Unit – 1: INTRODUCTION TO OOP AND JAVA

UNIT 1 **INTORDUCTION TO OOP AND JAVA**

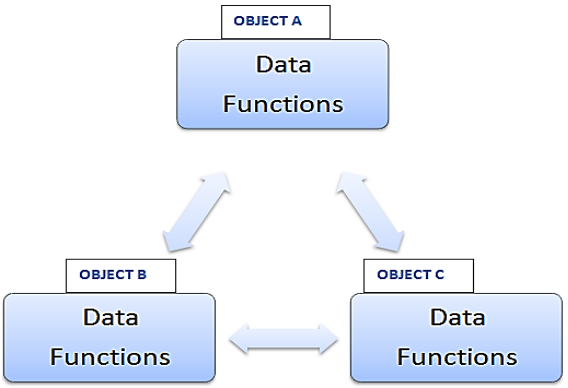
Overview of OOP – Object Oriented Programming Paradigms – Features of Object Oriented Programming – Java Buzzwords – Overview of Java – Data Types, Variables and Arrays – Operators – Control Statements – Programming Structures in Java – Defining Classes in Java – Constructors – Methods – Access Specifiers – Static Members – JavaDoc Comments.

1.1: Overview of OOP

OBJECT ORIENTED PROGRAMMING (OOP):

Object-Oriented Programming System (OOPs) is a programming paradigm based onthe concept of —objects that contain data and methods, instead of just functions and procedures.

- ✓ The primary **purpose** of object-oriented programming is to increase the flexibility and maintainability of programs.
- ✓ Object oriented programming brings together data and its behavior (methods) into a single entity (object) which makes it easier to understand how a program works.



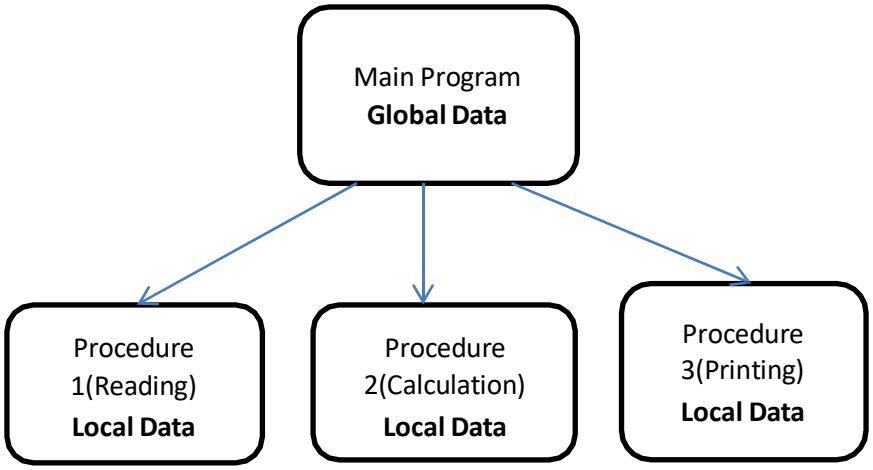
- **Features / advantages of Object Oriented Programming :-**
  1. It emphasis in own data rather than procedure.
  2. It is based on the principles of inheritance, polymorphism, encapsulation and dataabstraction.
  3. Programs are divided into objects.
  4. Data and the functions are wrapped into a single unit called class so that data ishidden and is safe from accidental alternation.
  5. Objects communicate with each other through functions.
  6. New data and functions can be easily added whenever necessary.
  7. Employs bottom-up approach in program design.

PROCEDURE-ORIENTED PROGRAMMING [POP]:

Procedure-Oriented Programming is a conventional programming which consists of writing a list of instructions for the computer to follow and organizing these instructions into groups known as Functions (or) Procedures (or) subroutines (or)Modules.

Example: A program may involve the following operations:

- ✓ Collecting data from user (Reading)
- ✓ Calculations on collected data (Calculation)
- ✓ Displaying the result to the user (Printing)



Characteristics of Procedural oriented programming:-

1. It focuses on process rather than data.
2. It takes a problem as a sequence of things to be done such as reading, calculating and printing. Hence, a number of functions are written to solve a problem.
3. A program is divided into a number of functions and each function has clearly defined purpose.
4. Most of the functions share global data.
5. Data moves openly around the system from function to function.
6. Employs top-down approach in program design.

Drawback of POP

- ➢ Procedural languages are difficult to relate with the real world objects.
- ➢ Procedural codes are very difficult to maintain, if the code grows larger.
- ➢ Procedural languages do not have automatic memory management as like in Java. Hence, it makes the programmer to concern more about the memory management of the program.
- ➢ The data, which is used in procedural languages, are exposed to the

program.So, there is no security for the data.

- Examples of Procedural languages :
  - BASIC
  - C
  - Pascal
  - FORTRAN

- **Difference between POP and OOP:**

	Procedure Oriented Programming	Object Oriented Programming
Divided Into	In POP, program is divided into smallparts called **functions**.	In OOP, program is divided into partscalled **objects**.
Importance	In POP, Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any accessspecifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other throughmember functions.
Expansion	To add new data and function in POPis not so easy.	OOP provides an easy way to add newdata and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper wayfor hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Examples of POP are: C,VB, FORTRAN, and Pascal.	Examples of OOP are: C++, JAVA, VB.NET, C#.NET.

1.2 : FEATURES / CHARACHTERISTICS OF OBJECT ORIENTED PROGRAMMING CONCEPTS

OOPs simplify the software development and maintenance by providing some concepts:

1. Class - Blue print of Object
2. Object - Instance of class
3. Encapsulation - Protecting our data
4. Polymorphism - Different behaviors at different instances
5. Abstraction - Hiding irrelevant data
6. Inheritance - An object acquiring the property of another object

1. Class:

A class is a collection of similar objects and it contains data and methods that operate on that data. In other words – **Class is a blueprint or template for a set of objects that share a common structure and a common behavior**. It is a logical entity.

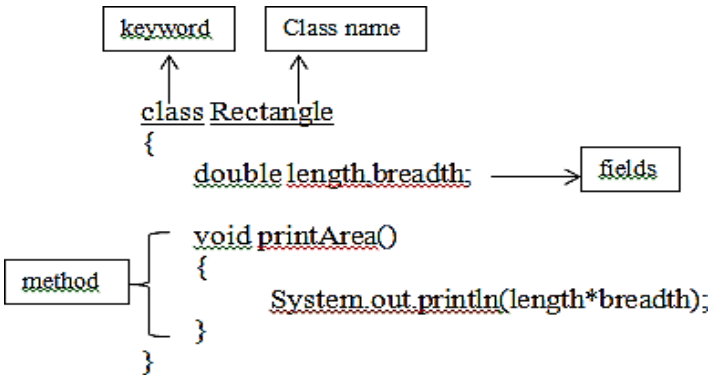
A class in Java can contain:

- **fields**
- **methods**
- **constructors**
- **blocks**
- **nested class and interface**

Syntax to declare a class:

Example:

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



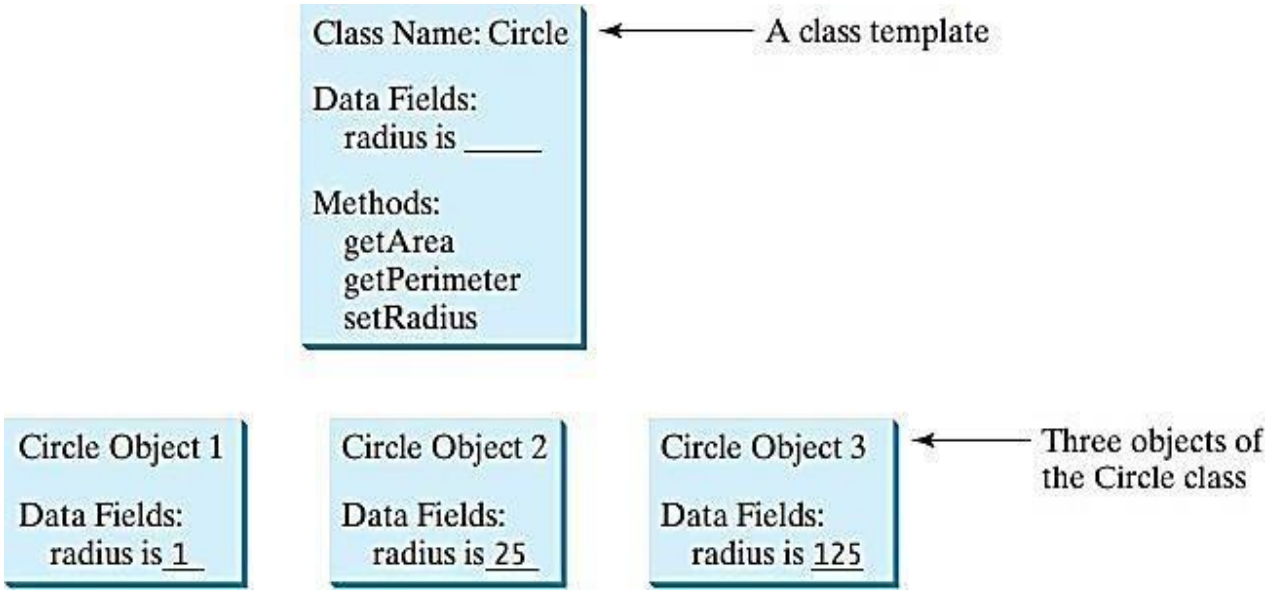
2. Object:

Any entity that has state and behavior is known as an object. **Object is an instance of a class.**

- ✓ For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.
- ✓ The object of a class can be created by using the **new** keyword in Java Programminglanguage.

```
class_name object_name = new class_name;
                (or)
class_name object_name;
object_name = new class_name();
```

Syntax to create Object in Java:



An object has three characteristics:

- **State:** represents data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **Identity:** Object identity is an unique ID used internally by the JVM to identify each object uniquely.
- For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.

Difference between Object and Class

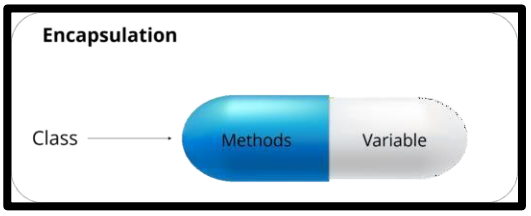
S.No.	Object	Class
1)	Object is an **instance** of a class.	Class is a **blueprint or template** from whichobjects are created.
2)	Object is a **real world entity** such aspen, laptop, mobile, bed, keyboard, mouse, chair etc.	Class is a **group of similar objects**.
3)	Object is a **physical** entity.	Class is a **logical** entity.
4)	Object is created through **new keyword** mainly e.g. Student s1=new Student();	Class is declared using **class keyword** e.g.class Student{}
5)	Object is created **many times** as perrequirement.	Class is declared **once**.

6)	Object **allocates memory when it iscreated**.	Class **doesn't allocated memory when it iscreated**.
7)	There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization.	There is only **one way to define class** in java using class keyword.

3. Encapsulation:

Wrapping of data and method together into a single unit is known as Encapsulation.

For example: capsule, it is wrapped with different medicines.



- ✓ **In OOP, data and methods operating on that data are combined together to form a single unit, this is referred to as a Class.**
- ✓ Encapsulation is the mechanism that binds together code and the data it manipulates and keeps both safe from outside interference and misuse.
- ✓ **The insulation of the data from direct access by the program is called —data hiding.** Since the data stored in an object cannot be accessed directly, the data is safe i.e.,the data is unknown to other methods and objects.

4. Polymorphism:

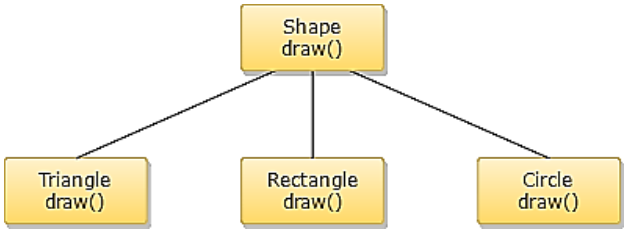
- ✓ Polymorphism is a concept by which we can perform a single action by different ways. It is the ability of an object to take more than one form.
- ✓ The word "poly" means many and "morphs" means forms. So polymorphism meansmany forms.
- ✓ An operation may exhibit different behaviors in different instances. The behavior depends on the data types used in the operation.
- ✓ For Example:- Suppose if you are in a classroom that time you behave like a student, when you are in the market at that time you behave like a customer, when you at your home at that time you behave like a son or daughter, Here one person present in different-different behaviors.

- Two types of polymorphism:

1. **Compile time polymorphism / Method Overloading: -** In this method, object is bound to the function call at the compile time itself.
2. **Runtime polymorphism / Method Overriding: -** In this method, object is bound to the function call only at the run time.

