flushes the data of FileWriter.
void close()	It is used to close the FileWriter.

Java FileWriter Example

In this example, we are writing the data in the file testout.txt using Java FileWriter class.

```
import java.io.*;
class Writer
{
 public static void main(String args[])
{
File outfile=new File("A2.txt");
FileWriter fout=null;
try
{
```

```
fout=new FileWriter(outfile);
String s="WELCOME TO JAVA";
fout.write(s);
fout.close();
}
catch(Exception e)
{
System.out.println(e);
}
 System.out.println("Success...");
 }
}
```

Output:

```
Success...
A2.txt:
WELCOME TO JAVA
```

4.4 Generic programming

- Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.
- Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.
- Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

4.4.1 Generic Methods

- We can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define **Generic Methods −**

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

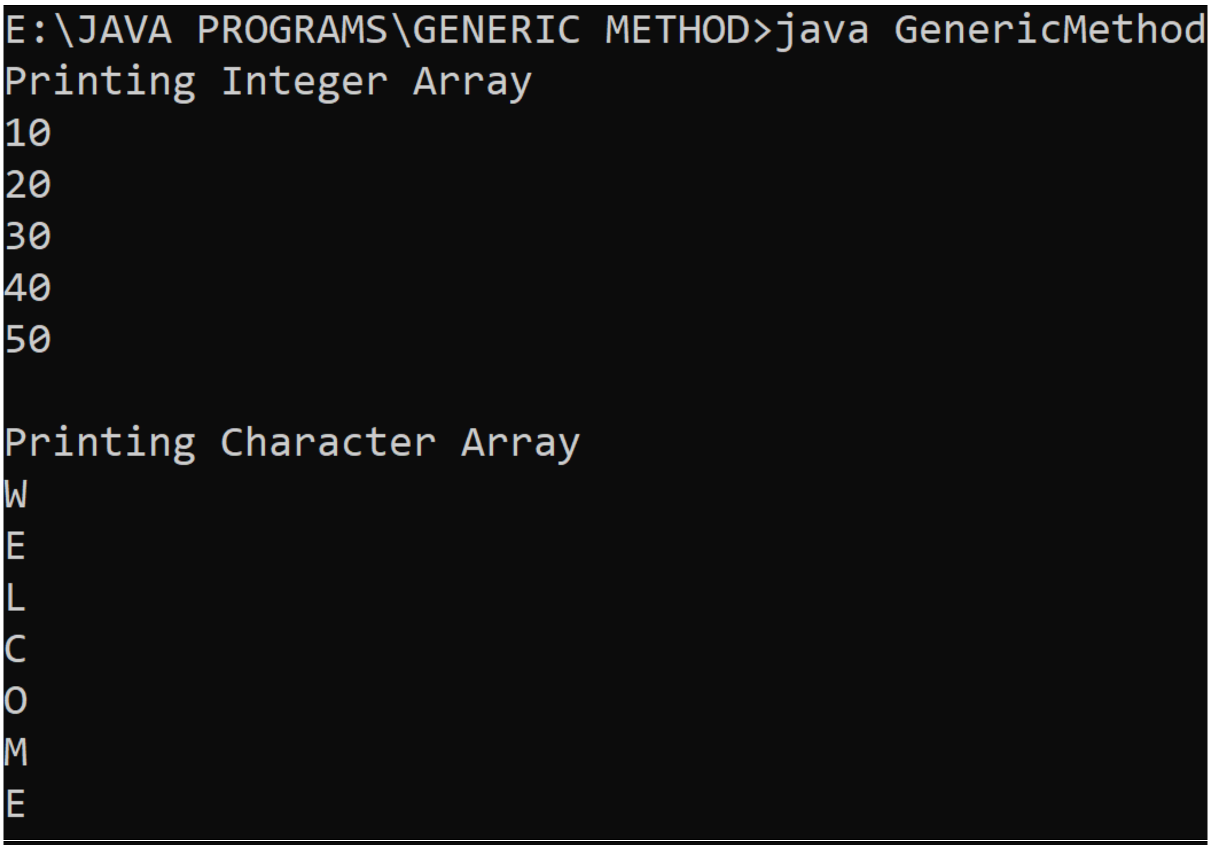
Example

- Following example illustrates how we can print an array of different type using a single Generic method –

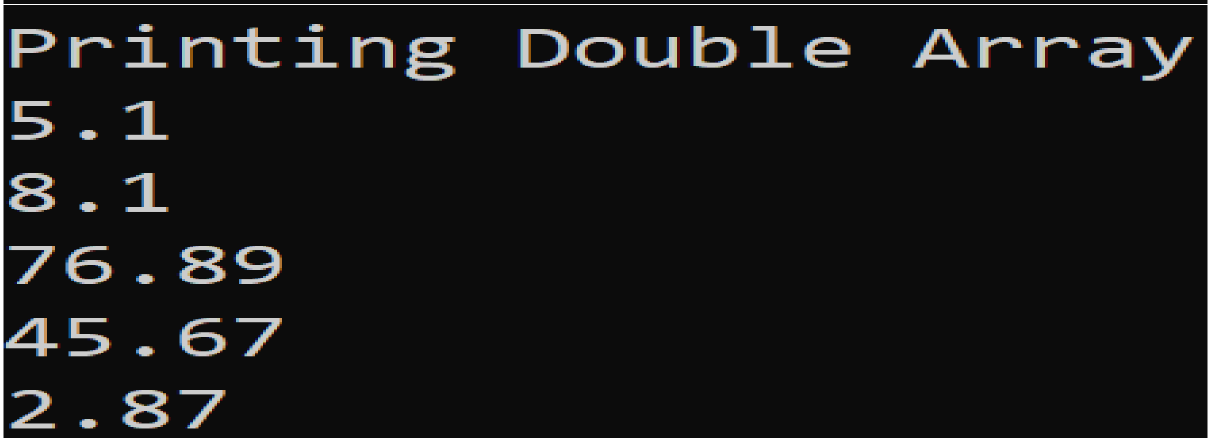
```
public class GenericMethod
 {
  public static < E > void printArray(E[] elements)
     {
     for ( E element : elements)
      {
        System.out.println(element );
      }
      System.out.println();
  }
  public static void main( String args[] )
  {
     Integer[] intArray = { 10, 20, 30, 40, 50 };
     Character[] charArray = { 'W','E','L','C','O','M','E' };
     Double[] doubleArray={5.10,8.10,76.89,45.67,2.87};
     System.out.println( "Printing Integer Array" );
     printArray( intArray  );
     System.out.println( "Printing Character Array" );
```

```
        printArray( charArray );
        System.out.println( "Printing Double Array" );
        printArray( doubleArray );
    }
}
```

Output



```
E:\JAVA PROGRAMS\GENERIC METHOD>java GenericMethod
Printing Integer Array
10
20
30
40
50

Printing Character Array
W
E
L
C
O
M
E
```



```
Printing Double Array
5.1
8.1
76.89
45.67
2.87
```

4.4.2 Bounded Type Parameters

- Whenever you want to restrict the type parameter to subtypes of a particular class you can use the bounded type parameter.
- If you just specify a type (class) as bounded parameter, only sub types of that particular class are accepted by the current generic class. These are known as bounded-types in generics in Java.
- For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

4.4.2.1 Defining bounded-types for class

- You can declare a bound parameter just by extending the required class with the type-parameter, within the angular braces as
- Syntax

  class Sample <T extends Number>

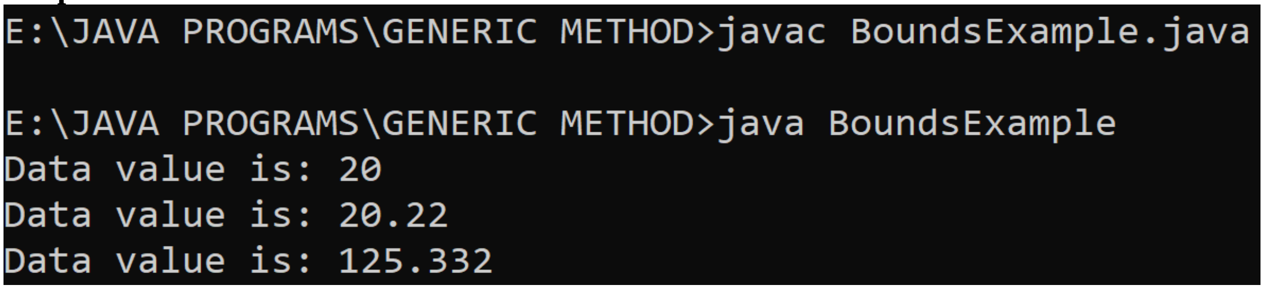
Example

- In the following Java example, the generic class Sample restricts the type parameter to the sub classes of the Number classes using the bounded parameter.

```
 class Sample <T extends Number>
  {
   T data;
   Sample(T data)
  {
     this.data = data;
   }
   public void display()
  {
     System.out.println("Data value is: "+this.data);
   }
}
public class BoundsExample
  {
   public static void main(String args[])
```

```
    {
     Sample<Integer> obj1 = new Sample<Integer>(20);
     obj1.display();
     Sample<Double> obj2 = new Sample<Double>(20.22d);
     obj2.display();
     Sample<Float> obj3 = new Sample<Float>(125.332f);
     obj3.display();
   }
}
```

Output



```
E:\JAVA PROGRAMS\GENERIC METHOD>javac BoundsExample.java

E:\JAVA PROGRAMS\GENERIC METHOD>java BoundsExample
Data value is: 20
Data value is: 20.22
Data value is: 125.332
```

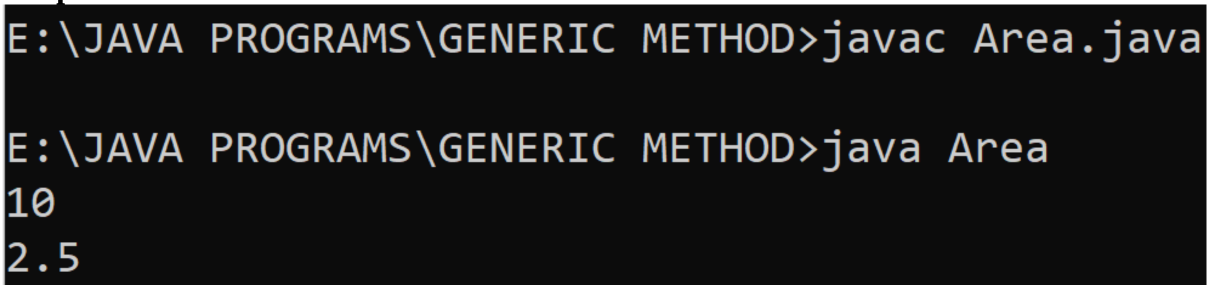
4.4.3 Generic Classes

- A Generic class simply means that the items or functions in that class can be generalized with the parameter(example T) to specify that we can add any type as a parameter in place of T like Integer, Character, String, Double or any other user-defined type.