- **<u>In java, we use method overloading and method overriding to achievepolymorphism.</u>**
- **Example:**

1. draw(int x, int y, int z)
2. draw(int l, int b)
3. draw(int r)



5. <u>Abstraction:</u>

- ✓ Abstraction refers to the act of representing essential features without including the background details or explanations**. i.e., *Abstraction* means hiding lower-level details and exposing only the essential and relevant details to the users.**
- ✓ For Example: - Consider an ATM Machine; All are performing operations on the ATM machine like cash withdrawal, money transfer, retrieve mini-statement…etc. but we can't know internal details about ATM.
- ✓ Abstraction provides advantage of code reuse.
- ✓ Abstraction enables program open for extension.
- ✓ **In java, abstract classes and interfaces are used to achieve Abstraction.**

6. <u>Inheritance:</u>

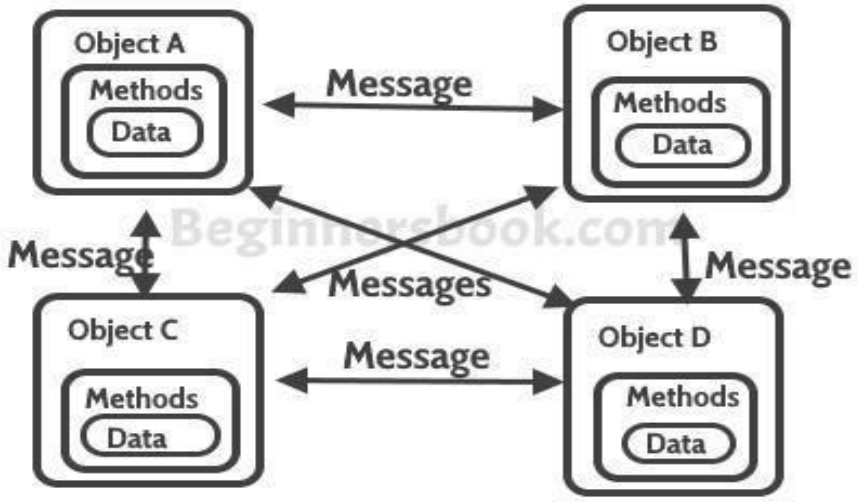
- ✓ **Inheritance in java** is a mechanism in which one object acquires all the propertiesand behaviors of another object.
- ✓ The idea behind inheritance in java is that we can create new classes that are built upon existing classes. When we inherit from an existing class, we can reuse methods and fields of parent class, and we can add new methods and fields also.
- ✓ Inheritance represents the **IS-A relationship**, also known **as *parent-child* relationship**.
- ✓ For example:- In a child and parent relationship, all the properties of a father areinherited by his son.
- ✓ **Syntax of Java Inheritance**

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

7. **Message Passing:**

Message Communication:

- ✓ Objects interact and communicate with each other by sending ***messages*** to eachother. This information is passed along with the message as parameters.



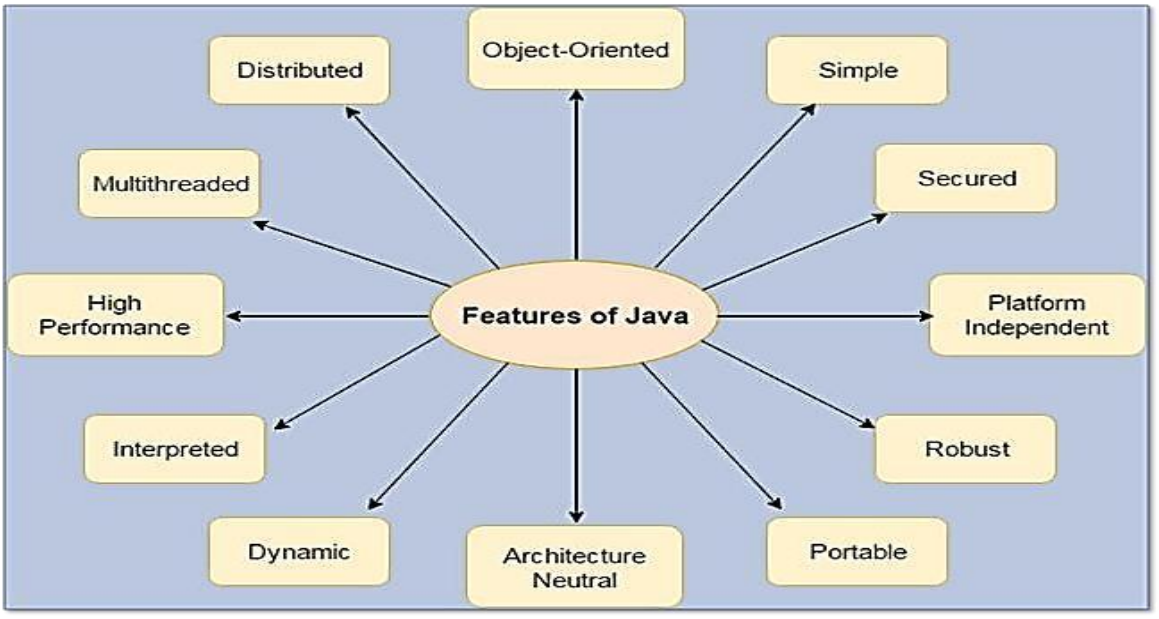
- ✓ A message for an object is a request for execution of a procedure and therefore will invoke a method (procedure) in the receiving object that generates the desired result.
- ✓ Message passing involves specifying the name of the object, the name of themethod (message) and the information to be sent.
- ✓ **Example:**

Employee.getName(name);
Where,
Employee – object name
getName – method name (message)
name - information

1.3 Java Buzzwords

The following are the features of the Java language:

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



1. Object Oriented:

- ✓ Java programming is pure object-oriented programming language. Like C++, Javaprovides most of the object oriented features.
- ✓ Though C++ is also an object oriented language, we can write programs in C++ without aclass but it is not possible to write a Java program without classes.
- ✓ Example: **Printing "Hello" Message.**

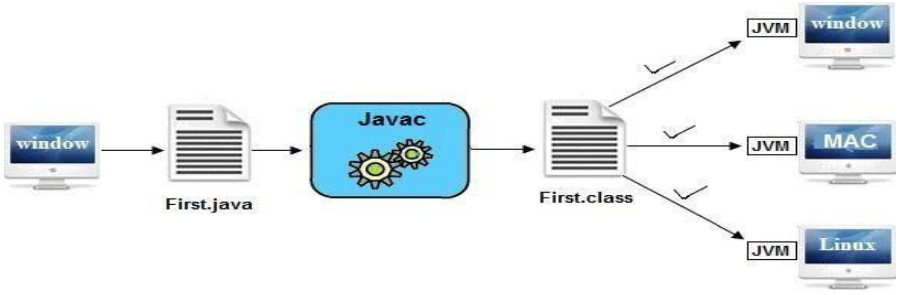
C++ (can be without class)	Java – No programs without classes and objects

With Class:
#include<iostream.h>
class display {
public:
void disp()
{
cout<<"Hello!";
}
};
main()
{
display d;
d.disp();
}
 	
Without class: #include<iostream.h>
void main()
{
clrscr();
cout<<"\n Hello!";
getch();
}
	With class:
	import java.io.*;
	class Hello {
	public static void main(String args[])
	{
	System.out.println("Hello!");
	}
	}
	
	Without class is not possible

2. **Simple:**
   - ✓ Java is Easy to write and more readable and eye catching.
   - ✓ Most of the concepts are drew from C++ thus making Java learning simpler.

3. **Secure :**
   - ✓ Since Java is intended to be used in networked/distributed environments, lot of emphasishas been placed on security.
   - ✓ Java provides a secure means of creating Internet applications and to access web applications.
   - ✓ Java enables the construction of secured, virus-free, tamper-free system.

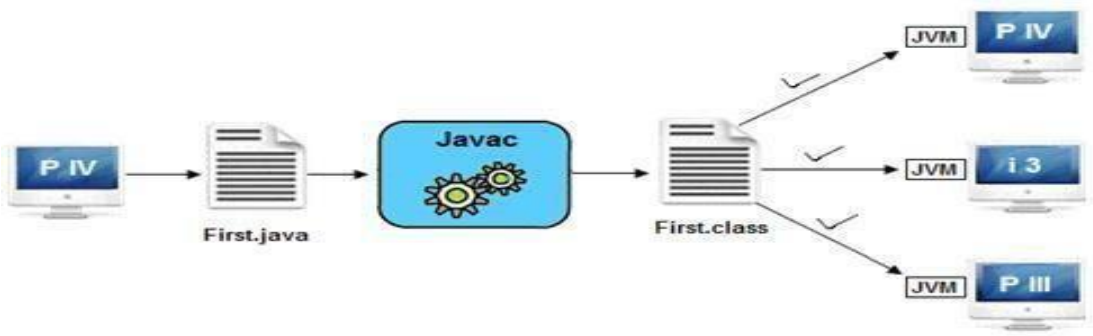
4. **Platform Independent:**
   - ✓ Unlike C, C++, when Java program is compiled, it is not compiled into platform-specific machine code, rather it is converted into platform independent code called **bytecode**.
   - ✓ The Java bytecodes are not specific to any processor. They can be executed in anycomputer without any error.
   - ✓ Because of the bytecode, Java is called as **Platform Independent.**



5. **Robust:**
   - ✓ Java encourages error-free programming by being strictly typed and performing run-time checks.

6. **Portable:**
   - ✓ Java bytecode can be distributed over the web and interpreted by **Java Virtual Machine (JVM)**
   - ✓ Java programs can run on any platform (Linux, Window, Mac)
   - ✓ Java programs can be transferred over world wide web (e.g applets)

7. **Architecture Neutral:**
   - ✓ Java is not tied to a specific machine or operating system architecture.
   - ✓ Machine Independent i.e Java is independent of hardware.
   - ✓ Bytecode instructions are designed to be both easy to interpret on any machine and easilytranslated into native machine code.



8. **Dynamic and Extensible:**
   - ✓ Java is a more dynamic language than C or C++. It was developed to adapt to an evolvingenvironment.
   - ✓ Java programs carry with them substantial amounts of run-time information that are usedto verify and resolve accesses to objects at run time.
9. **Interpreted:**
   - ✓ Java supports cross-platform code through the use of Java bytecode.
   - ✓ The Java interpreter can execute Java Bytecodes directly on any machine to which theinterpreter has been ported.

10. **High Performance:**
    - ✓ Bytecodes are highly optimized.
    - ✓ JVM can execute the bytecodes much faster.
    - ✓ With the use of Just-In-Time (JIT) compiler, it enables high performance.

11. **Multithreaded:**
    - ✓ Java provides integrated support for multithreaded programming.
    - ✓ Using multithreading capability, we can write programs that can do many tasks simultaneously.
    - ✓ The benefits of multithreading are better responsiveness and real-time behavior.

12. **Distributed:**
    - ✓ Java is designed for the distributed environment for the Internet because it handles TCP/IP protocols.
    - ✓ Java programs can be transmit and run over internet.

1.4: Overview of Java

- Java programming language was originally developed by Sun Microsystems which was initiated by **James Gosling** and released in **1995** as core component of Sun Microsystems' Java platform **(Java 1.0 [J2SE]).**

Java is a high-level object-oriented programming language, which provides developers with the means to create powerful applications, which are very small in size, platform independent, secure and robust.

- Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions ofUNIX.
- Java is mainly used for Internet Programming.
- Java is related to the languages C and C++. From C, Java inherits its syntax and from C++,Java inherits its OOP concepts.
- Ancestors of Java: - C, C++, B, BCPL.

Five primary goals in the creation of the Java language:

1. It should use the object-oriented programming methodology.
2. It should allow the same program to be executed on multiple operating systems.
3. It should contain built-in support for using computer networks.
4. It should be designed to execute code from remote sources securely.
5. It should be easy to use.

1.4.1 : BASIC JAVA TERMINALOGIES:

1. **BYTECODE:**
   Byte code is an intermediate code generated from the source code by java compiler and it is platform independent.