Example Program

```
public class Area<T>
   {
    // T is the Datatype like String,
    // Integer of which Parameter type,
    // the class Area is of
    private T t;
    public void add(T t)
    {
        // this.t specify the t variable inside
        // the Area Class whereas the right hand
        // side t simply specify the value as the
        // parameter of the function add()
        this.t = t;
    }
    public T get()
    {
    return t;
    }
    public void getArea()
    {
    }
    public static void main(String[] args)
    {
        // Object of generic class Area with parameter Type
        // as Integer
        Area<Integer> rectangle = new Area<Integer>();
        // Object of generic class Area with parameter Type
        // as Double
        Area<Double> circle = new Area<Double>();
        rectangle.add(10);
        circle.add(2.5);
        System.out.println(rectangle.get());
        System.out.println(circle.get());
    }
}
```

Output



```
E:\JAVA PROGRAMS\GENERIC METHOD>javac Area.java

E:\JAVA PROGRAMS\GENERIC METHOD>java Area
10
2.5
```

4.4.5 Restrictions and limitations

To use Java generics effectively, we must consider the following restrictions:

- Cannot Instantiate Generic Types with Primitive Types
- Cannot Create Instances of Type Parameters
- Cannot Declare Static Fields Whose Types are Type Parameters
- Cannot Use Casts or instanceof With Parameterized Types
- Cannot Create Arrays of Parameterized Types

- Cannot Create, Catch, or Throw Objects of Parameterized Types
- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

4.9.1 Cannot Instantiate Generic Types with Primitive Types

Consider the following parameterized type:

```
class Pair<K, V> {
   private K key;
   private V value;
   public Pair(K key, V value) {
      this.key = key;
      this.value = value;
   }
   // ...
}
```

When creating a Pair object, you cannot substitute a primitive type for the type parameter K or V:

Pair<**int, char**> p = new Pair<>(8, 'a'); // compile-time error

You can substitute only non-primitive types for the type parameters K and V:

Pair<**Integer, Character**> p = new Pair<>(8, 'a');

Note that the Java compiler autoboxes 8 to Integer.valueOf(8) and 'a' to Character('a'):

Pair<Integer, Character> p = new Pair<>(Integer.valueOf(8), new Character('a'));

4.9.2 Cannot Create Instances of Type Parameters

We cannot create an instance of a type parameter. For example, the following code causes a compile-time error:

```
public static <E> void append(List<E> list) {
   E elem = new E();  // compile-time error
   list.add(elem);
}
```

As a workaround, you can create an object of a type parameter through reflection:

```
public static <E> void append(List<E> list, Class<E> cls) throws Exception {
   E elem = cls.newInstance();   // OK
   list.add(elem);
}
```

We can invoke the append method as follows:

```
List<String> ls = new ArrayList<>();
append(ls, String.class);
```

4.9.3 Cannot Declare Static Fields Whose Types are Type Parameters

A class's static field is a class-level variable shared by all non-static objects of the class. Hence, static fields of type parameters are not allowed. Consider the following class:

```
public class MobileDevice<T> {
    private static T os;

    // ...
}
```

If static fields of type parameters were allowed, then the following code would be confused:

```
MobileDevice<Smartphone> phone = new MobileDevice<>();
MobileDevice<Pager> pager = new MobileDevice<>();
MobileDevice<TabletPC> pc = new MobileDevice<>();
```

Because the static field os is shared by phone, pager, and pc, what is the actual type of os? It cannot be Smartphone, Pager, and TabletPC at the same time. You cannot, therefore, create static fields of type parameters.

4.9.4 Cannot Use Casts or instanceof with Parameterized Types

Because the Java compiler erases all type parameters in generic code, you cannot verify which parameterized type for a generic type is being used at runtime:

```
public static <E> void rtti(List<E> list) {
    if (list instanceof ArrayList<Integer>) {  // compile-time error
        // ...
    }
}
```

The set of parameterized types passed to the rtti method is:

S = { ArrayList<Integer>, ArrayList<String> LinkedList<Character>, ... }

The runtime does not keep track of type parameters, so it cannot tell the difference between an ArrayList<Integer> and an ArrayList<String>. The most you can do is to use an unbounded wildcard to verify that the list is an ArrayList:

```
public static void rtti(List<?> list) {
    if (list instanceof ArrayList<?>) {  // OK; instanceof requires a reifiable type
        // ...
    }
}
```

Typically, you cannot cast to a parameterized type unless it is parameterized by unbounded wildcards. For example:

```
List<Integer> li = new ArrayList<>();
List<Number>  ln = (List<Number>) li;  // compile-time error
```

However, in some cases the compiler knows that a type parameter is always valid and allows the cast. For example:

```
List<String> l1 = ...;
ArrayList<String> l2 = (ArrayList<String>)l1;  // OK
```

4.9.5 Cannot Create Arrays of Parameterized Types

You cannot create arrays of parameterized types. For example, the following code does not compile:

```
List<Integer>[] arrayOfLists = new List<Integer>[2];  // compile-time error
```

The following code illustrates what happens when different types are inserted into an array:

```
Object[] strings = new String[2];
strings[0] = "hi";   // OK
strings[1] = 100;    // An ArrayStoreException is thrown.
```

If you try the same thing with a generic list, there would be a problem:

```
Object[] stringLists = new List<String>[];  // compiler error, but pretend it's allowed
stringLists[0] = new ArrayList<String>();   // OK
stringLists[1] = new ArrayList<Integer>();  // An ArrayStoreException should be thrown,
                                            // but the runtime can't detect it.
```

If arrays of parameterized lists were allowed, the previous code would fail to throw the desired ArrayStoreException.

4.9.6 Cannot Create, Catch, or Throw Objects of Parameterized Types

A generic class cannot extend the Throwable class directly or indirectly. For example, the following classes will not compile:

```
// Extends Throwable indirectly
class MathException<T> extends Exception { /* ... */ }    // compile-time error
// Extends Throwable directly
```

```
class QueueFullException<T> extends Throwable { /* ... */ // compile-time error
```

A method cannot catch an instance of a type parameter:

```
public static <T extends Exception, J> void execute(List<J> jobs) {
   try {
       for (J job : jobs)
          // ...
   } catch (T e) {   // compile-time error
       // ...
   }
}
```

You can, however, use a type parameter in a throws clause:

```
class Parser<T extends Exception> {
   public void parse(File file) throws T {     // OK
       // ...
   }
}
```

4.9.7 Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

A class cannot have two overloaded methods that will have the same signature after type erasure.

```
public class Example {
   public void print(Set<String> strSet) { }
   public void print(Set<Integer> intSet) { }
}
```

The overloads would all share the same classfile representation and will generate a compile-time error.

4.6 Strings

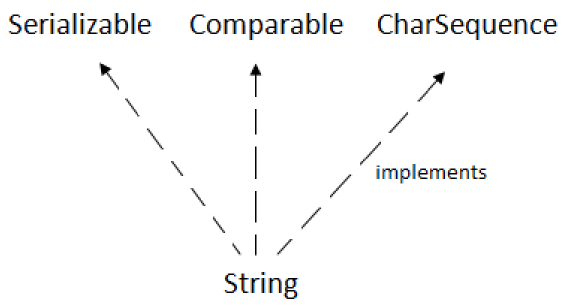
In java, string is basically an object that represents sequence of char values. An array of characters works same as java string. For example:

```
     char[] ch={'j','a','v','a','t','p','o','i','n','t'};
     String s=new String(ch);
```

is same as:

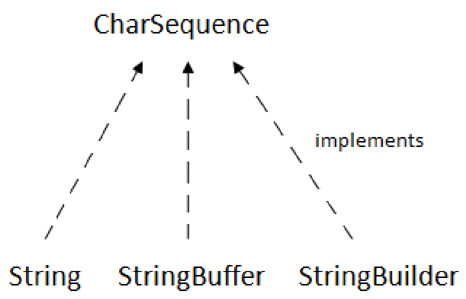
```
          String s="javatpoint";
```

Java String class provides a lot of methods to perform operations on string such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc. Thejava.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.



CharSequence Interface

The CharSequence interface is used to represent sequence of characters. It is implemented by String, StringBuffer and StringBuilder classes. It means, we can create string in java by using these 3 classes.



The java String is immutable i.e. it cannot be changed. Whenever we change any string, a new instance is created. For mutable string, you can use StringBuffer and StringBuilder classes.

There are two ways to create String object:

1. By string literal
2. By new keyword

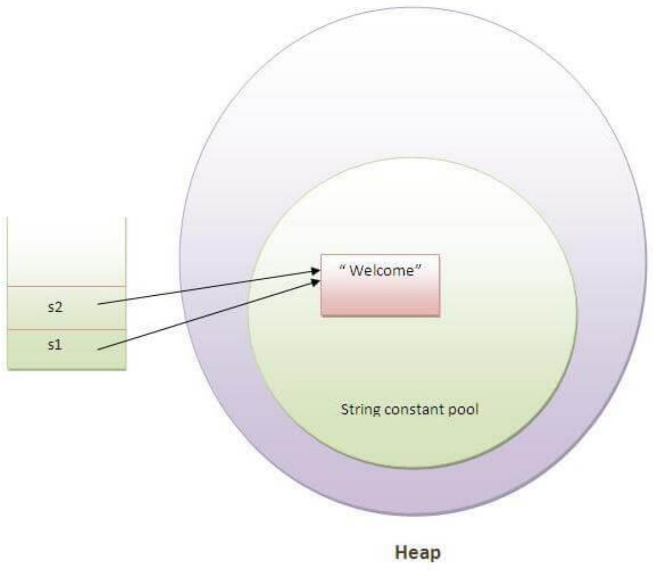
4.6.1 By String literal

Java String literal is created by using double quotes. For Example:

1. String s="welcome";

Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance is returned. If string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
String s1="Welcome";
String s2="Welcome";//will not create new instance
```



In the above example only one object will be created. Firstly JVM will not find any string object with the value "Welcome" in string constant pool, so it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create new object but will return the reference to the same instance.

4.6.2 By new keyword

String s=**new** String("Welcome");//creates two objects and one reference variable

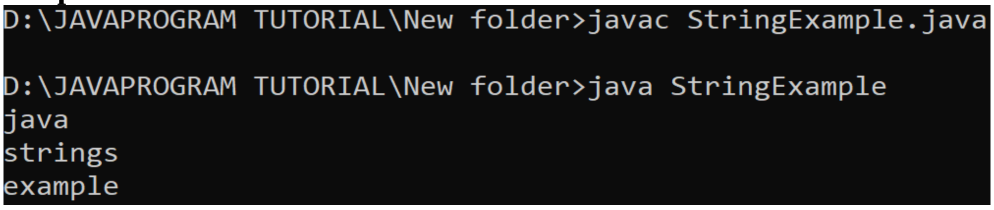
In such case, JVM will create a new string object in normal(non-pool) heap memory and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in heap(non-pool).

Example

```
public class StringExample
{
        public static void main(String args[])
        {
                String s1="java";//creating string by java string literal
                char ch[]={'s','t','r','i','n','g','s'};
                String s2=new String(ch);//converting char array to string
                String s3=new String("example");//creating java string by new keyword
                System.out.println(s1);
                System.out.println(s2);
                System.out.println(s3);
        }
```

}

Output



```
D:\JAVAPROGRAM TUTORIAL\New folder>javac StringExample.java

D:\JAVAPROGRAM TUTORIAL\New folder>java StringExample
java
strings
example
```

4.6.3 String methods

Method	Description
char charAt(int index)	returns char value for the particular index
int length()	returns string length
static String format(String format, Object... args)	returns formatted string
static String format(Locale l, String format, Object... args)	returns formatted string with given locale
String substring(int beginIndex)	returns substring for given begin index
String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index
boolean contains(CharSequence s)	returns true or false after matching the sequence of char value
static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string
static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	returns a joined string
boolean equals(Object another)	checks the equality of string with object
boolean isEmpty()	checks if string is empty
String concat(String str)	concatinates specified string
String replace(char old, char new)	replaces all occurrences of specified char value
String replace(CharSequence old, CharSequence new)	replaces all occurrences of specified CharSequence

static String equalsIgnoreCase(String another)	compares another string. It doesn't check case.
String[] split(String regex)	returns splitted string matching regex
String[] split(String regex, int limit)	returns splitted string matching regex and limit
String intern()	returns interned string
int indexOf(int ch)	returns specified char value index
int indexOf(int ch, int fromIndex)	returns specified char value index starting with given index
int indexOf(String substring)	returns specified substring index
int indexOf(String substring, int fromIndex)	returns specified substring index starting with given index
String toLowerCase()	returns string in lowercase.
String toLowerCase(Locale l)	returns string in lowercase using specified locale.
String toUpperCase()	returns string in uppercase.
String toUpperCase(Locale l)	returns string in uppercase using specified locale.
String trim()	removes beginning and ending spaces of this string.
static String valueOf(int value)	converts given type into string. It is overloaded.

4.6.4 JAVA STRINGBUFFER CLASS

- Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

IMPORTANT CONSTRUCTORS OF STRINGBUFFER CLASS

Constructor	Description
StringBuffer()	It creates an empty String buffer with the initial capacity of 16.
StringBuffer(String str)	It creates a String buffer with the specified string..

StringBuffer(int capacity)	It creates an empty String buffer with the specified capacity as length.

IMPORTANT METHODS OF STRINGBUFFER CLASS

Modifier and Type	Method	Description
public synchronized StringBuffer	append(String s)	It is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public synchronized StringBuffer	insert(int offset, String s)	It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public synchronized StringBuffer	replace(int startIndex, int endIndex, String str)	It is used to replace the string from specified startIndex and endIndex.
public synchronized StringBuffer	delete(int startIndex, int endIndex)	It is used to delete the string from specified startIndex and endIndex.
public synchronized StringBuffer	reverse()	is used to reverse the string.
public int	capacity()	It is used to return the current capacity.
public void	ensureCapacity(int minimumCapacity)	It is used to ensure the capacity at least equal to the given minimum.
public char	charAt(int index)	It is used to return the character at the specified position.
public int	length()	It is used to return the length of the string i.e. total number of characters.

public String	substring(int beginIndex)	It is used to return the substring from the specified beginIndex.
public String	substring(int beginIndex,int endIndex)	It is used to return the substring from the specified beginIndex and endIndex.

WHAT IS A MUTABLE STRING?

- A String that can be modified or changed is known as mutable String. StringBuffer and StringBuilder classes are used for creating mutable strings.

STRING BUFFER CLASS METHODS

1) StringBuffer Class append() Method

- The append() method concatenates the given argument with this String.

Example Program

```
class StringBufferExample
{
public static void main(String args[])
{
StringBuffer sb=new StringBuffer("Hello ");
sb.append("Java"); //now original string is changed
System.out.println(sb); //prints Hello Java
}
}
```

OUTPUT

```
Hello Java
```

2. StringBuffer insert() Method

- The insert() method inserts the given String with this string at the given position.

Example Program

```
class StringBufferExample2{

public static void main(String args[])

{

StringBuffer sb=new StringBuffer("Hello ");

sb.insert(1,"Java");//now original string is changed

System.out.println(sb);//prints HJavaello

}

}
```

OUTPUT

```
HJavaello
```

3. StringBuffer replace() Method

- The replace() method replaces the given String from the specified beginIndex and endIndex.

Example Program

```
class StringBufferExample3{
public static void main(String args[])
{
StringBuffer sb=new StringBuffer("Hello");
sb.replace(1,3,"Java");
System.out.println(sb); //prints HJavalo
}
}
```

OUTPUT

```
HJavalo
```

4. StringBuffer delete() Method

- The delete() method of the StringBuffer class deletes the String from the specified beginIndex to endIndex.

Example Program

```
class StringBufferExample4{
public static void main(String args[])
{
StringBuffer sb=new StringBuffer("Hello");
sb.delete(1,3);
System.out.println(sb); //prints Hlo
}
}
```

OUTPUT

```
Hlo
```

5.StringBuffer reverse() Method

- The reverse() method of the StringBuilder class reverses the current String.

Example Program

```
class StringBufferExample5{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.reverse();
System.out.println(sb);// prints olleH
}
}
```

OUTPUT

olleH

6.StringBuffer capacity() Method

- The capacity() method of the StringBuffer class returns the current capacity of the buffer.

- The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

Example Program

```
class StringBufferExample6{
public static void main(String args[])
{
StringBuffer sb=new StringBuffer();
System.out.println(sb.capacity());//default 16
sb.append("Hello");
System.out.println(sb.capacity());//now 16
sb.append("java is my favourite language");
System.out.println(sb.capacity());  //now (16*2)+2=34 i.e (oldcapacity*2)+2
}
}
```

Output:

```
16
16
34
```

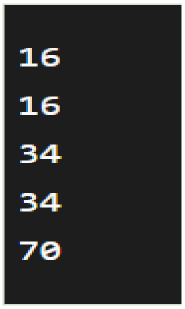
7.StringBuffer ensureCapacity() method

- The ensureCapacity() method of the StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

Example Program

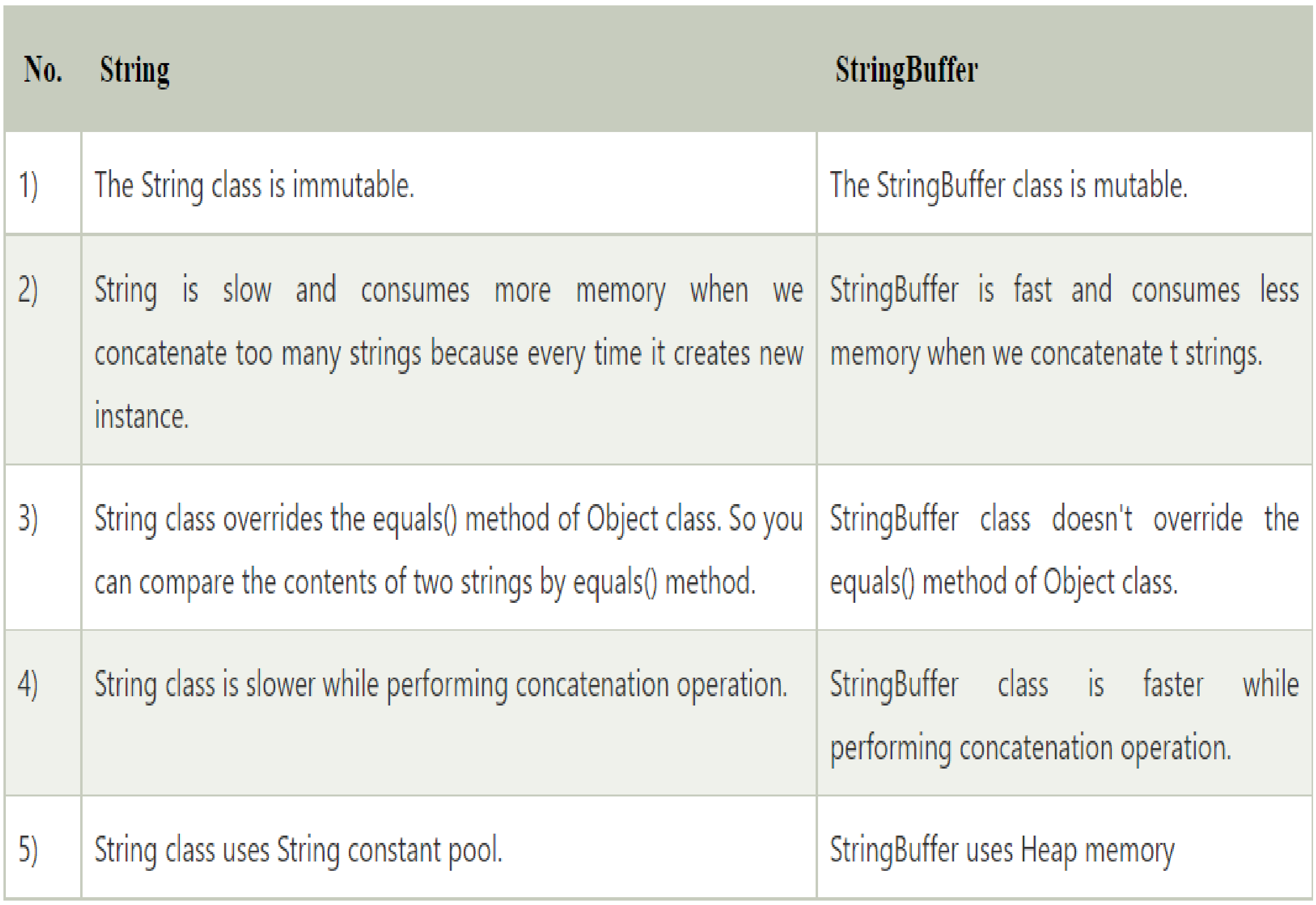
```
class StringBufferExample7
{
public static void main(String args[])
{
StringBuffer sb=new StringBuffer();
System.out.println(sb.capacity());  //default 16
sb.append("Hello");
System.out.println(sb.capacity());  //now 16
sb.append("java is my favourite language");
System.out.println(sb.capacity());  //now (16*2)+2=34 i.e (oldcapacity*2)+2
sb.ensureCapacity(10); //now no change
System.out.println(sb.capacity()); //now 34
sb.ensureCapacity(50);  //now (34*2)+2
System.out.println(sb.capacity());  //now 70
}
}
```

Output