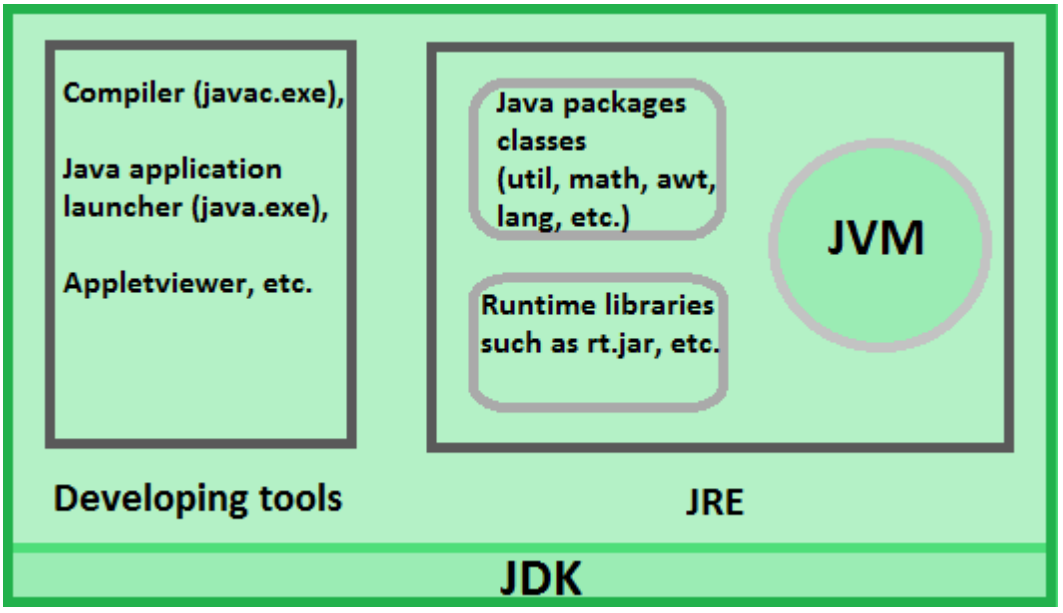
2. **JAVA DEVELOPMENT KIT (JDK):**

- The Java Development Kit (JDK) is a software development environment usedfor developing Java applications and applets.
- It includes the Java Runtime Environment (JRE), an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (javadoc) and other tools needed in Java development.

3. **JAVA RUNTIME ENVIRONMENT (JRE):**
   JRE is used to provide runtime environment for JVM. It contains set of libraries +other files that JVM uses at runtime.

4. **JAVA VIRTUAL MACHINE (JVM):**

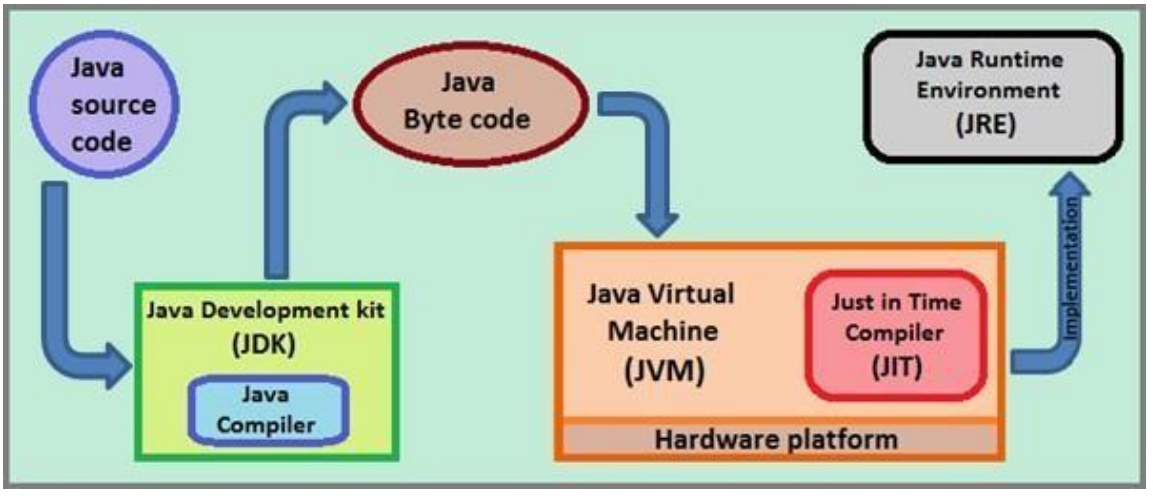
- JVM is an interpreter that converts a program in Java bytecode (intermediate language) into native machine code and executes it.
- JVM needs to be implemented for each platform because it will differ from platform to platform.



- The JVM performs following main tasks:
  - Loads code
  - Verifies code
  - Executes code
  - Provides runtime environment

5. <u>JIT (JUST IN TIME) COMPILER:</u>

It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time neededfor compilation.



<u>Types of Java program:</u>

In Java, there are two types of programs namely,

1. Application Program
2. Applet Program

1. Application Programs

Application programs are stand-alone programs that are written to carry out certain tasks on local computer such as solving equations, reading and writing files etc. The application programs can be executed using two steps:

1. Compile source code to generate Byte code using javac compiler.

2. Execute the byte code program using Java interpreter.

2. Applet programs:

Applets are small Java programs developed for Internet applications. An applet located in distant computer can be downloaded via Internet and executed on a local computer using Java capable browser. The Java applets can also be executed in the command line using appletviewer, which is part of the JDK.

1.4.2 : JAVA SOURCE FILE - STRUCTURE – COMPILATION

THE JAVA SOURCE FILE:

A Java source file is a plain text file containing Java source code and having .java extension. The .java extension means that the file is the Java source file. Java source code file contains source code for a class, interface, enumeration, or annotation type. There are some rules associated to Java source file.

Java Program Structure:

Java program may contain many classes of which only one class defines the main method.

A Java program may contain one or more sections.

Documentation Section
Package Statement
Import Statements
Interface Statements
Class Definitions

main Method Calss
{
Main Method Definition
}

Of the above Sections shown in the figure, the Main Method class is Essential part, Documentation Section is a suggested part and all the other parts are optional.

Documentation Section

- ✓ It Comprises a Set of comment lines giving the name of the program, the authorand other details.

- ✓ Comments help in Maintaining the Program.
- ✓ Java uses a Style of comment called *documentation comment.*
  /* * …… */
- ✓ This type of comment helps is generating the documentation automatically.
- ✓ **Example:**

```
/*
 * Title: Conversion of Degrees
 * Aim: To convert Celsius to Fahrenheit and vice versa
 * Date: 31/08/2000
 * Author: tim
*/
```

Package Statement

- ✓ The first statement allowed in a Java file is a package statement.
- ✓ It declares the package name and informs the compiler that the classes defined belong to this package.
- ✓ **Example :**

  package student;

  package basepackage.subpackage.class;
- ✓ It is an optional declaration.

Import Statements

- ✓ The statement instructs the interpreter to load a class contained in a particular package.
- ✓ Example :

  import student.test;

  Where, student is the package and test is the class.

Interface Statements

- ✓ An interface is similar to classes which consist of group of method declaration.
- ✓ Like classes, interfaces contain methods and variable.
- ✓ To link the interface to our program, the keyword **implements** is used.
- ✓ **Example:**

  public class xx extends Applet implements ActionListener

  where, xx – class name (subclass of Applet)Applet – Base class name
  ActionListener – interface Extends & implements - keywords
- ✓ It is used when we want to implement the feature of Multiple Inheritance in Java
- ✓ It is an optional declaration.

Class Definitions

- ✓ A Java Program can have any number of class declarations.

- ✓ The number of classes depends on the complexity of the program.

Main Method Class

- ✓ Every Java Standalone program requires a main method as its starting point.
- ✓ A Simple Java Program will contain only the main method class.
- ✓ It creates objects of various classes and uses those objects for performing various operations.
- ✓ When the end of main is reached the program terminates and the control transferred back to the Operating system.
- ✓ **Syntax for writing main:**
  public static void main(String arg[])
  where,
  public – It is an access specifier to control the visibility of class members. main() must be declared as public, since it must be called by code outside of its class when the program is started.
  static – this keyword allows main() method to be called without having to instantiate the instance of the class.
  void – this keyword tells the compiler that main() does not return any value.
  main() – is the method called when a Java application begins.
  String arg[] – arg is an string array which receives any command-line arguments present when the program is executed.

<u>Rules to be followed to write Java Programs:</u>

About Java programs, it is very important to keep in mind the following points.

- ***Case Sensitivity - Java is case sensitive, which means identifier Hello and hellowould have different meaning in Java.***
- ***Class Names - For all class names the first letter should be in Upper Case.***
  If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.
  Example class MyFirstJavaClass
- ***Method Names - All method names should start with a Lower Case letter.***
  If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.
  Example public void myMethodName()
- ***Program File Name - Name of the program file should exactly match the classname.***
  When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and theclass name do not match your program will not compile).
  Example : Assume 'MyFirstJavaProgram' is the class name. Then the file shouldbe saved as

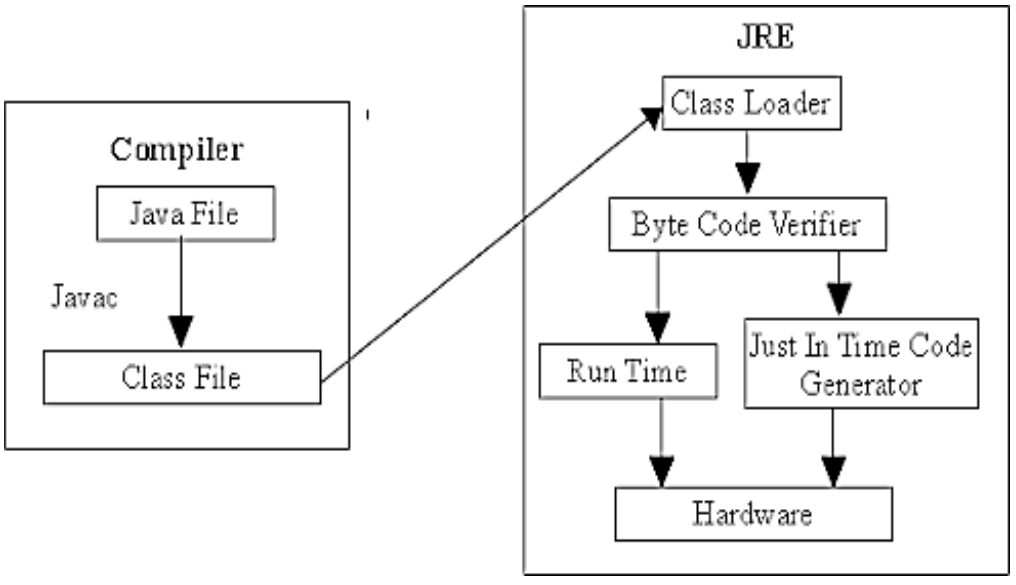
'MyFirstJavaProgram.java'

- ***public static void main(String args[]) - Java program processing starts from the main() method which is a mandatory part of every Java program.***

Compiling and running a java program in command promptSTEPS:

1. Set the path of the compiler as follows (type this in command prompt):
   Set path="C:\Program Files\Java\jdk1.6.0_20\bin";
2. To create a Java program, ensure that the name of the class in the file is the sameas the name of the file.
3. Save the file with the extension .java (Example: HelloWorld.java)
4. To compile the java program use the command javac as follows:
   javac HelloWorld.java
   This will take the source code in the file HelloWorld.java and create the javabytecode in a file HelloWorld.class
5. To run the compiled program use the command java as follows:
   java HelloWorld
   (Note that you do not use any file extension in this command.)

   At compile time, java file is compiled by Java Compiler (It does not interact with OS) andconverts the java code into bytecode.



Class Loader : is the subsystem of JVM that is used to load class files.
Bytecode Verifier : checks the code fragments for illegal code that can violate access right to objects
Interpreter : read bytecode stream then execute the instructions.

Example 1: A First Java Program:

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

Save: HelloWorld.java **Compile:** javac HelloWorld.java**Run:** java HelloWorld
Output: Hello World

Program Explanation:
public is the access specifier, **class** is a keyword and **HelloWorld** is the class name. **{** indicates the start of program block and **}** indicates the end of the program block. **System.out.println()** – is the output statement to print some message on the screen. Here, **System** is a predefined class that provides access to the system, **out** is the output stream that is connected to the console and **println()** is method to display the given string.

Example 2: A Second Java Program:

```
import java,util.Scanner; // Scanner is a class which contains necessary methods
                             to provide a user an access to the i/p console.
public class Example2                    // class declaration
{                                        // class definition starts
public static void main(String args[])
{
//main() definition starts
    int num=0,res;  // declares two integer with initial value 0
    Scanner in=new Scanner(System.in); //creating object of Scanner class toaccess the i/p stream.
    System.out.println("Enter a Number : ");
     num=in.nextInt();                   // to read the next integer value from the i/pstream
    res=num*2;                           // manipulation of the data
     System.out.println("The value of "+num+" * 2 = "+res); //displaysresult
}
}
```

Save: Example2.java **Compile:** javac Example2.java**Run:** java Example2

Output:
Enter a Number: 25
The value of 25 * 2 = 50

1.5: JAVA – DATA TYPES

Data type is used to allocate sufficient memory space for the data. Data types specify the different sizes and values that can be stored in the variable.