```
16
16
34
34
70
```

Difference between String and StringBuffer



No.	String	StringBuffer
1)	The String class is immutable.	The StringBuffer class is mutable.
2)	String is slow and consumes more memory when we concatenate too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when we concatenate t strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.
4)	String class is slower while performing concatenation operation.	StringBuffer class is faster while performing concatenation operation.
5)	String class uses String constant pool.	StringBuffer uses Heap memory

UNIT V

JAVAFX EVENT HANDLING, CONTROLS AND COMPONENTS

JAVAFX Events and Controls: Event Basics – Handling Key and Mouse Events. Controls: Checkbox, ToggleButton – RadioButtons – ListView – ComboBox – ChoiceBox – Text Controls – ScrollPane. Layouts – FlowPane – HBox and VBox – BorderPane – StackPane – GridPane. Menus – Basics – Menu – Menu bars – MenuItem.

What is JavaFX?

- JavaFX is a Java library used to develop Desktop applications as well as Rich Internet Applications (RIA). The applications built in JavaFX, can run on multiple platforms including Web, Mobile and Desktops.

History of JavaFX

- JavaFX was developed by Chris Oliver. Initially the project was named as Form Follows Functions (F3). It is intended to provide the richer functionalities for the GUI application development. Later, Sun Micro-systems acquired F3 project as JavaFX in June 2005.
- Sun Micro-systems announces it officially in 2007 at W3 Conference. In October 2008, JavaFX 1.0 was released. In 2009, ORACLE corporation acquires Sun Micro-Systems and released JavaFX 1.2. the latest version of JavaFX is JavaFX 1.8 which was released on 18th March 2014.

5.3 JavaFX Controls

5.3.1 CHECKBOX

- The Check Box is used to provide more than one choice to the user. It can be used in a scenario where the user is prompted to select more than one option or the user wants to select multiple options.
- It is different from the radiobutton in the sense that, we can select more than one checkboxes in a scenerio.
- Instantiate **javafx.scene.control.CheckBox** class to implement CheckBox.

States of CheckBox:

- Checked: When indeterminate is false and checked is true
- Unchecked: When indeterminate is false and checked is false
- Undefined: When indeterminate is true

Constructor of the class are:

CheckBox() : Creates a check box with an empty string for its label.

CheckBox(String t) : Creates a check box with the given text as its label.

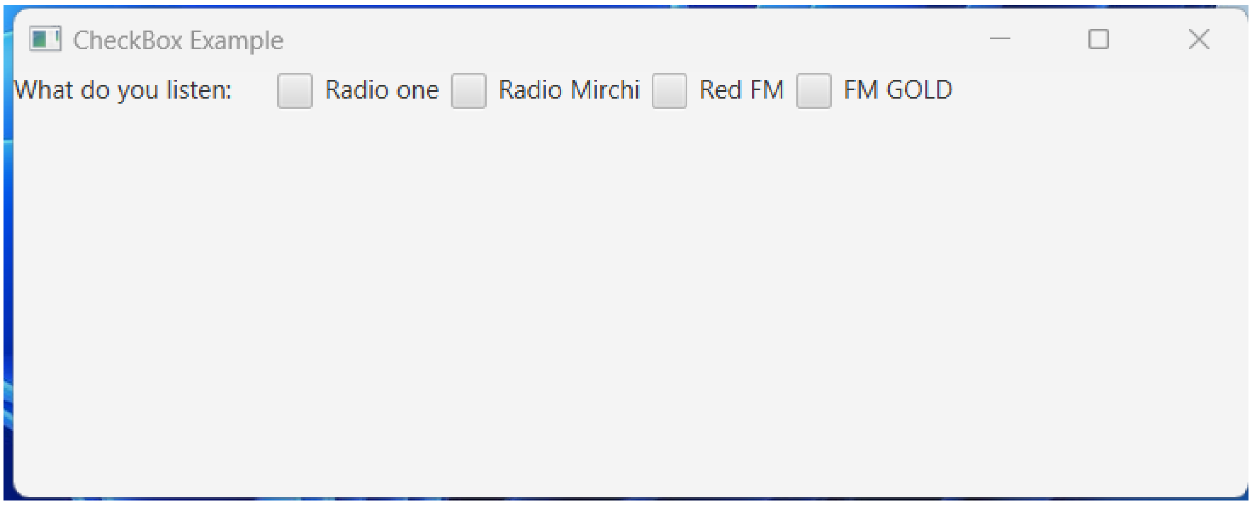
Commonly used methods:

Method	Explanation
isIndeterminate()	Gets the value of the property indeterminate.
isSelected()	Gets the value of the property selected.
selectedProperty()	Indicates whether this CheckBox is checked.
setIndeterminate(boolean v)	Sets the value of the property indeterminate.
setSelected(boolean v)	Sets the value of the property selected.

Example Program

```
package CONTROLS;

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.CheckBox;

import javafx.scene.control.Label;

import javafx.scene.layout.HBox;

import javafx.stage.Stage;

public class CHECKBOX extends Application {

public static void main(String[] args) {

launch(args);
}

@Override

public void start(Stage primaryStage) throws Exception {

    // TODO Auto-generated method stub

    Label l = new Label("What do you listen: ");

    CheckBox c1 = new CheckBox("Radio one");

    CheckBox c2 = new CheckBox("Radio Mirchi");

    CheckBox c3 = new CheckBox("Red FM");

    CheckBox c4 = new CheckBox("FM GOLD");

    HBox root = new HBox();

    root.getChildren().addAll(l,c1,c2,c3,c4);

    root.setSpacing(5);

    Scene scene=new Scene(root,800,200);

    primaryStage.setScene(scene);

    primaryStage.setTitle("CheckBox Example");

    primaryStage.show();

}

}
```

OUTPUT



5.3.2 TOGGLEBUTTON

- A JavaFX ToggleButton is a button that can be selected or not selected. Like a button that stays in when you press it, and when you press it the next time it comes out again. Toggled - not toggled.
- The JavaFX ToggleButton is represented by the class **javafx.scene.control.ToggleButton** .

Creating a ToggleButton

- You create a JavaFX ToggleButton by creating an instance of the ToggleButton class. Here is an example of creating a JavaFX ToggleButton instance:

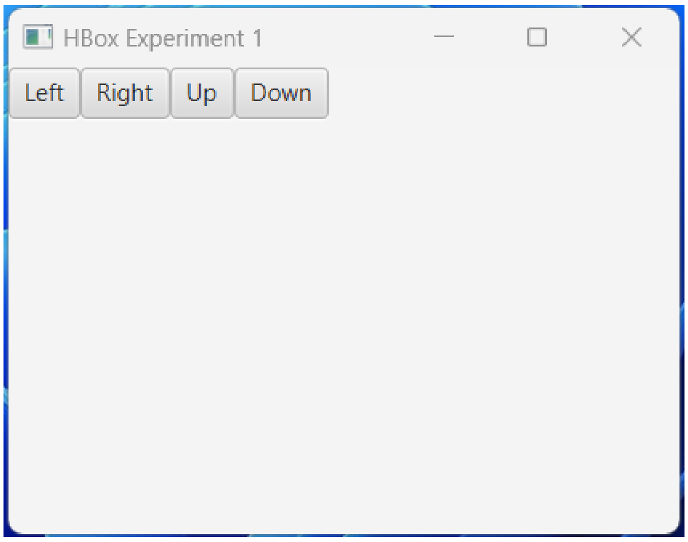
ToggleButton toggleButton1 = new ToggleButton("Left");

- This example creates a ToggleButton with the text Left on**.**

Example Program

```
package controls;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ToggleButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
public class ToggleButtonExperiments extends Application  {
   @Override
   public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("HBox Experiment 1");
        ToggleButton toggleButton1 = new ToggleButton("Left");
        ToggleButton toggleButton2 = new ToggleButton("Right");
        ToggleButton toggleButton3 = new ToggleButton("Up");
        ToggleButton toggleButton4 = new ToggleButton("Down");
        ToggleGroup toggleGroup = new ToggleGroup();
        toggleButton1.setToggleGroup(toggleGroup);
        toggleButton2.setToggleGroup(toggleGroup);
        toggleButton3.setToggleGroup(toggleGroup);
        toggleButton4.setToggleGroup(toggleGroup);

        HBox hbox = new HBox(toggleButton1, toggleButton2, toggleButton3, toggleButton4);
        Scene scene = new Scene(hbox, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] args) {
    launch(args);
    }
}
```

OUTPUT



5.3.3 RADIOBUTTONS

- The Radio Button is used to provide various options to the user. The user can only choose one option among all. A radio button is either selected or deselected. It can be used in a scenario of multiple-choice questions in the quiz where only one option needs to be chosen by the student.

Constructors of the RadioButton class:

1. RadioButton():Creates a radio button with an empty string for its label.
2. RadioButton(String t):Creates a radio button with the specified text as its label

Commonly used methods:

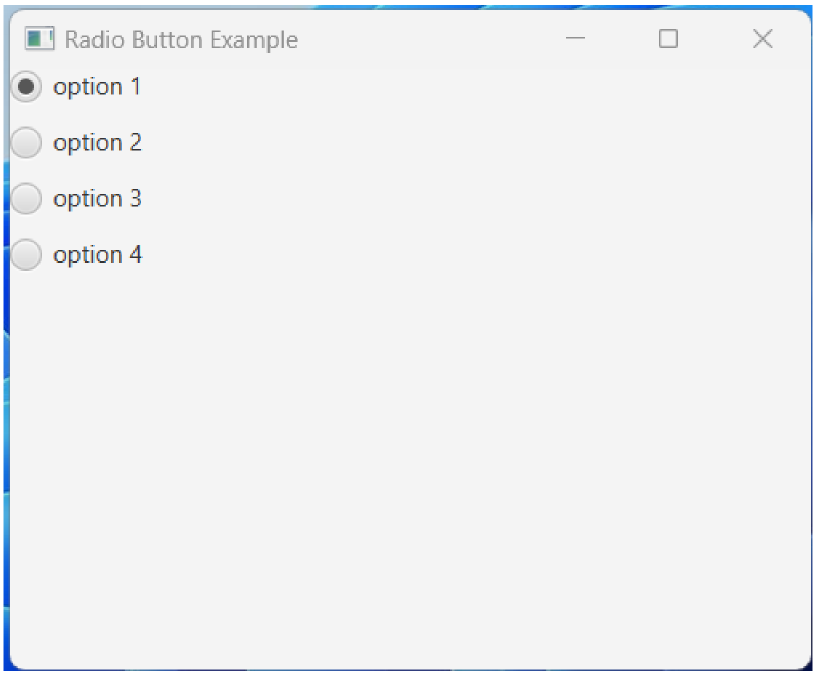
Method	Explanation
getText()	returns the textLabel for radio button
isSelected()	returns whether the radiobutton is selected or not
setSelected(boolean b)	sets whether the radiobutton is selected or not
setToggleGroup(ToggleGroup tg)	sets the toggle group for the radio button

Example Program

```
package CONTROLS;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.RadioButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
public class RADIOBUTTON extends Application
{
public static void main(String[] args)
{
launch(args);
}
@Override
public void start(Stage primaryStage) throws Exception {
   // TODO Auto-generated method stub
```

```
    ToggleGroup group = new ToggleGroup();
    RadioButton button1 = new RadioButton("option 1");
    RadioButton button2 = new RadioButton("option 2");
    RadioButton button3 = new RadioButton("option 3");
    RadioButton button4 = new RadioButton("option 4");
    button1.setToggleGroup(group);
    button2.setToggleGroup(group);
    button3.setToggleGroup(group);
    button4.setToggleGroup(group);
    VBox root=new VBox();
    root.setSpacing(10);
    root.getChildren().addAll(button1,button2,button3,button4);
    Scene scene=new Scene(root,400,300);
    primaryStage.setScene(scene);
    primaryStage.setTitle("Radio Button Example");
    primaryStage.show();
}
}
```

OUTPUT



5.3.4 LISTVIEW

- A list view is a scrollable list of items from which you can select desired items. You can create a list view component by instantiating the javafx.scene.control.ListView class. You can create either a vertical or a horizontal ListView.

Syntax:

Creating a ListView

```
ListView listView = new ListView();
```

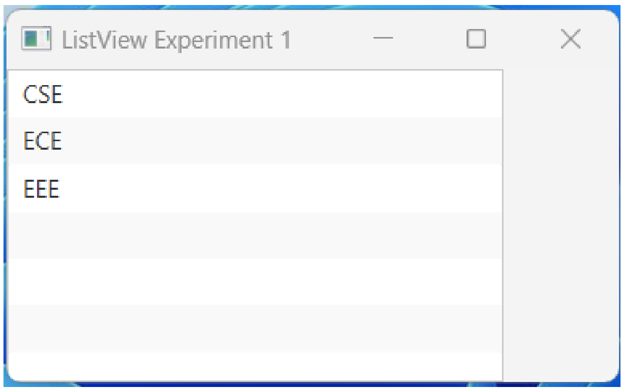
Adding Items to a ListView

```
listView.getItems().add("Item 1");
listView.getItems().add("Item 2");
listView.getItems().add("Item 3");
```

Example Program

```
package CONTROLS;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
public class LISTVIEW extends Application
{
 @Override
    public void start(Stage primaryStage) throws Exception
      {
        primaryStage.setTitle("ListView Experiment 1");
        ListView<String> listView = new ListView<String>();
        listView.getItems().add("Item 1");
        listView.getItems().add("Item 2");
        listView.getItems().add("Item 3");
        HBox hbox = new HBox(listView);
        Scene scene = new Scene(hbox, 300, 120);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] args)
   {
        launch(args);
    }
}
```

OUTPUT



CSE
ECE
EEE

5.3.5 COMBOBOX

- ComboBox is a part of the JavaFX library. JavaFX ComboBox is an implementation of simple ComboBox which shows a list of items out of which user can select at most one item, it inherits the class ComboBoxBase.

Constructors of ComboBox:

- ComboBox(): creates a default empty combo box
- ComboBox(ObservableList i): creates a combo box with the given items

Commonly used Methods:

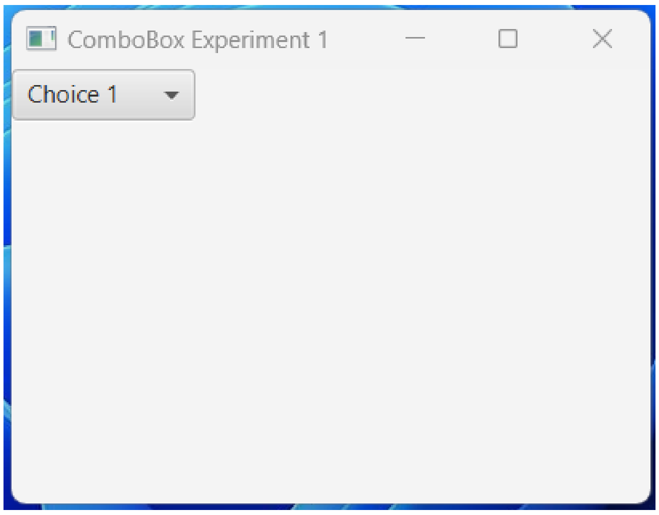
Method	Explanation
getEditor()	This method gets the value of the property editor
getItems()	This method returns the items of the combo box
getVisibleRowCount()	This method returns the value of the property visibleRowCount.

setItems(ObservableList v)	This method Sets the items of the combo box
setVisibleRowCount(int v)	This method sets the value of the property VisibleRowCount

Example Program

```
package CONTROLS;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
public class COMBOBOX extends Application
{
   @Override
   public void start(Stage primaryStage) throws Exception
     {
      primaryStage.setTitle("ComboBox Experiment 1");
      ComboBox<String> comboBox = new ComboBox<String>();
      comboBox.getItems().add("Choice 1");
      comboBox.getItems().add("Choice 2");
      comboBox.getItems().add("Choice 3");
      HBox hbox = new HBox(comboBox);
      Scene scene = new Scene(hbox, 200, 120);
      primaryStage.setScene(scene);
      primaryStage.show();
     }
   public static void main(String[] args) {
      launch(args);
   }
}
```

OUTPUT



5.3.6 CHOICEBOX

- ChoiceBox is a part of the JavaFX package. ChoiceBox shows a set of items and allows the user to select a single choice and it will show the currently selected item on the top. ChoiceBox by default has no selected item unless otherwise selected.

Constructor of the ChoiceBox class are:

- ChoiceBox(): Creates a new empty ChoiceBox.
- ChoiceBox(ObservableList items): Creates a new ChoiceBox with the given set of items.

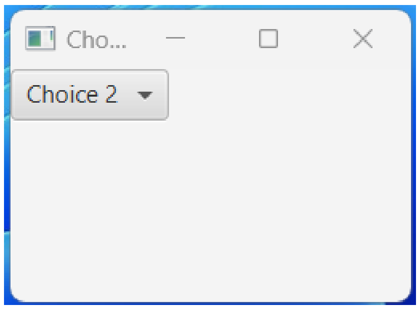
Commonly used methods:

Method	**Explanation**
getItems()	Gets the value of the property items.
getValue()	Gets the value of the property value.
hide()	Closes the list of choices.
setItems(ObservableList value)	Sets the value of the property items.
setValue(T value)	Sets the value of the property value.
show()	Opens the list of choices.

Example Program

```
package CONTROLS;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ChoiceBox;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
public class CHOICEBOX extends Application
{
   @Override
   public void start(Stage primaryStage) throws Exception {
      primaryStage.setTitle("ChoiceBox Experiment 1");
      ChoiceBox<String> choiceBox = new ChoiceBox<String>();
      choiceBox.getItems().add("Choice 1");
      choiceBox.getItems().add("Choice 2");
      choiceBox.getItems().add("Choice 3");
      HBox hbox = new HBox(choiceBox);
      Scene scene = new Scene(hbox, 200, 100);
      primaryStage.setScene(scene);
      primaryStage.show();
   }
   public static void main(String[] args) {
      launch(args);
   }
}
```

OUTPUT



5.3.7 TEXT CONTROLS

5.3.7.1 JavaFX | TextField

- TextField class is a part of JavaFX package. It is a component that allows the user to enter a line of unformatted text, it does not allow multi-line input it only allows the user to enter a single line of text. The text can then be used as per requirement.

Constructor of the TextField class:

- TextField(): creates a new TextField with empty text content
- TextField(String s): creates a new TextField with a initial text .

Commonly used methods:

Method	Explanation
setPrefColumnCount(int v)	Sets the value of the property prefColumnCount.
setOnAction(EventHandler value)	Sets the value of the property onAction.
setAlignment(Pos v)	Sets the value of the property alignment.
prefColumnCountProperty()	The preferred number of text columns
onActionProperty()	The action handler associated with this text field, or null if no action handler is assigned.
getPrefColumnCount()	Gets the value of the property prefColumnCount.
getOnAction()	Gets the value of the property onAction.
getAlignment()	Gets the value of the property alignment.
getCharacters()	Returns the character sequence backing the text field's content.

5.3.7.2 JavaFX |Password Field

- PasswordField class is a part of JavaFX package. It is a Text field that masks entered characters (the characters that are entered are not shown to the user). It allows the user to enter a single-line of unformatted text, hence it does not allow multi-line input.

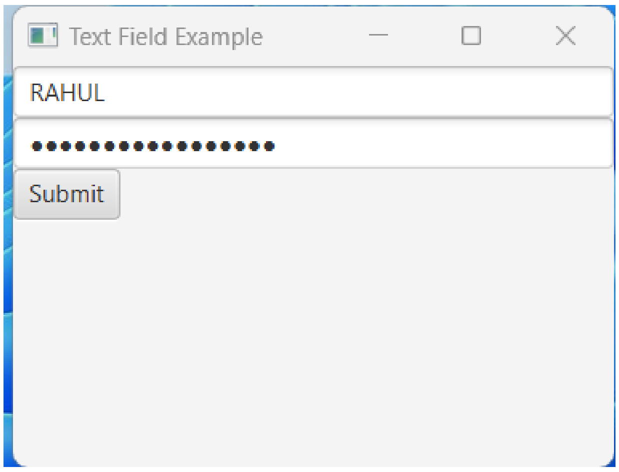
Constructor of the PasswordField class :

PasswordField(): creates a new PasswordField

Example Program for creating TextField and PasswordField

```
package CONTROLS;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.PasswordField;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
public class TEXTFIELD extends Application
{
public static void main(String[] args)
{
launch(args);
}
@Override
public void start(Stage primaryStage) throws Exception {
    // TODO Auto-generated method stub
    TextField tf1=new TextField();
    PasswordField pass=new PasswordField();
    Button b = new Button("Submit");
    VBox root=new VBox();
    root.getChildren().addAll(tf1,pass,b);
    Scene scene=new Scene(root,300,200);
    primaryStage.setScene(scene);
    primaryStage.setTitle("Text Field Example");
    primaryStage.show();
}
}
```

OUTPUT



5.3.7.3 Text Area in JavaFx

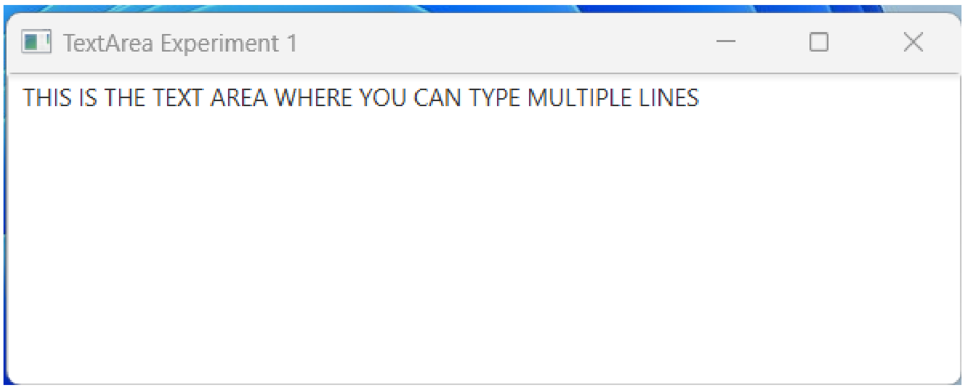
- A text area is a multi-line editor where you can enter text. Unlike previous versions, in the latest versions of JavaFX, a TextArea does not allow single lines in it. You can create a text area by instantiating the javafx.scene.control.TextArea class.

Example Program

```
package controls;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TextArea;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
public class TEXTAREA extends Application
{
   @Override
   public void start(Stage primaryStage) throws Exception
     {
       primaryStage.setTitle("TextArea Experiment 1");
       TextArea textArea = new TextArea();
       VBox vbox = new VBox(textArea);
       Scene scene = new Scene(vbox, 200, 100);
       primaryStage.setScene(scene);
       primaryStage.show();
   }
```

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

OUTPUT



THIS IS THE TEXT AREA WHERE YOU CAN TYPE MULTIPLE LINES

5.3.8 SCROLLPANE

- ScrollPane is a scrollable component used to display a large content in a limited space. It contains horizontal and vertical scroll bars.

Syntax for Creating a ScrollPane

ScrollPane scrollPane = new ScrollPane();

ScrollBar Policy

You can set up display policy for scroll bar:

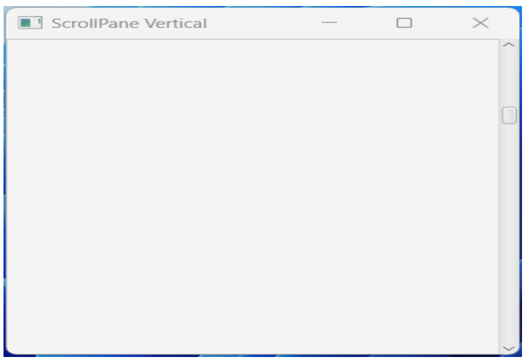
- NEVER - Never display
- ALWAYS - Always display
- AS_NEEDED - Display if needed.

Example Program

```
package CONTROLS;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.ScrollPane.ScrollBarPolicy;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
public class SCROLLPANE extends Application
{
```