- ***Java is a strongly Typed Language.***
- Definition: ***strongly Typed Language:***

Java is a strongly typed programming language because every variable must be declared with a data type. A variable cannot start off life without knowing the range of values it can hold, and once it is declared, the data type of the variable cannot change.

Data types in Java are of **two types**:

1. **Primitive data types (Intrinsic or built-in types) :- :** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types (Derived or Reference Types):** The non-primitive data types include Classes, Interfaces, Strings and Arrays.

1. Primitive Types:

Primitive data types are those whose variables allow us to store only one value and neverallow storing multiple values of same type. This is a data type whose variable can hold maximum one value at a time.

There are eight primitive types in Java:

Integer Types:

1. int
2. short
3. lomg
4. byte

Floating-point Types:

5. float
6. double

Others:

7. char
8. Boolean

- **Integer Types:**

The integer types are form numbers without fractional parts. Negative values are allowed.Java provides the four integer types shown below:

Type	Storage Requirement	Range	Example	Default Value
int	4 bytes	-2,147,483,648(-2^31) to 2,147,483,647 (2^31-1)	int a = 100000, int b = -200000	0
short	2 bytes	-32,768 (-2^15) to 32,767 (2^15-1)	short s = 10000, short r = -20000	0
long	8 bytes	-9,223,372,036,854,775,808 (-2^63) to 9,223,372,036,854,775,808 (2^63-1)	long a = 100000L, int b = -200000L	0L
byte	1 byte	-128 (-2^7) to 127 (2^7-1)	byte a = 100 , byte b = -50	0

- **Floating-point Types:**

The floating-point types denote numbers with fractional parts. The two floating-pointtypes are shown below:

Type	Storage Requirement	Range	Example	Default Value
float	4 bytes	Approximately ±3.40282347E+38F (6-7 significant decimal digits)	float f1 =234.5f	0.0f
double	8 bytes	Approximately ±1.79769313486231570E+308 (15 significant decimal digits)	double d1 = 123.4	0.0d

- **char:**
  - char data type is a single 16-bit Unicode character.
  - Minimum value is '\u0000' (or 0).
  - Maximum value is '\uffff' (or 65,535 inclusive).
  - Char data type is used to store any character.
  - Example: char letterA ='A'

- **boolean:**
  - boolean data type represents one bit of information.
  - There are only two possible values: true and false.
  - This data type is used for simple flags that track true/false conditions.
  - Default value is false.
  - Example: boolean one = true

2. Derived Types (Reference Types):

- **Derived data types are those whose variables allow us to store multiple values of same type. But they never allow storing multiple values of different types.**
- A reference variable can be used to refer to any object of the declared type or any

compatible type.

- These are the data type whose variable can hold more than one value of similar type.
- **The value of a reference type variable, in contrast to that of a primitive type, is a reference to (an address of) the value or set of values represented by the variable.**
- Example

```
int a[] = {10,20,30};                  // valid
int b[] = {100, 'A', "ABC"};        // invalid
Animal animal = new Animal("giraffe");  //Object
```

1.6: JAVA - VARIABLES

- **A Variable is a named piece of memory that is used for storing data in java Program.**
- A variable is an identifier used for storing a data value.
- A Variable may take different values at different times during the execution if the program, unlike the constants.
- The variable's type determines what values it can hold and what operations can beperformed on it.
- **Syntax to declare variables:**

  datatype identifier [=value][,identifier [=value] ...];

- Example of Variable names:

  int average=0.0, height, total height;

- **Rules followed for variable names (consist of alphabets, digits, underscore anddollar characters)**
  1. A variable name must begin with a letter and must be a sequence of letter or digits.
  2. They must not begin with digits.
  3. Uppercase and lowercase variables are not the same.
     a. **Example:** Total and total are two variables which are distinct.
  4. It should not be a keyword.
  5. Whitespace is not allowed.
  6. Variable names can be of any length.

- **Initializing Variables:**
  - ✓ After the declaration of a variable, it must be initialized by means of assignment statement.
  - ✓ It is not possible to use the values of uninitialized variables.

- ✓ Two ways to initialize a variable:

1. Initialize after declaration:

Syntax: `variablename=value;`

```
int months;
months=1;
```

2. Declare and initialize on the same line:

Syntax: `Datatype variablename=value;`

```
int months=12;
```

- **Dynamic Initialization of a Variable:**

Java allows variables to be initialized dynamically using any valid expression at the timethe variable is declared.

Example: Program that computes the remainder of the division operation:

```
class FindRemainer
{
public static void main(String arg[]) {int num=5,den=2;
int rem=num%den; System.out.println(—Remainder is —+rem);
}
}
```

Output:

Remainder is 1

In the above program there are three variables **num, den** and **rem. num** and **den** ate initialized by constants whereas **rem** is initialized dynamically by the modulo division operation on **num** and **den.**

JAVA - VARIABLE TYPES

There are three kinds of variables in Java:

1. Local variables
2. Instance variables
3. Class/static variables

Local Variables	Instance Variable	Class / Static Variables
Local variables aredeclared in methods, constructors, or blocks.	Instance variables are declared in a class, but outside a method, constructor or any block.	Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block. There would only be one copy of each class variable per class, regardless of how many objects are created from it.
Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.	Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.	Static variables are created when the program starts and destroyed when the program stops.
Access modifiers cannot be used for local variables.	Access modifiers can be used forinstance variables.	Access modifiers can be used for class variables.
Local variables are visible only within the declared method, constructor or block.	The instance variables are visible for all methods, constructors and block in the class.	Visibility is similar to instance variables.
There is no default value for local variables so local variables should be declared and an initial value shouldbe assigned before the first use.	Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.	Default values are same as instance variables.
Local variables can only be access inside the declared block.	Instance variables can be accesseddirectly by calling the variable name inside the class. However within static methods and different class should be called using the fully qualified name as follows: **ObjectReference.VariableName.**	Static variables can be accessed by calling with the class name. **ClassName.VariableName.**

Example program illustrating the use of all the above variables:

```
class area
{
int length=20;
int breadth=30;
```

```
  static int classvar=2500;
  void calc()
  {
            int areas=length*breadth;
            System.out.println("The area is "+areas+" sq.cms");
  }

 public static void main(String args[])
 {
    area a=new area();
    a.calc();
    System.out.println("Static Variable Value : "+classvar);
  }
  }
```

Output:

The area is 600 sq.cms
Static Variable Value : 2500

Program Explanation:

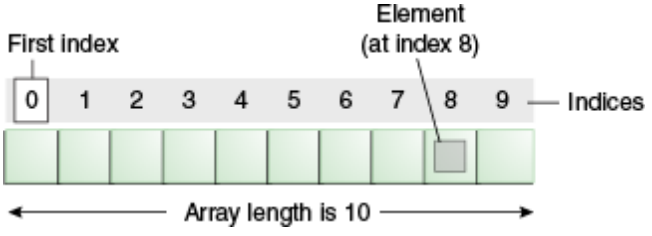
Class name: area
Method names: calc() and main()
Local variables: areas (accessed only in the particular method)
Instance variables: length and breadth (accessed only through the object's method)
Static variable: accessed anywhere in the program, without object reference

1.7: ARRAYS

Definition:

An array is a collection of similar type of elements which has contiguous memory location.

Java array is an object which contains elements of a similar data type.
Additionally, The elements of an array are stored in a contiguous memory location.
It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.
Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.



Advantage of Array:

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

Disadvantage of Array:

- **Size Limit:** We can store only fixed size of elements in the array. It doesn't growits size at runtime.

Types of Array:

There are two types of array.

1. One-Dimensional Arrays
2. Multidimensional Arrays

1. **One-Dimensional Array:**

Definition: One-dimensional array is an array in which the elements are stored in onevariable name by using only one subscript.

- **Creating an array:**

Three steps to create an array:

1. Declaration of the array
2. Instantiation of the array
3. Initialization of arrays

1. Declaration of the array:

Declaration of array means the specification of array variable, data_type and array_name.

Syntax to Declare an Array in java:

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

Example:

int[] floppy; (or) int []floppy (or) int floppy[];

2. **Instantiation of the array:**

Definition:

Allocating memory spaces for the declared array in memory (RAM) is called as **Instantiation of an array.**

Syntax:

arrayRefVar=new datatype[size];

Example: floppy=new int[10];

3. **Initialization of arrays:Definition:**

Storing the values in the array element is called as **Initialization of arrays.**

Syntax to initialize values to array element:

arrayRefVar[index value]=constant or value;

Example:

floppy[0]=20;

SHORTHAND TO CREATE AN ARRAY OBJECT:

Java has shorthand to create an array object and supply initial values at the same timewhen it is created.

```
dataType[] arrayRefVar={list of values};
                    (or)
dataType []arrayRefVar={list of values};
                    (or)
dataType arrayRefVar[]={list of values};
                    (or)
dataType arrayRefVar[]=arrayVariable;
```

Example 1:

```
int regno[]={101,102,103,104,105,106};
int reg[]=regno;
```

Example 2: double[] myList = new double[10];

ARRAY LENGTH:

The variable **length** can identify the length of array in Java. To find the number of elements of an array, use **array.length.**

Example1:

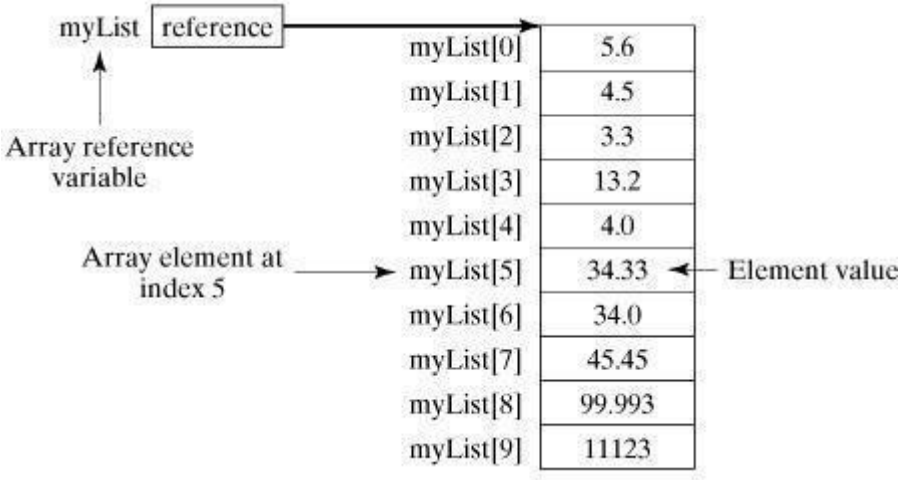
int regno[10]; len1=regno.length;

Example 2:

```
for(int i=0;i<reno.length;i++)
        System.out.println(regno[i]);
```

Following picture represents array myList. Here, myList holds ten double values and the indicesare from 0 to 9.

Example: (One-Dimensional Array)




```
class Array
{
public static void main(String[] args)
{
        int month_days[];
        month_days=new int[12];
        month_days[0]=31;
        month_days[1]=28;
        month_days[2]=31;
        month_days[3]=30;
        month_days[4]=31;
        month_days[5]=30;
        month_days[6]=31;
        month_days[7]=31;
        month_days[8]=30;
        month_days[9]=31;
        month_days[10]=30;
        month_days[11]=31;

        System.out.println("April has "+month_days[3]+ " days.");
}
}
```

Output:

April has 30 days.

Example 2: Finding **sum of the array elements** and **maximum** from the array:

```
public class TestArray
{
public static void main(String[] args)
{
        double[] myList = {1.9, 2.9, 3.4, 3.5};

        // Print all the array elements
        for (double element: myList)
        {
                System.out.println(element);
        }

        // Summing all elementsdouble total = 0;
        for (int i = 0; i < myList.length; i++)
        {
                total += myList[i];
        }
        System.out.println("Total is " + total);

        // Finding the largest elementdouble

        max = myList[0];
        for (int i = 1; i < myList.length; i++)
        {
                if (myList[i] > max)
                        max = myList[i];
        }
        System.out.println("Max is " + max);
    }
}
```

Output:

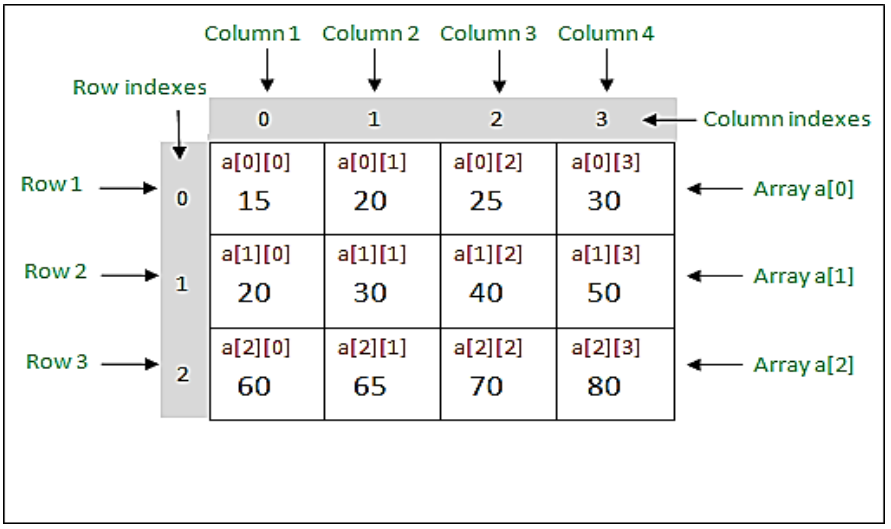
1.9
2.9
3.4
3.5

Total is 11.7
Max is 3.5

2. Multidimensional Arrays:

Definition:

Multidimensional arrays are ***arrays of arrays.*** It is an array which uses more than one index to access array elements. In multidimensional arrays, data is stored in row and column based index (also known as matrix form).



Uses of Multidimensional Arrays:

- ✓ Used for table
- ✓ Used for more complex arrangements

Syntax to Declare Multidimensional Array in java:

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

Example to instantiate Multidimensional Array in java:

```
int[][] arr=new int[3][3]; //3 row and 3 column - internally this matrix is implemented as arrays of arrays of int.
```

Example to initialize Multidimensional Array in java:

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
```

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

Examples to declare, instantiate, initialize and print the 2Dimensional array:

```
class twoDarray
{
public static void main(String args[])
{
int array1[][]=new int[4][5];// declares an 2D array.
int array2[][]={{1,2,3},{2,4,5},{4,4,5}}; //declaring and initializing 2D arrayint i,j,k=0;

  // Storing and printing the values of Array1

System.out.println("-------Array 1--------" );
for(i=0;i<4;i++)
{
     for(j=0;j<5;j++)
     {
          array1[i][j]=k;k++;
          System.out.print(array1[i][j]+ " ");
     }
     System.out.println();
}

// printing 2D array2
System.out.println("-------Array 2--------");
for(int i=0;i<3;i++)
     {
     for(int j=0;j<3;j++)
     {
     System.out.print(array2[i][j]+
     }
     System.out.println();
     }
     }
}
```

Output:

----------Array1----------

0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19

----------Array2----------

1 2 3
2 4 5
4 4 5

In the above program, the statement **int array1[][]=new int[4][5];** is interpreted automatically as follows:
array1[0]=new int[5];array1[1]=new int[5];array1[2]=new int[5];array1[3]=new int[5];

It means that, when we allocate memory for a multidimensional array, we need to only specify the memory for the first (leftmost) dimension. We can allocate the remaining dimensions separately with different sizes.

Example: Manually allocate differing size second dimensions:

```
class twoDarray
{
public static void main(String args[])
{
int array1[][]=new int[4][];                    // declares an 2D array.

array1[0]=new int[1];
array1[1]=new int[2];
array1[2]=new int[3];
array1[3]=new int[4];
int i,j,k=0;

  // Storing and printing the values of Array
for(i=0;i<4;i++)
{
     for(j=0;j<i+1;j++)
     {
               array1[i][j]=k;k++;
               System.out.print(array1[i][j]+ " ");
```

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

Output:

0
1 2
3 4 5
6 7 8 9

1.8: OPERATORS

Operators are used to manipulate primitive data types.
Java operators can be classified as unary,binary, or ternary—meaning taking one, two, or three arguments, respectively.

Java Unary Operator

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

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

Java Unary Operator Example: ++ and –

```
1. class OperatorExample
2. {
3. public static void main(String args[])
4. {
5. int x=10;
6. System.out.println(x++);          //10 (11)
7. System.out.println(++x);          //12
8. System.out.println(x--);          //12 (11)
9. System.out.println(--x);          //108.
10.}
11.}
```

Output:

10
12
12
10

Java Unary Operator Example 2: ++ and –

```
1. class OperatorExample
2. {
3. public static void main(String args[])
4. {
5. int a=10;
6. int b=10;
7. System.out.println(a++ + ++a);          //10+12=22
8. System.out.println(b++ + b++);          //10+11=21 7.
9.  }
10. }
```

Output:

22
21

Java Unary Operator Example: ~ and !

```
1. class OperatorExample{
2. public static void main(String args[]){
3.       int a=10;
4.       int b=-10;
5. boolean c=true;
6. boolean d=false;
7. System.out.println(~a);          //-11 (minus of total positive value which starts from 0)
8. System.out.println(~b);                  //9 (positive of total minus, positive starts from 0)
9. System.out.println(!c);          //false (opposite of boolean value)
10.      System.out.println(!d);                  //true
11.      }
12.      }
```

Output:

-11
9
False
true

A binary or ternary operator appears between its arguments.Java operators

eight different categories:

1. **Assignment**
2. **Arithmetic**
3. **Relational**
4. **Logical**
5. **Bitwise**
6. **Compound assignment**
7. **Conditional**
8. **Type**.

Assignment Operators	**=**
Arithmetic Operators	**- + * / % ++ --**
Relational Operators	**> < >= <= == !=**
Logical Operators	**&& || & | ! ^**
Bit wise Operator	**& | ^ >> >>>**
Compound Assignment Operators	**+= -= *= /= %= <<= >>= >>>=**
Conditional Operator	**?:**

1. Java Assignment Operator

The java assignment operator statement has the following syntax:

<*variable*> = <*expression*>

If the value already exists in the variable it is overwritten by the assignment operator (=).

Java Assignment Operator Example

```
1. class OperatorExample{
2. public static void main(String args[])
3. {
4. int a=10;
5. int b=20;
6. a+=4;                //a=a+4 (a=10+4)
7. b-=4;                //b=b-4 (b=20-4)
8. System.out.println(a);
9. System.out.println(b);
10. }
11. }
```

Output:

```
14
16
```

2. Java Arithmetic Operators

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

Assume integer variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	A - B will give -10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200
/	Division - Divides left hand operand by right hand operand	B / A will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0
++	Increment - Increases the value of operand by 1	B++ gives 21
--	Decrement - Decreases the value of operand by 1	B-- gives 19

Java Arithmetic Operator Example: Expression

```
1. class OperatorExample
2. {
3. public static void main(String args[])
4. {
5.     System.out.println(10*10/5+3-1*4/2);4.
6. }
7. }
```

Output:

```
21
```

3. Relational Operators

Relational operators in Java are used to compare 2 or more objects. Java provides sixrelational operators: Assume variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, ifvalues are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value ofright operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal tothe value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomestrue.	(A <= B) is true.

Example:

```
public RelationalOperatorsDemo( )
{
      int x = 10, y = 5;
      System.out.println("x > y : "+(x > y));
      System.out.println("x < y : "+(x < y));
      System.out.println("x >= y : "+(x >= y));
      System.out.println("x <= y : "+(x <= y));
      System.out.println("x == y : "+(x == y));
      System.out.println("x != y : "+(x != y));

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

Output:

```
$java RelationalOperatorsDemo
```

x > y : true
x < y : false
x >= y : true
x <= y : false
x == y : false
x != y : true

4. Logical Operators

Logical operators return a true or false value based on the state of the Variables. Given that x and y represent boolean expressions, the boolean logical operators are defined in the Table below.

x	y	!x	x & y x && y	x | y x || y	x ^ y
true	true	false	true	true	False
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

Example:

```
public class LogicalOperatorsDemo
{
   public LogicalOperatorsDemo()
   {
      boolean x = true;
      boolean y = false;
      System.out.println("x & y : " + (x & y));
      System.out.println("x && y : " + (x && y));
      System.out.println("x | y : " + (x | y));
      System.out.println("x || y: " + (x || y));
      System.out.println("x ^ y : " + (x ^ y));
      System.out.println("!x : " + (!x));
   }
public static void main(String args[])
{
new LogicalOperatorsDemo();
}
}
```

Output:

```
$java LogicalOperatorsDemo
```

x & y : false
x && y : false
x | y : true
x || y: true
x ^ y : true
!x : false

5. Bitwise Operators

Java provides Bit wise operators to manipulate the contents of variables at the bit level. The result of applying bitwise operators between two corresponding bits in the operandsis shown in the Table below.

A	B	~A	A & B	A | B	A ^ B
1	**1**	0	1	1	0
1	**0**	0	0	1	1
0	**1**	1	0	1	1
0	**0**	1	0	0	0

```
public class Test
{
public static void main(String args[])
{
    int a = 60;  /* 60 = 0011 1100 */
    int b = 13;    /* 13 = 0000 1101 */int c = 0;
    c = a & b;      /* 12 = 0000 1100 */
    System.out.println("a & b = " + c );
    c = a | b;     /* 61 = 0011 1101 */
    System.out.println("a | b = " + c );
    c = a ^ b;     /* 49 = 0011 0001 */
    System.out.println("a ^ b = " + c );
    c = ~a;        /*-61 = 1100 0011 */
    System.out.println("~a = " + c );
    c = a << 2;    /* 240 = 1111 0000 */
    System.out.println("a << 2 = " + c );
    c = a >> 2;    /* 215 = 1111 */
    System.out.println("a >> 2  = " + c );
    c = a >>> 2;    /* 215 = 0000 1111 */
```

```
    System.out.println("a >>> 2 = " + c );
}
}
```

Output:

```
$java Test

  a & b = 12
  a | b = 61
  a ^ b = 49
  ~a = -61
  a << 2 = 240
  a >> 2  = 15
  a >>> 2 = 15
```

6. Compound Assignment operators

The compound operators perform shortcuts in common programming operations. Javahas eleven compound assignment operators.

Syntax: argument1 **operator** = argument2.

Java Assignment Operator Example

```
1. class OperatorExample
2. {
3. public static void main(String[] args)
4. {
5. int a=10;
6. a+=3;     //10+3
7. System.out.println(a);
8. a-=4;     //13-4
9. System.out.println(a);
10. a*=2;    //9*2
11. System.out.println(a);
12. a/=2;    //18/2
13. System.out.println(a);12.
14. }
15. }
```

Output:

9
18
9

7. Conditional Operators

The Conditional operator is the only ternary (operator takes three arguments) operator in Java. The operator evaluates the first argument and, if true, evaluates the second argument.

If the first argument evaluates to false, then the third argument is evaluated. The conditional operator is the expression equivalent of the if-else statement.

The conditional expression can be nested and the conditional operator associates from right to left: **(a?b?c?d:e:f:g) evaluates as (a?(b?(c?d:e):f):g)**

Example:

```
public class TernaryOperatorsDemo {

        public TernaryOperatorsDemo() {
                int x = 10, y = 12, z = 0;
                z = x > y ? x : y;
                System.out.println("z : " + z);
        }
        public static void main(String args[]) {
                new TernaryOperatorsDemo();
        }
}
```

Output:

```
$java TernaryOperatorsDemo
z : 12
```

8. instanceof Operator:

This operator is used only for object reference variables. The operator checks whether the object is of a particular type(class type or interface type). instanceof operator is written as:

(Object reference variable) instanceof (class/interface type)

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is the

Example:

```
public class Test
{
   public static void main(String args[])
  {
      String name = "James";
       // following will return true since name is type of String
       boolean result = name instanceof String;
       System.out.println( result );
   }
}
```

This would produce the following result:

True

OPERATOR PRECEDENCE:

The order in which operators are applied is known as precedence. Operators with a higher precedence are applied before operators with a lower precedence.

The operator precedence order of Java is shown below. Operators at the top of the table are applied before operators lower down in the table.

If two operators have the same precedence, they are applied in the order they appear in a statement. That is, from left to right. You can use parentheses to override the default precedence.

Category	**Operator**	**Associativity**
Postfix	() [] . (dot operator)	Left to right
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right

Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Example:

In an operation such as,

result = 4 + 5 * 3

First (5 * 3) is evaluated and the result is added to 4 giving the Final Result value as 19. Note that _*' takes higher precedence than _+' according to chart shown above. This kindof precedence of one operator over another applies to all the operators.

1.9: CONTROL-FLOW STATEMENTS

Java Control statements control the order of execution in a java program, based on data values and conditional logic.

There are three main categories of control flow statements;

- **Selection statements:** if, if-else and switch.
- **Loop statements:** while, do-while and for.
- **Transfer statements:** break, continue, return, try-catch-finally and assert.