```
@Override
public void start(Stage primaryStage)
{
// setting the title of application
primaryStage.setTitle("ScrollPane Vertical");
// Create a ScrollPane
ScrollPane scrollPane = new ScrollPane();
VBox vBox=new VBox();
// Setting the content to the ScrollPane
scrollPane.setContent(vBox);
// Always show vertical scroll bar for scrolling
scrollPane.setVbarPolicy(ScrollBarPolicy.ALWAYS);
scrollPane.setHbarPolicy(ScrollBarPolicy.NEVER);
//adding scroll pane to the scene
Scene scene = new Scene(scrollPane,200,300);
primaryStage.setScene(scene);
//showing the output
primaryStage.show();
}
public static void main(String[] args)
{
//invoking main method from JVM
launch(args);
}
}
```

OUTPUT



5.4 LAYOUTS

5.4.1 FlowPane

- FlowPane class is a part of JavaFX. Flowpane lays out its children in such a way that wraps at the flowpane's boundary.
- A horizontal flowpane (the default) will layout nodes in rows, wrapping at the flowpane's width.
- A vertical flowpane lays out nodes in columns, wrapping at the flowpane's height. FlowPane class inherits Pane class.

Constructors of the class:

- FlowPane(): Creates a new Horizontal FlowPane layout.
- FlowPane(double h, double v): Creates a new Horizontal FlowPane layout, with specified horizontal and vertical gap.
- FlowPane(double h, double v, Node… c): Creates a new Horizontal FlowPane layout, with specified horizontal, vertical gap and nodes.
- FlowPane(Node… c): Creates a FlowPane with specified childrens.
- FlowPane(Orientation o): Creates a FlowPane with specified orientation
- FlowPane(Orientation o, double h, double v): Creates a FlowPane with specified orientation and specified horizontal and vertical gap.
- FlowPane(Orientation o, double h, double v, Node… c): Creates a FlowPane with specified orientation and specified horizontal and vertical gap and specified childrens.
- FlowPane(Orientation o, Node… c): Creates a FlowPane with specified orientation and specified nodes.

Commonly Used Methods:

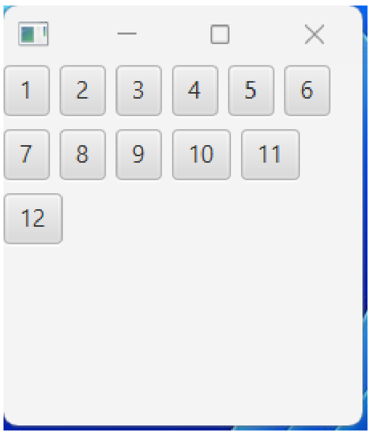
Method	Explanation
getAlignment()	Returns the value of Alignment of the pane.
getHgap()	Returns the horizontal gap of the flow pane.
getOrientation()	Returns the orientation of the pane.
getRowValignment()	Gets the value of the property rowValignment.
getVgap()	Returns the vertical gap of the flow pane.
setAlignment(Pos v)	Set the value of Alignment of the pane.
setHgap(double v)	Sets the horizontal gap of the flow pane.

setOrientation(Orientation o)	Set the orientation of the pane.
setRowValignment(double v)	Sets the value of the property rowValignment.
setVgap(double v)	Sets the vertical gap of the flow pane.
Method	Explanation

Example Program

```
package LAYOUTS;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;
  public class FLOWPANE extends Application
        {

          @Override
          public void start(Stage primaryStage) throws Exception
          {
             Button btn1 = new Button("1");
             Button btn2 = new Button("2");
             Button btn3 = new Button("3");
             Button btn4 = new Button("4");
             Button btn5 = new Button("5");
             Button btn6 = new Button("6");
             Button btn7 = new Button("7");
             Button btn8 = new Button("8");
             Button btn9 = new Button("9");
             Button btn10 = new Button("10");
             Button btn11 = new Button("11");
             Button btn12 = new Button("12");
             FlowPane root = new FlowPane();
             Scene scene = new Scene(root,100,100);
root.getChildren().addAll(btn1,btn2,btn3,btn4,btn5,btn6,btn7,btn8,btn9,btn10,btn11,btn12);
                root.setVgap(6);
                root.setHgap(5);
                primaryStage.setScene(scene);
                primaryStage.show();
            }
            public static void main(String[] args)
            {
                launch(args);
            }
        }
```

OUTPUT



5.4.2 HBox

- The JavaFX HBox component is a layout component which positions all its child nodes(components) in a horizontal row. It is represented by javafx.scene.layout.HBox class. We just need to instantiate HBox class in order to create HBox layout.

Constructors:

- new HBox() : create HBox layout with 0 spacing
- new Hbox(Double spacing) : create HBox layout with a spacing value

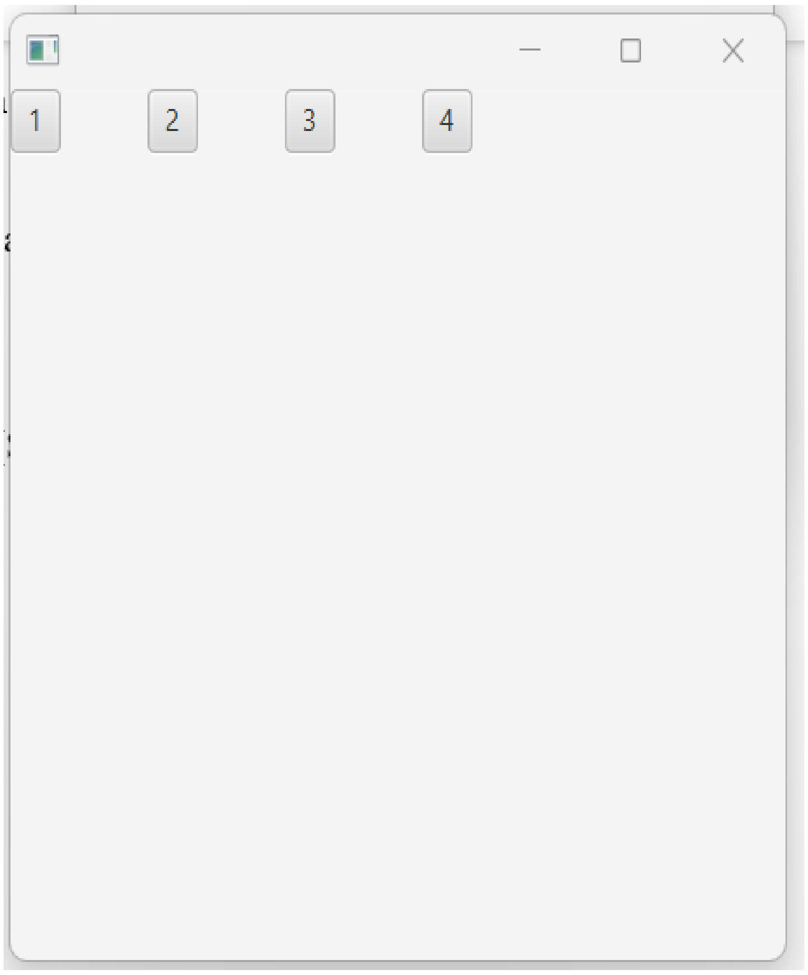
Methods

Property	Description	Setter Methods
alignment	This represents the alignment of the nodes.	setAlignment(Double)
fillHeight	This is a boolean property. If you set this property to true the height of the nodes will become equal to the height of the HBox.	setFillHeight(Double)
spacing	This represents the space between the nodes in the HBox. It is of double type.	setSpacing(Double)

Example Program

```
package LAYOUTS;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
public class HORIZONTAL extends Application {
        @Override
        public void start(Stage primaryStage) {
                try
                {
                        Button btn1 = new Button("1");
                        Button btn2 = new Button("2");
                        Button btn3 = new Button("3");
                        Button btn4 = new Button("4");
                        HBox root = new HBox();
                        Scene scene = new Scene(root,200,200);
                        root.getChildren().addAll(btn1,btn2,btn3,btn4);
                        root.setSpacing(40);
                        primaryStage.setScene(scene);
                        primaryStage.setHeight(800);
                        primaryStage.setWidth(500);
                        primaryStage.show();
                }
                catch(Exception e)
                {
                        e.printStackTrace();
                }
        }
        public static void main(String[] args) {
                launch(args);
        }
}
```

OUTPUT



5.4.3 VBox

- Instead of arranging the nodes in horizontal row, Vbox Layout Pane arranges the nodes in a single vertical column.
- It is represented by javafx.scene.layout.VBox class which provides all the methods to deal with the styling and the distance among the nodes. This class needs to be instantiated in order to implement VBox layout in our application.

Constructors

- VBox() : creates layout with 0 spacing
- Vbox(Double spacing) : creates layout with a spacing value of double type
- Vbox(Double spacing, Node? children) : creates a layout with the specified spacing among the specified child nodes
- Vbox(Node? children) : creates a layout with the specified nodes having 0 spacing among them

Methods

This Method Provides various properties which are described in the table below.

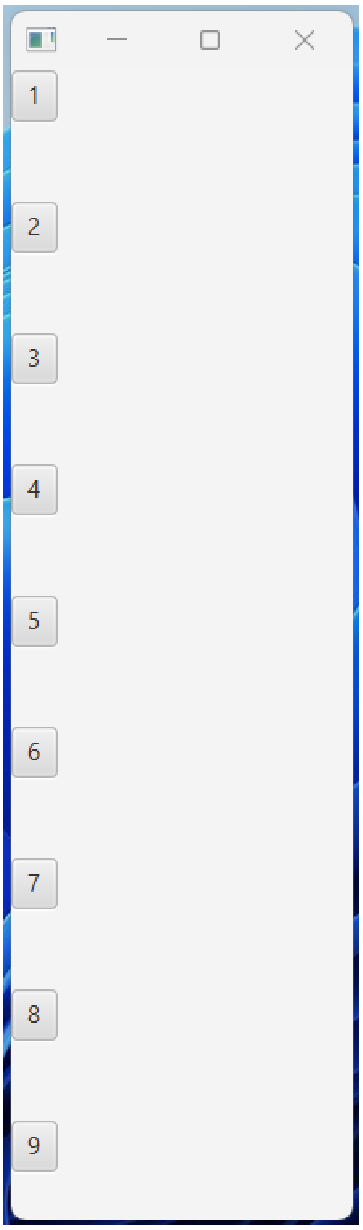
Property	Description	Setter Methods
Alignment	This property is for the alignment of the nodes.	setAlignement(Double)
FillWidth	This property is of the boolean type. The Widtht of resizeable nodes can be made equal to the Width of the VBox by setting this property to true.	setFillWidth(boolean)
Spacing	This property is to set some spacing among the nodes of VBox.	setSpacing(Double)

Example Program

```
package LAYOUTS;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
  public class VERTICAL extends Application
         {
           @Override
           public void start(Stage primaryStage) throws Exception
           {
              Button btn1 = new Button("1");
              Button btn2 = new Button("2");
              Button btn3 = new Button("3");
              Button btn4 = new Button("4");
              Button btn5 = new Button("5");
              Button btn6 = new Button("6");
              Button btn7 = new Button("7");
              Button btn8 = new Button("8");
              Button btn9 = new Button("9");
              VBox root = new VBox();
              root.setSpacing(20);
              Scene scene = new Scene(root);
              root.getChildren().addAll(btn1,btn2,btn3,btn4,btn5,btn6,btn7,btn8,btn9);
              root.setSpacing(40);
              primaryStage.setScene(scene);
              primaryStage.show();
           }
           public static void main(String[] args)
           {
              launch(args);
           }
```