We use control statements when we want to change the default sequential order of execution

1. Selection statements (Decision Making Statement)

There are two types of decision making statements in Java. They are:

- if statements
- if-else statements
- nested if statements
- if-else if-else statements
- switch statements

if Statement:

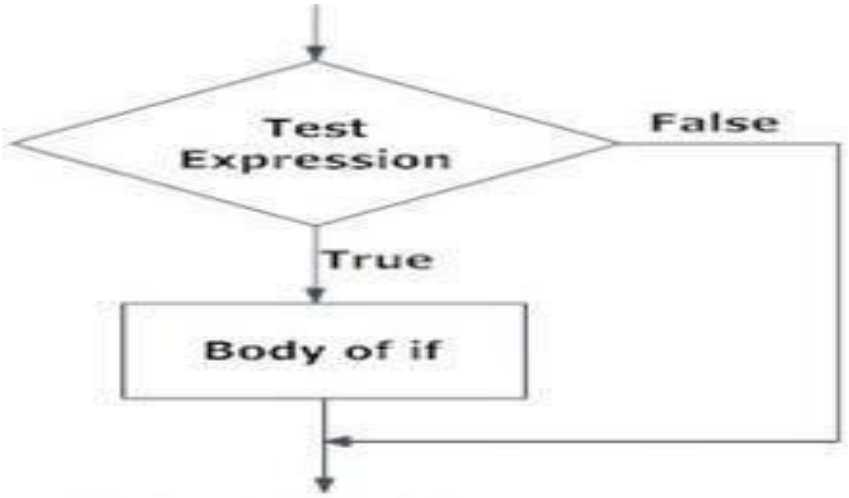
- An if statement consists of a Boolean expression followed by one or more statements.
- Block of statement is executed when the condition is true otherwise no statement will beexecuted.

Syntax:

if(<conditional expression>)
{
< Statement Action>
}

If the Boolean expression evaluates to true then the block of code inside the if statement will be executed.
If not the first set of code after the end of the if statement (after the closingcurly brace) will be executed.

Flowchart:



Example:

```
public class IfStatementDemo {

        public static void main(String[] args)
        {
        int a = 10, b = 20;
        if (a > b)
              System.out.println("a > b");
        if (a < b)
              System.out.println("b > a");
        }
      }
```

Output:

```
$java IfStatementDemo
b > a
```

if-else Statement:

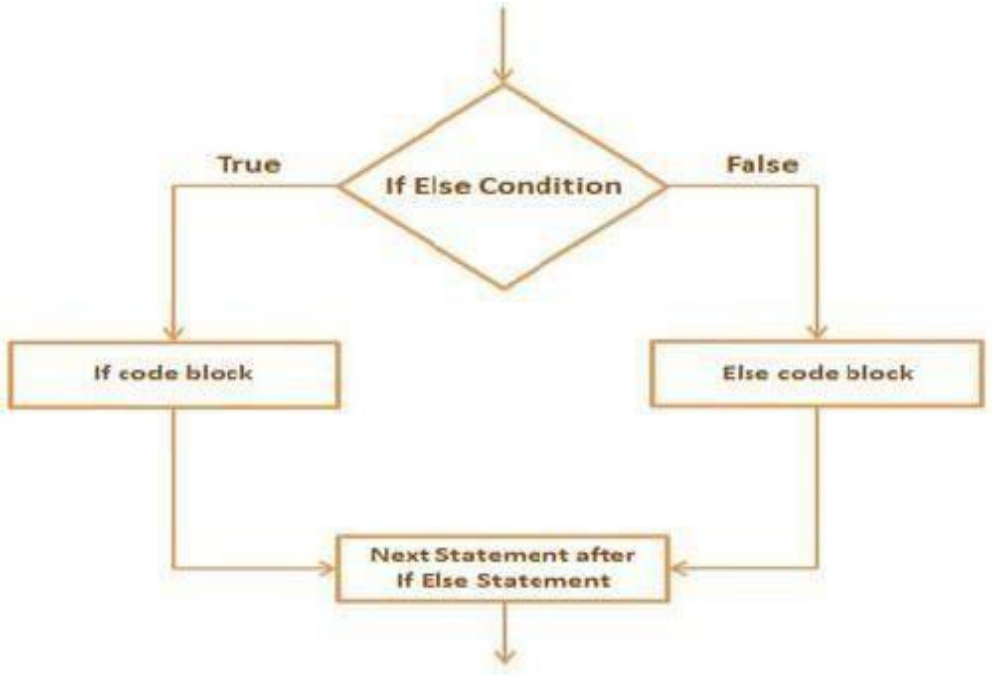
The if/else statement is an extension of the if statement. If the statements in the

ifstatement fails, the statements in the else block are executed.

Syntax:

The if-else statement has the following syntax:

```
if(<conditional expression>)
{
  < Statement Action1>
}
else
{
  < Statement Action2>
}
```



Example:

```
public class IfElseStatementDemo {

        public static void main(String[] args)
        {
                int a = 10, b = 20;
                if (a > b) {
                        System.out.println("a > b");
                }
                else {
                        System.out.println("b > a");
                }
        }
}
```

Output:

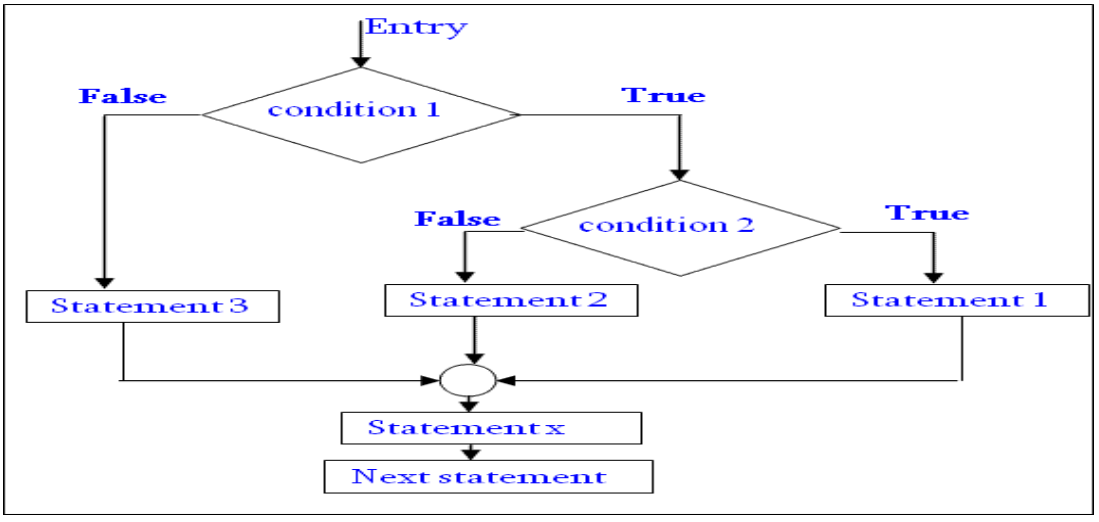
```
$java IfElseStatementDemo
      b > a
```

Nested if Statement:

Nested if-else statements, is that using one if or else if statement inside another if or else ifstatement(s).

Syntax:

```
if(condition1)
{
    if(condition2)
    {
    //Executes this block if  condition is True
    }
    else
    {
     //Executes this block if condition is false
    }
}
else
{
 //Executes this block if condition is false
}
```



Example-nested-if statement:

```
class NestedIfDemo
{
```

```
public static void main(String args[])
{
int i = 10;
if (i ==10)
{
if (i < 15)
{
System.out.println("i is smaller than 15");
}
else
{
System.out.println("i is greater than 15");
}
}
else
{
System.out.println("i is greater than 15");
}
}
}
```

Output:

i is smaller than 15

if...else if...else Statement:

An if statement can be followed by an optional *else if...else* statement, which is veryuseful to test various conditions using single if...else if statement.

Syntax:

```
if(Boolean_expression 1){
  //Executes when the Boolean expression 1 is true
}else if(Boolean_expression 2){
  //Executes when the Boolean expression 2 is true
}else if(Boolean_expression 3){
  //Executes when the Boolean expression 3 is true
}else {
  //Executes when the none of the above condition is true.
}
```

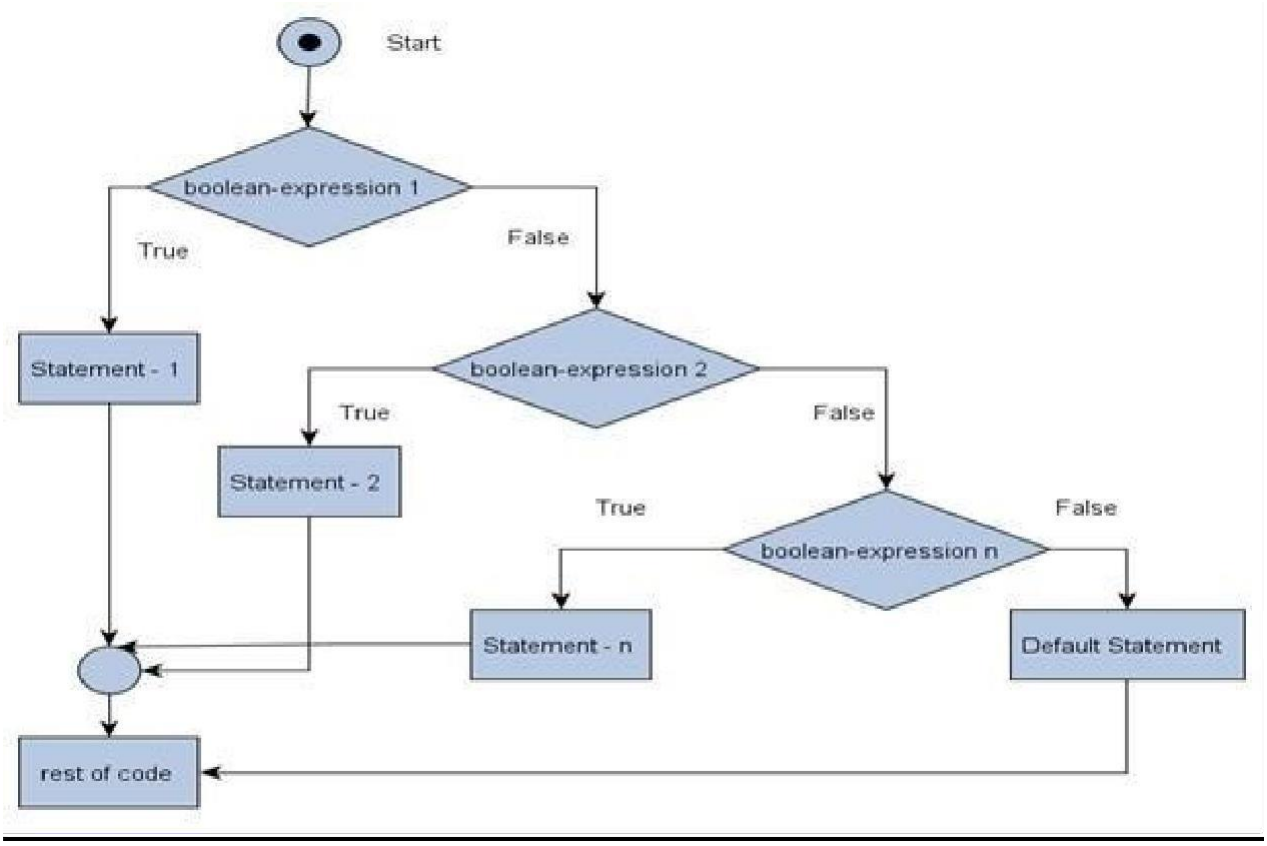
Example:

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

```
    int x = 30;

    if( x == 10 ){

      System.out.print("Value of X is 10");
    }else if( x == 20 ){
      System.out.print("Value of X is 20");
    }else if( x == 30 ){
      System.out.print("Value of X is 30");
    }else{
      System.out.print("This is else statement");
    }
  }
}
```

Output:

Value of X is 30



switch Statement:

- **The switch case statement, also called a case statement is a multi-way branch with several choices. A switch is easier to implement than a series of if/else statements.**
- A *switch* statement allows a variable to be tested for equality against a list of

values. Eachvalue is called a case, and the variable being switched on is checked for each case.

- The switch statement begins with a keyword, followed by an expression that equates to a no long integral value. Following the controlling expression is a code block that contains zero or more labeled cases. Each label must equate to an integer constant and each must be unique.
- When the switch statement executes, it compares the value of the controlling expression to the values of each case label.
- The program will select the value of the case label that equals the value of the controllingexpression and branch down that path to the end of the code block.
- If none of the case label values match, then none of the codes within the switch statementcode block will be executed.
- Java includes a default label to use in cases where there are no matches. We can have a nested switch within a case block of an outer switch.