```
        }
```

OUTPUT



5.4.4 BorderPane

- If we use the BorderPane, the nodes are arranged in the Top, Left, Right, Bottom and Center positions.
- The class named BorderPane of the package javafx.scene.layout represents the BorderPane.

This class contains five properties, which include

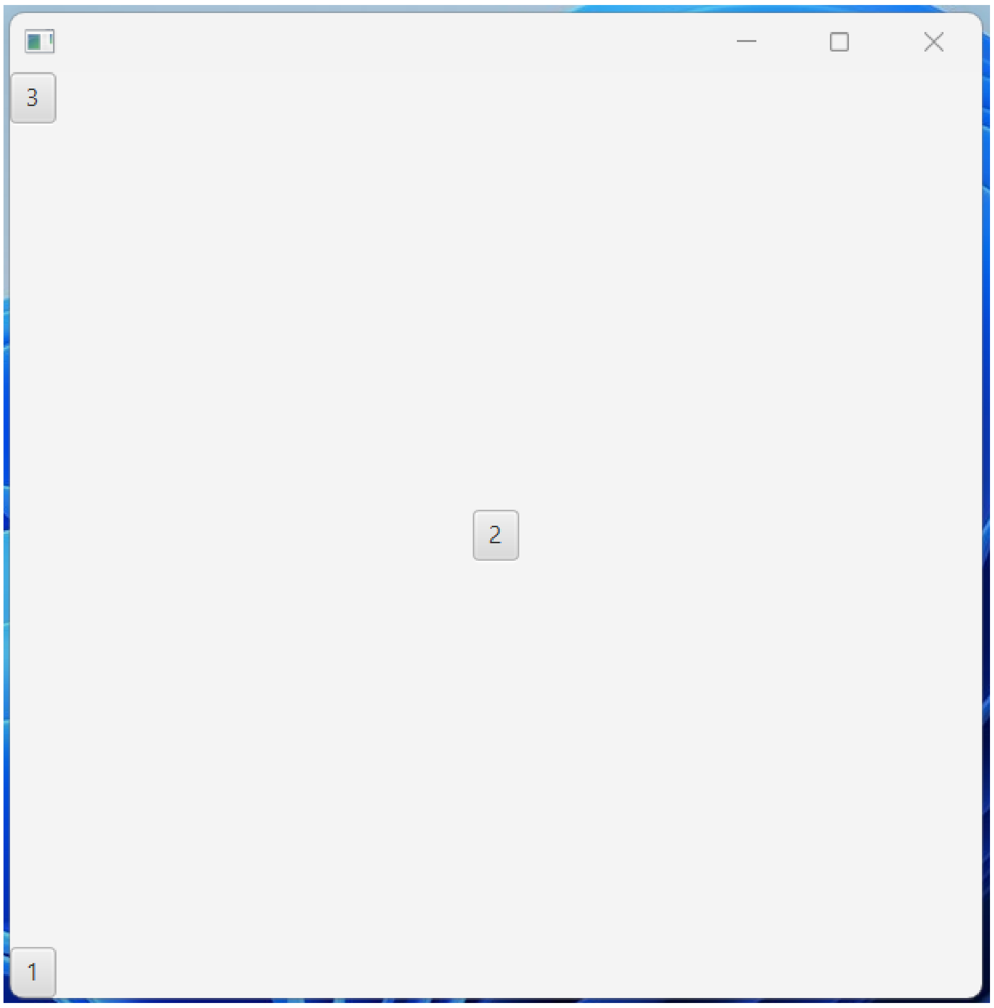
- **bottom** − This property is of **Node** type and it represents the node placed at the bottom of the BorderPane. You can set value to this property using the setter method **setBottom()**.
- **center** − This property is of **Node** type and it represents the node placed at the center of the BorderPane. You can set value to this property using the setter method **setCenter()**.
- **left** − This property is of **Node** type and it represents the node placed at the left of the BorderPane. You can set value to this property using the setter method **setLeft()**.
- **right** − This property is of **Node** type and it represents the node placed at the right of the BorderPane. You can set value to this property using the setter method **setRight()**.
- **top** − This property is of **Node** type and it represents the node placed at the top of the BorderPane. You can set value to this property using the setter method **setTop()**.

Example Program

```
package LAYOUTS;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;
  public class BORDERPANE extends Application
         {
           @Override
           public void start(Stage primaryStage) throws Exception
           {
              Button btn1 = new Button("1");
              Button btn2 = new Button("2");
              Button btn3 = new Button("3");
              BorderPane root = new BorderPane();
              root.setBottom(btn1);
              root.setCenter(btn2);
              root.setTop(btn3);
              Scene scene = new Scene(root);
```

```
            primaryStage.setScene(scene);
            primaryStage.setWidth(500);
            primaryStage.setHeight(500);
            primaryStage.show();
        }
        public static void main(String[] args)
        {
            launch(args);
        }
    }
```

OUTPUT



5.4.5 StackPane

- The StackPane layout pane places all the nodes into a single stack where every new node gets placed on the top of the previous node. It is represented by javafx.scene.layout.StackPane class. We just need to instantiate this class to implement StackPane layout into our application.

Properties

The class contains only one property that is given below along with its setter method.

Property	Description	Setter Method
alignment	It represents the default alignment of children within the StackPane's width and height	setAlignment(Node child, Pos value)

Constructors

The class contains two constructors that are given below.

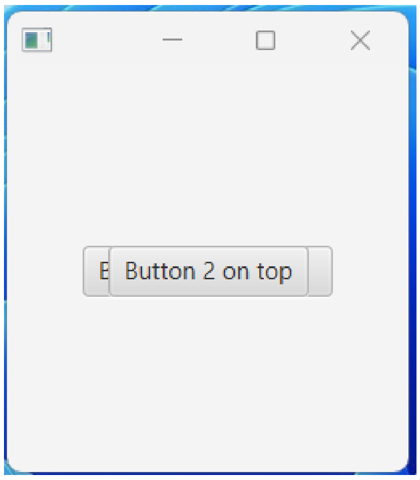
- StackPane()
- StackPane(Node? Children)

Example Program

```
package LAYOUTS;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
public class STACKPANE extends Application {
   @Override
   public void start(Stage primaryStage) throws Exception {
      Button btn1 = new Button("Button 1 on bottom ");
      Button btn2 = new Button("Button 2 on top");
      StackPane root = new StackPane();
      Scene scene = new Scene(root,200,200);
      root.getChildren().addAll(btn1,btn2);
```

```
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

OUTPUT



5.4.6 GridPane

- GridPane Layout pane allows us to add the multiple nodes in multiple rows and columns. It is seen as a flexible grid of rows and columns where nodes can be placed in any cell of the grid.
- It is represented by javafx.scence.layout.GridPane class. We just need to instantiate this class to implement GridPane.

Following are the cell positions in the grid pane of JavaFX −

(0, 0)	(1, 0)	(2, 0)
(2, 1)	(1, 1)	(0, 1)
(2, 2)	(1, 2)	(0, 2)

Properties

- The properties of the class along with their setter methods are given in the table below.

Property	Description	Setter Methods
alignment	Represents the alignment of the grid within the GridPane.	setAlignment(Pos value)
gridLinesVisible	This property is intended for debugging. Lines can be displayed to show the gridpane's rows and columns by setting this property to true.	setGridLinesVisible(Boolean value)
hgap	Horizontal gaps among the columns	setHgap(Double value)
vgap	Vertical gaps among the rows	setVgap(Double value)

Constructors

- The class contains only one constructor that is given below.

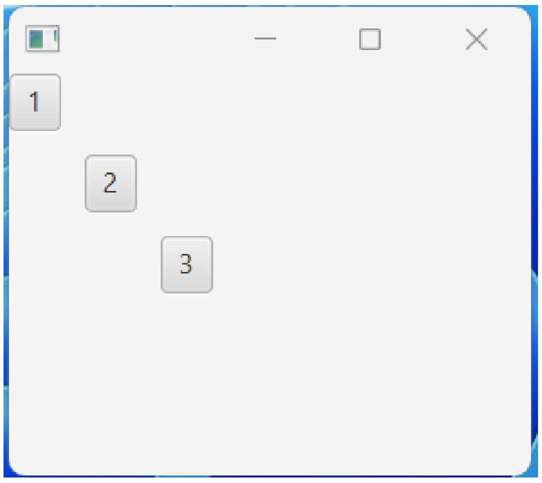
Public GridPane(): creates a gridpane with 0 hgap/vgap

Example Program

```
package LAYOUTS;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;
  public class GRIDPANE extends Application
         {
           @Override
           public void start(Stage primaryStage) throws Exception
           {
              Button btn1 = new Button("1");
```

```
            Button btn2 = new Button("2");
            Button btn3 = new Button("3");
            GridPane root = new GridPane();
            root.add(btn1, 0, 0);
            root.add(btn2, 1, 1);
            root.add(btn3, 2, 2);
            Scene scene = new Scene(root);
            root.setGridLinesVisible(false);
            root.setVgap(10);
            root.setHgap(10);
            primaryStage.setScene(scene);
            primaryStage.show();
        }
        public static void main(String[] args)
        {
            launch();
        }
    }
```

OUTPUT



5.5 MENUS, MENU ITEMS AND MENU BAR

- Menu is the main component of a any application. Menu is a popup menu that contains several menu items that are displayed when the user clicks a menu. The user can select a menu item after which the menu goes into a hidden state.
- JavaFX provides a Menu class to implement menus.
- In JavaFX, javafx.scene.control.Menu class provides all the methods to deal with menus. This class needs to be instantiated to create a Menu.
- MenuBar is usually placed at the top of the screen which contains several menus. JavaFX MenuBar is typically an implementation of a menu bar.

Constructor of the MenuBar class are:

- MenuBar(): creates a new empty menubar.
- MenuBar(Menu… m): creates a new menubar with the given set of menu.

Constructor of the Menu class are:

- Menu(): creates an empty menu
- Menu(String s): creates a menu with a string as its label
- Menu(String s, Node n):Constructs a Menu and sets the display text with the specified text and sets the graphic Node to the given node.
- Menu(String s, Node n, MenuItem… i):Constructs a Menu and sets the display text with the specified text, the graphic Node to the given node, and inserts the given items into the items list.

Commonly used methods:

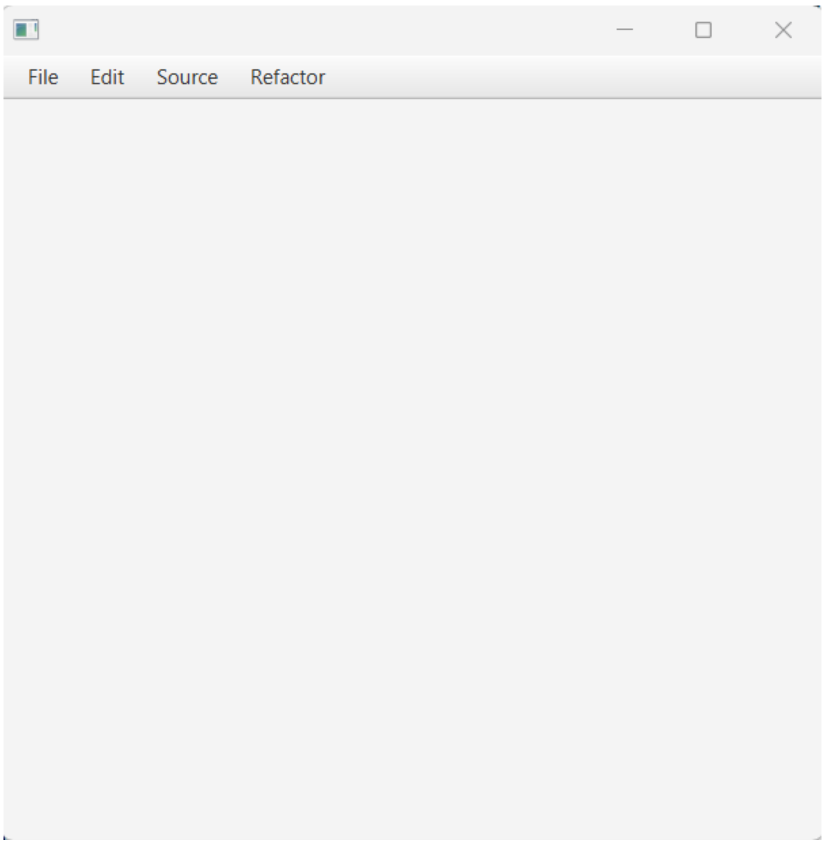
Method	**Explanation**
getItems()	returns the items of the menu
hide()	hide the menu
show()	show the menu
getMenus()	The menus to show within this MenuBar.
isUseSystemMenuBar()	Gets the value of the property useSystemMenuBar
setUseSystemMenuBar(boolean v)	Sets the value of the property useSystemMenuBar.
setOnHidden(EventHandler v)	Sets the value of the property onHidden.
setOnHiding(EventHandler v)	Sets the value of the property onHiding.
setOnShowing(EventHandler v)	Sets the value of the property onShowing.