Syntax:

switch (<expression>)
{
case label1:
<statement1>
case label2:
<statement2>
…
case labeln:
<statementn>
default:
<statement>
}

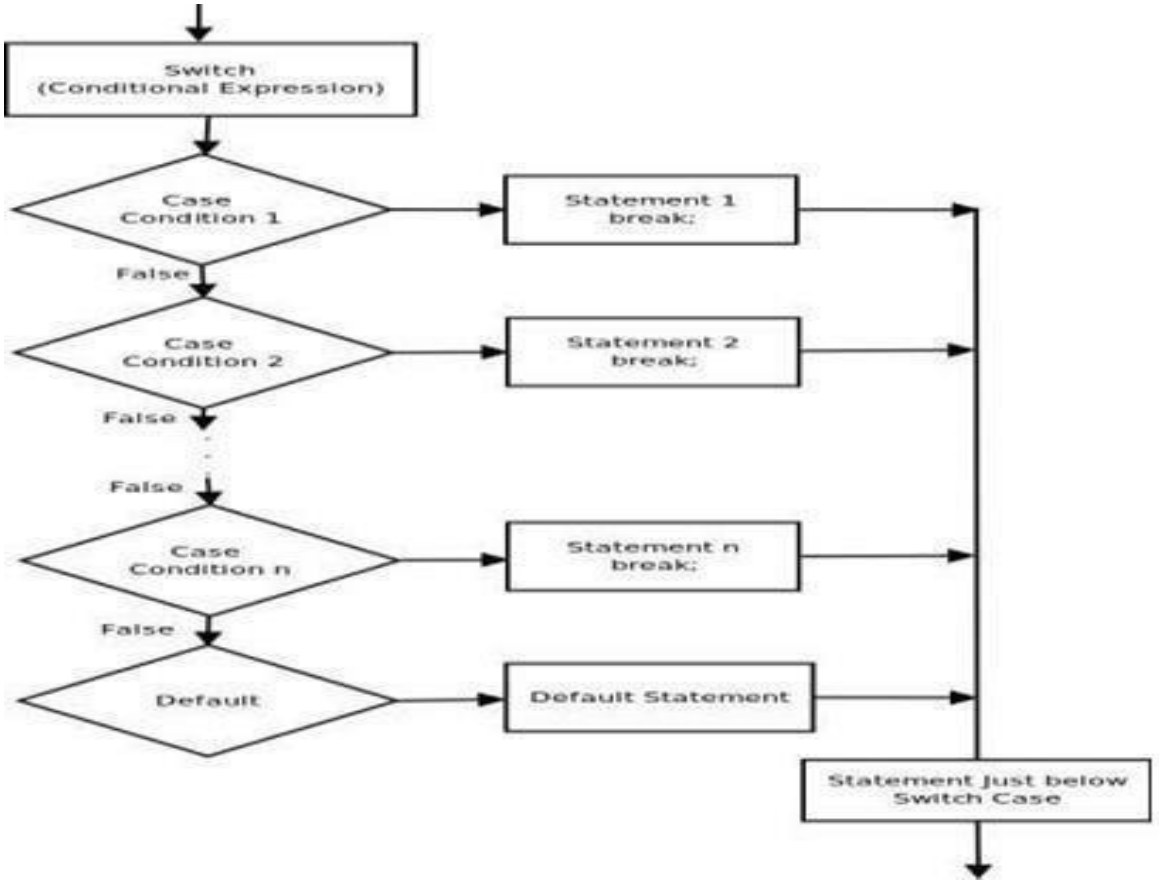
Example:

```
public class SwitchCaseStatementDemo {

    public static void main(String[] args) {int a =
            10, b = 20, c = 30;
            int status = -1;
            if (a > b && a > c) {
                    status = 1;
            } else if (b > c) {
                    status = 2;
            } else {
```

```
                status = 3;
            }
            switch (status) {case 1:
                System.out.println("a is the greatest");break;
            case 2:
                System.out.println("b is the greatest");break;
            case 3:
                System.out.println("c is the greatest");break;
            default:
                System.out.println("Cannot be determined");
            }
        }
}
```

utput:

c is the greatest



2. Looping Statements (Iteration Statements)

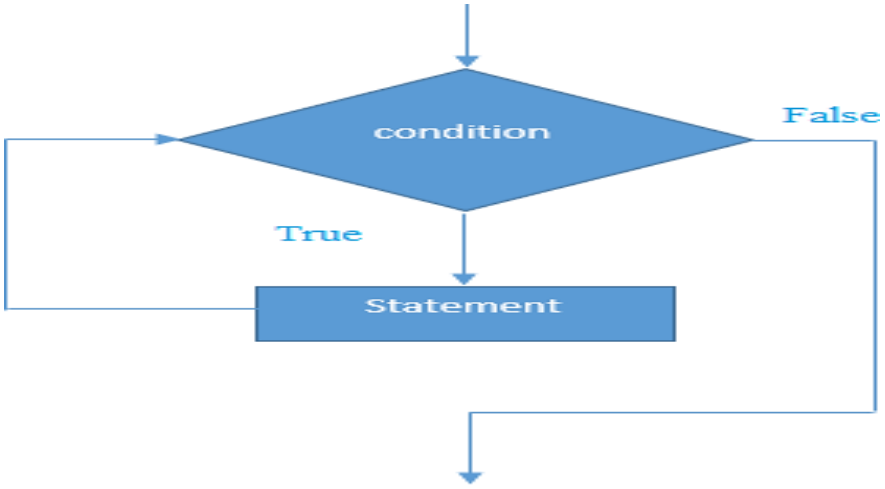
While Statement

- The while statement is a looping control statement that executes a blockof code while a condition is true. It is entry controlled loop.
- You can either have a single statement or a block of code within the while loop.The loop will never be executed if the testing expression evaluates to false.
- The loop condition must be a boolean expression.

Syntax:

The syntax of the while loop is

```
while (<loop condition>)
{
<statements>
}
```



Example:

```
public class WhileLoopDemo {
        public static void main(String[] args) {int
                count = 1;
                System.out.println("Printing Numbers from 1 to 10");
                while (count <= 10) {
                        System.out.println(count++);
                }
        }
}
```

Output

Printing Numbers from 1 to 10

1
2
3

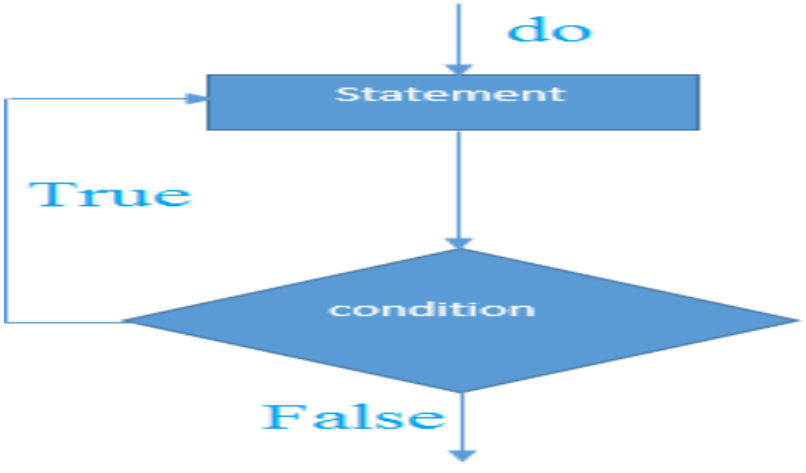
4
5
6
7
8
9
10

do-while Loop Statement

- do while loop checks the condition after executing the statements atleast once.
- Therefore it is called as Exit Controlled Loop.
- The do-while loop is similar to the while loop, except that the test is performed at the endof the loop instead of at the beginning.
- This ensures that the loop will be executed at least once. A do-while loop begins with the keyword do, followed by the statements that make up the body of the loop.

Syntax:

```
do
{
<loop body>
}while (<loop condition>);
```



Example:

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

```
        int count = 1;
        System.out.println("Printing Numbers from 1 to 10");
        do {
                System.out.println(count++);
        } while (count <= 10);
  }
}
```

Output:

```
Printing Numbers from 1 to 10
1
2
3
4
5
6
7
8
9
10
```

For Loops

The for loop is a looping construct which can execute a set of instructions a specifiednumber of times. It's a counter controlled loop. A for statement consumes the initialization, condition and increment/decrement in one line. It is the entry controlled loop.

Syntax:

```
for (<initialization>; <loop condition>; <increment expression>)
{
        <loop body>
}
```

- ✓ The first part of a for statement is a starting initialization, which executes once before the loop begins. The <initialization> section can also be a comma-separatedlist of expression statements.
- ✓ The second part of a for statement is a test expression. As long as the expression istrue, the loop will continue. If this expression is evaluated as false the first time, theloop will never be executed.

- ✓ The third part of the for statement is the body of the loop. These are the instructionsthat are repeated each time the program executes the loop.
- ✓ The final part of the for statement is an increment expression that automatically executes after each repetition of the loop body. Typically, this statement changes the value of the counter, which is then tested to see if the loop should continue.

Exmple:

```
public class ForLoopDemo {
        public static void main(String[] args)
        {
        System.out.println("Printing Numbers from 1 to
                10");
        for (int count = 1; count <= 10; count++)
        {
                        System.out.println(count);
       }
    }
}
```

Output:

```
Printing Numbers from 1 to 10
1
2
3
4
5
6
7
8
9
10
```

Enhanced for loop or for- each loop:

As of Java 5, the enhanced for loop was introduced. This is mainly used for Arrays.

- ✓ The for-each loop is used to traverse array or collection in java.
- ✓ It is easier to use than simple for loop because we don't need to increment value and usesubscript notation.
- ✓ It works on elements basis not index.
- ✓ It returns element one by one in the defined variable.

Syntax:

```
for(declaration : expression)
{
   //Statements
}
```

- **Declaration:** The newly declared block variable, which is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
- **Expression:** This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

Example:

```
public class Test {


public static void main(String args[])
{
   int [] numbers = {10, 20, 30, 40, 50};

   for(int x : numbers )
   {
      System.out.print( x );
      System.out.print(",");
   }
   System.out.print("\n\n");
   String [] names ={"B", "C", "C++", "JAVA"};
   for( String name : names )
   {
      System.out.print( name );
      System.out.print(",");
   }
   }
   }
```

Output:

```
10,20,30,40,50,
B,C,C++,JAVA
```

3. Transfer Statements / Loop Control Statements/Jump Statements)

1. break statement
2. continue statement

1. Using break Statement:

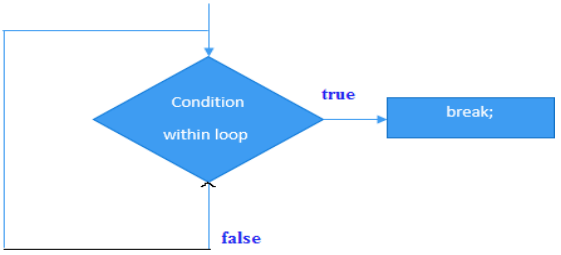
- ✓ The *break* keyword is used to stop the entire loop. The break keyword must beused inside any loop or a switch statement.
- ✓ The break keyword will stop the execution of the innermost loop and start executingthenext line of code after the block.

Syntax:

The syntax of a break is a single statement inside any loop:

```
break;
```

Flowchart:



Example:

```
public class Test {

public static void main(String args[]) { int [] numbers = {10, 20, 30, 40, 50};

for(int x : numbers ) { if( x == 30 ) { break;
}
System.out.print( x ); System.out.print("\n");
}
}
}
```

Output:

10

20

2. Using continue Statement:

- The *continue* keyword can be used in any of the loop control structures. It causes theloop to immediately jump to the next iteration of the loop.
- **The Java *continue statement* is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.**

Syntax:

The syntax of a continue is a single statement inside any loop:

continue;

Example:

```
public class Test {

  public static void main(String args[]) {int
    [] numbers = {10, 20, 30, 40, 50};

    for(int x : numbers )
    {
      if( x == 30 )
      {
        continue;
      }
      System.out.print( x );
      System.out.print("\n");
    }
  }
}
```

Output:

10
20
40
50

1.10: DEFINING CLASSES and OBJECTS

A class is a collection of similar objects and it contains data and methods that operate on that data. In other words – **Class is a blueprint or template for a set of objects that share a common structure and a common behavior**.

DEFINING A CLASS:

The keyword **class** is used to define a class.

Rules to be followed:

1. Classes must be enclosed in parentheses.
2. The class name, superclass name, instance variables and method names may be any validJava identifiers.
3. The instance variable declaration and the statements of the methods must end with ;(semicolon).
4. The keyword **extends** means derived from i.e. the class to the left of the **extends**
   (subclass) is derived from the class to the right of the **extends** (superclass).

Syntax to declare a class:

```
[public|abstract|final] class class_name [extends superclass_name implements interface_name]
{
        data_type instance_variable1;
        data_type instance_variable2;
        .
        .
        data_type instance_variableN;

        return_type method_name1(parameter list)
        {
        Body of the method
        }
        .
        .
        return_type method_nameN(parameter list)
        {
        Body of the method
        }
}
```

- ✓ The data, or variables, defined within a **class** are called *instance variables*.
- ✓ The code to do operations is contained within *methods*.
- ✓ Collectively, the methods and variables defined within a class are called *members* of

theclass.