Example Program for Creating Menus

```
package Menues;
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuBar;
import javafx.scene.layout.BorderPane;
public class Main extends Application {
    @Override
    public void start(Stage primaryStage) {
        try {
                //how to create menu in JavaFx
                //Let us now create a Menu bar
                MenuBar main_menu=new MenuBar();
                Menu File=new Menu("File");
                Menu Edit=new Menu("Edit");
                Menu Source=new Menu("Source");
                Menu Refactor=new Menu("Refactor");
                // Mapping all the menu objects to menu bar
                main_menu.getMenus().add(File);
                main_menu.getMenus().add(Edit);
                main_menu.getMenus().add(Source);
                main_menu.getMenus().add(Refactor);
               // Create a Layout and add the menu bar to the Layout
                BorderPane root=new BorderPane();
                root.setTop(main_menu);
                //we need to add this Layout to the Scene
                Scene sc=new Scene(root);
                primaryStage.setScene(sc);
                primaryStage.setWidth(500);
                primaryStage.setHeight(500);
                primaryStage.show();
```

```
            } catch(Exception e) {
                    e.printStackTrace();
            }
    }
    public static void main(String[] args) {
            launch(args);
    }
}
```

OUTPUT

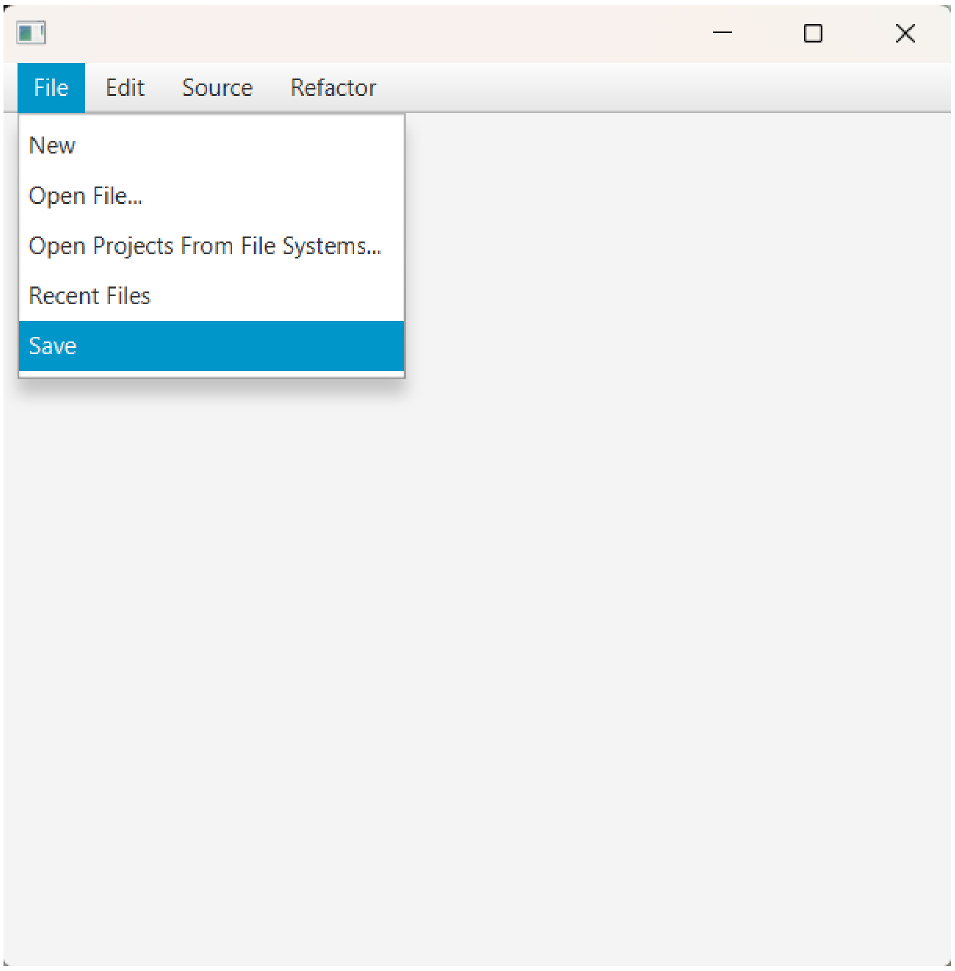


Example Program for Creating Menu Items

```
package Menues;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuBar;
import javafx.scene.control.MenuItem;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;
public class MENUITEMS extends Application
{
		@Override
		public void start(Stage primaryStage)
		{
		try
		{
		MenuBar main_menu=new MenuBar();
		Menu File=new Menu("File");
		Menu Edit=new Menu("Edit");
		Menu Source=new Menu("Source");
		Menu Refactor=new Menu("Refactor");
		// Mapping all the menu objects to menu bar
		main_menu.getMenus().add(File);
		main_menu.getMenus().add(Edit);
		main_menu.getMenus().add(Source);
		main_menu.getMenus().add(Refactor);
		//Let us add Menu Items for File Menu
		MenuItem New=new MenuItem("New");
		MenuItem OpenFile=new MenuItem("Open File...");
		MenuItem OpenProjects=new MenuItem("Open Projects From File
Systems...");
		MenuItem RecentFiles=new MenuItem("Recent Files");
		MenuItem Save=new MenuItem("Save");
```

```
                    //Map the Menu Items to the File Menu
                    File.getItems().add(New);
                    File.getItems().add(OpenFile);
                    File.getItems().add(OpenProjects);
                    File.getItems().add(RecentFiles);
                    File.getItems().add(Save);
                   // Create a Layout and add the menu bar to the Layout
                    BorderPane root=new BorderPane();
                    root.setTop(main_menu);
                    // we need to add this Layout to the Scene
                    Scene sc=new Scene(root);
                    primaryStage.setScene(sc);
                    primaryStage.setWidth(500);
                    primaryStage.setHeight(500);
                    primaryStage.show();
  }
 catch(Exception e)
   {
                     e.printStackTrace();
   }
}
public static void main(String[] args)
{
launch(args);
}
}
```

OUTPUT



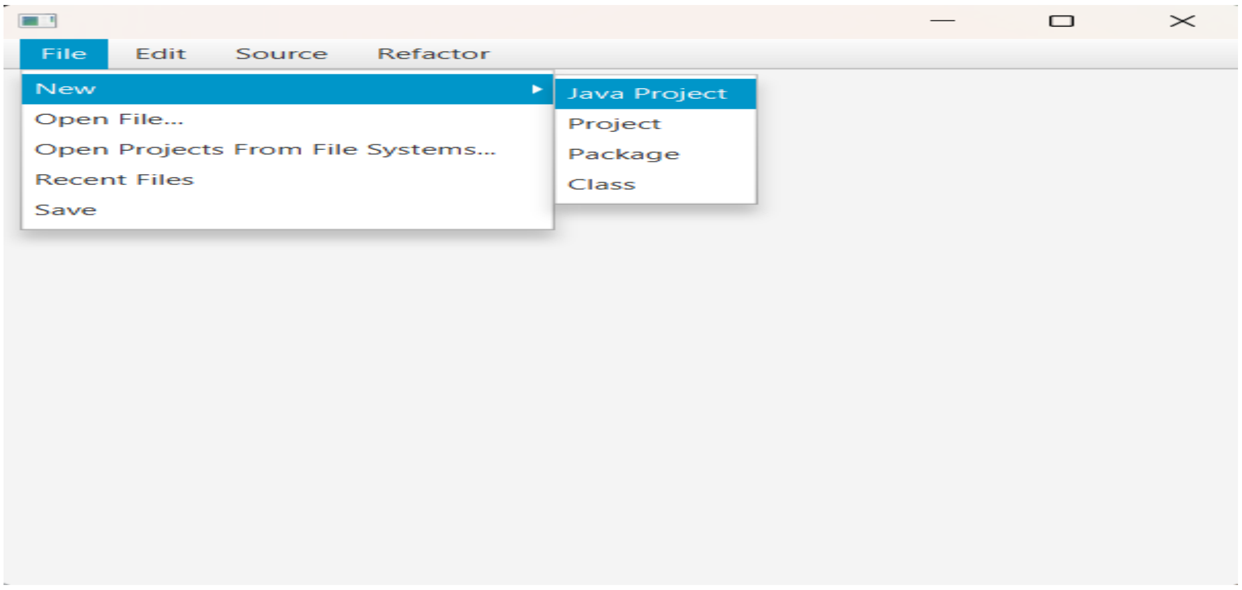
Example Program for Creating Sub Menus using JavaFx

```
package Menues;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuBar;
import javafx.scene.control.MenuItem;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;
public class SUBMENU extends Application
{
        @Override
        public void start(Stage primaryStage)
```

```
{
try
{
MenuBar main_menu=new MenuBar();
Menu File=new Menu("File");
Menu Edit=new Menu("Edit");
Menu Source=new Menu("Source");
Menu Refactor=new Menu("Refactor");
// Mapping all the menu objects to menu bar
main_menu.getMenus().add(File);
main_menu.getMenus().add(Edit);
main_menu.getMenus().add(Source);
main_menu.getMenus().add(Refactor);
//Let us add Menu Items for File Menu
Menu New=new Menu("New"); //New is not a menu item its a menu
MenuItem OpenFile=new MenuItem("Open File...");
MenuItem OpenProjects=new MenuItem("Open Projects From File Systems...");
MenuItem RecentFiles=new MenuItem("Recent Files");
MenuItem Save=new MenuItem("Save");
//We will create Menu Items for New Menu
MenuItem JavaProject=new MenuItem("Java Project");
MenuItem Project=new MenuItem("Project");
MenuItem Package1=new MenuItem("Package");
MenuItem Class1=new MenuItem("Class");
//Mapping Menu Items to Menu New
New.getItems().add(JavaProject);
New.getItems().add(Project);
New.getItems().add(Package1);
New.getItems().add(Class1);
//Map the Menu Items to the File Menu
File.getItems().add(New);
File.getItems().add(OpenFile);
File.getItems().add(OpenProjects);
```

```
            File.getItems().add(RecentFiles);
            File.getItems().add(Save);
            // Create a Layout and add the menu bar to the Layout
            BorderPane root=new BorderPane();
            root.setTop(main_menu);
            //we need to add this Layout to the Scene
            Scene sc=new Scene(root);
            primaryStage.setScene(sc);
            primaryStage.setWidth(500);
            primaryStage.setHeight(500);
            primaryStage.show();
   }
 catch(Exception e)
    {
            e.printStackTrace();
    }
}
public static void main(String[] args)
{
launch(args);
}
}
```

OUTPUT


File
Edit
Source
Refactor
New
Open File...
Open Projects From File Systems...
Recent Files
Save
Java Project
Project
Package
Class