- ✓ Variables defined within a class are called instance variables because each instance of theclass (that is, each object of the class) contains its own copy of these variables.
- ✓ Thus, the data for one object is separate and unique from the data for another.

✓ **Example:**

```
class box {
  double width;
  double height;
  double depth;
  void volume()
 {
            System.out.println(–\n Volume is : –);
            Systme.out.println(width*height*depth);
  }
}
```

Program Explanation:

Class : keyword that initiates a class definition
Box : class name
Double : primitive data type
Height, depth, width: Instance variables
Void : return type of the method
Volume() : method name that has no parameters

DEFINING OBJECTS

An **Object** is an instance of a class. It is a blending of methods and data.

Object = Data + Methods

- It is a structured set of data with a set of operations for manipulating that data.
- The methods are the only gateway to access the data. In other words, the methods and dataare grouped together and placed in a container called Object.

Characteristics of an object:

An object has three characteristics:

1) **State:** represents data (value) of an object.

2) **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.

3) **Identity:** Object identity is an unique ID used internally by the JVM to identify each object uniquely.

For Example: Pen is an object. Its name is Reynolds, color is white etc. known

state. It is used to write, so writing is its behavior.

CREATING OBJECTS:

Obtaining objects of a class is a two-step process:

1. Declare a variable of the class type – this variable does not define an object. Instead,it is simply a variable that can refer to an object.
2. Use **new** operator to create the physical copy of the object and assign the referenceto the declared variable.

NOTE: The **new** operator dynamically allocates memory for an object and returns a referenceto it. This reference is the address in memory of the object allocated by **new.**

Advantage of using new operator: A program can create as many as objects it needs duringthe execution of the program.

Syntax:

```
class_name object_name = new class_name();
    (or)
class_name object_name;
object_name = new class_name();
```

Example:

box b1=new box();(or)
box b2; b2=new box();

ACCESSING CLASS MEMBERS:

- ✓ Accessing the class members means accessing instance variable and instance methods in a class.
- ✓ To access these members, a dot (.) operator is used along with the objects.

Syntax for accessing the instance members and methods:

```
object_name.variable_name;
object_name.method_name(parameter_list);
```

Example:

```
class box
{
    double width;
    double height;
```

```
        double depth;
void volume()
{
        System.out.print("\n Box Volume is : ");
        System.out.println(width*height*depth+" cu.cms");
}
}
public class BoxVolume
{
public static void main(String[] args)
{
     box b1=new box(); // creating object of type box
     b1.width=10.00;      // Accessing instance variables through object
     b1.height=10.00;
     b1.depth=10.00;
     b1.volume();          // Accessing method through object
}
}
```

Output:

Box Volume is: 1000.0 cu.cms

1.11: METHODS

DEFINITION :

A Java method is a collection of statements that are grouped together to perform an operation.

Syntax: Method:

```
modifier Return –type method_name(parameter_list) throws exception_list
   {
// method body
    }
```

The syntax shown above includes:

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

Example:

This method takes two parameters num1 and num2 and returns the maximum between the two:

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

- **METHOD CALLING (Example for Method that takes parameters and returning value):**
  - ✓ For using a method, it should be called.

- ✓ A method may take any no. of arguments.
- ✓ A *parameter* is a variable defined by a method that receives a value when the method is called. For example, in **square( )**, **i** is a parameter.
- ✓ An *argument* is a value that is passed to a method when it is invoked. For example, **square(100)** passes 100 as an argument. Inside **square( )**, the parameter **i** receives that value.
- ✓ There are two ways in which a method is called.
- calling a method that returns a value or
- calling a method returning nothing (no return value).
- ✓ The process of method calling is simple. When a program invokes a method, the programcontrol gets transferred to the called method.
- ✓ This called method then returns control to the caller in two conditions, when:

1. return statement is executed.
2. reaches the method ending closing brace.

- ✓ **Example:**

Following is the example to demonstrate how to define a method and how to call it:

```
public class ExampleMinNumber
{
public static void main(String[] args)
{
int a = 11;
int b = 6;
int c = minFunction(a, b);
System.out.println("Minimum Value = " + c);
}

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

This would produce the following result:

```
Minimum value = 6
```

1.12: CONTRUCTORS

Definition:

Constructor is a **special type of method** that is used to initialize the object. Constructor is **invoked at the time of object creation**. Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes.

- **Rules for creating constructor:**

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type
3. Constructors can be declared public or private (for a Singleton)
4. Constructors can have no-arguments, some arguments and var-args;
5. A constructor is always called with the **new** operator
6. The default constructor is a no-arguments one;
7. If you don't write ANY constructor, the compiler will generate the default one;
8. Constructors CAN'T be **static**, **final** or **abstract**;
9. When overloading constructors (defining methods with the same name but with different arguments lists) you must define them with different arguments lists (as number or as type)

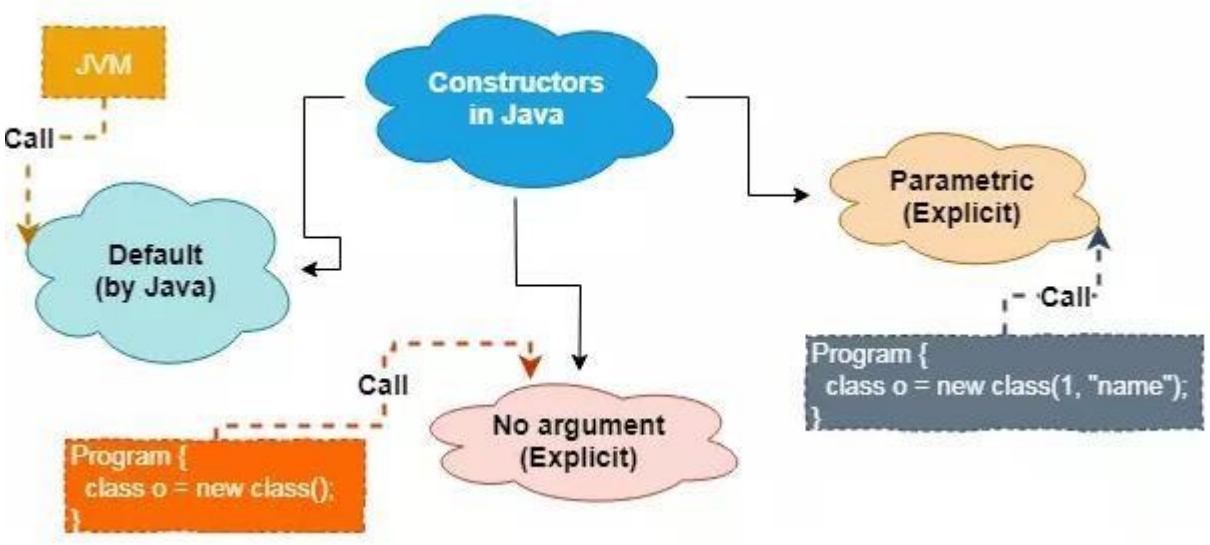
- **What happens when a constructor is called?**

1. All data fields are initialized to their default value (0, false or null).
2. All field initializers and initialization blocks are executed, in the order in which theyoccur in the class declaration.
3. If the first line of the constructor calls a second constructor, then the body of thesecond constructor is executed.
4. The body of the constructor is executed.

- **Types of constructors**

There are two types of constructors:

1. Default constructor
2. no-arg constructor
3. Parameterized constructor



1. Default Constructor

- **Default constructor refers to a constructor that is automatically created by compilerin the absence of explicit constructors.**

Rule: If there is no constructor in a class, compiler automaticaaly creates a default constructor.

Purpose of Default Constructor: It is used to provide the default values to the object members like 0, null etc. depending on the data type.

Example:

```
class student
{
        int id;
        String name;
        void display()
        {
                System.out.println(id+&amp;quot; &amp;quot;+name);
        }
        public static void main(String args[])
        {
                student s1=new student();
                student s2=new student();
                s1.display();
                s2.display();
        }
}
```

Output:

```
0 null
1  null
```

2) No-Argument Constructor

- **Constructor without parameters is called no-argument constructor**.

Purpose of No-Arg Constructor: It is used to provide values to be common for all objects of the class.

Syntax of default constructor:

```
Classname()
{
    // Constructor body
}
```

Example:

```
class Box
{
  double  width;
  double height;
  double depth;

  // This is the constructor for Box
  Box()
  {
        System.out.println("Constructing Box…");
        width=10;
        height=10;
        depth=10;
  }
// Compute and return volume
 double volume()
 {
          return width*height*depth;
 }
}
class BoxDemo
{
        public static void main(String arg[])
        {
                // declare, allocate and initialize Box objects
```

```
        Box mybox1=new Box();
        Box mybox2=new Box();
        double vol;

        // Get volume of first box
        vol=mybox1.volume();
        System.out.println("Volume is " +vol);

        // Get volume of second box
        vol=mybox2.volume();
        System.out.println("Volume is "+vol);
    }
}
```

Output:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

As you can see, both **mybox1** and **mybox2** were initialized by the **Box( )** constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both **mybox1** and **mybox2** will have the same volume.

3. Parameterized Constructor

A constructor that takes parameters is known as parameterized constructor.

Purpose of parameterized constructor

Parameterized constructor is used to provide different values to the distinct objects.

Example:

```
class Box
{
  double  width;
  double height;
  double depth;

  // This is the constructor for Box
```

```
  Box(double w, double h, double d)
  {
        width=w;height=h;depth=d;
  }

// Compute and return volume
  double volume()
  {
 return width*height*depth;
  }
}
class BoxDemo
{
public static void main(String arg[])
{
        // declare, allocate and initialize Box objects
        Box mybox1=new Box(10,20,15);
        Box mybox2=new Box(3,6,9);
        double vol;
        // Get volume of first box

        vol=mybox1.volume();
        System.out.println("Volume is " +vol);

        // Get volume of second box
        vol=mybox2.volume();
        System.out.println("Volume is " +vol);
}
}
```

Output:

Volume is 3000.0
Volume is 162.0

As you can see, each object is initialized as specified in the parameters to its constructor. Forexample, in the following line,

Box mybox1 = new Box(10, 20, 15);

the values 10, 20, and 15 are passed to the **Box( )** constructor when **new** creates the object. Thus,

mybox1's copy of **width**, **height**, and **depth** will contain the values 10, 20, and 15, respectively.

Difference between constructor and method:

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

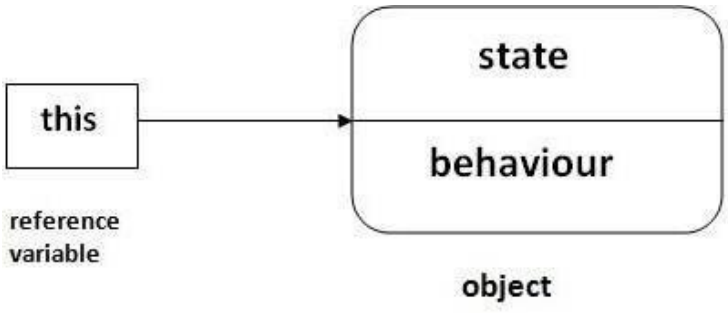
Constructor	Method
Constructor is used to initialize the state of anobject.	Method is used to expose behaviour of anobject.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in anycase.
Constructor name must be same as the classname.	Method name may or may not be same asclass name.

"this" KEYWORD:

Definition:

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

- **Usage of this keyword**
  1. this keyword can be used to refer current class instance variable.
  2. this() can be used to invoke current class constructor.
  3. this keyword can be used to invoke current class method (implicitly)
  4. this can be passed as an argument in the method call.
  5. this can be passed as argument in the constructor call.
  6. this keyword can also be used to return the current class instance.



Instance Variable Hiding:

It is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.

We can also have local variables, which overlap with the names of the class' instance variables.

However, **when a local variable has the same name as an instance variable, the local variable *hides* the instance variable.**
We can use **"this"** keyword to to resolve any namespace collisions that might occur betweeninstance variables and local variables.

Example:

```
class Student
{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee)
{
        this.rollno=rollno;
        this.name=name;
        this.fee=fee;
}
void display()
{
        System.out.println(rollno+" "+name+" "+fee);
}
}

class TestThis2
{
public static void main(String args[])
{
        Student s1=new Student(111,"ankit",5000f);
        Student s2=new Student(112,"sumit",6000f);
        s1.display();
        s2.display();
}
}
```

Output:

```
ankit 5000
sumit 6000
```

CONSTRUCTOR OVERLOADING:

Definition:

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Example of Constructor Overloading:

```
class Box
{
        double  width;
        double height;
        double depth;

        // constructor used when all the dimensions are specified
        Box(double w, double h, double d)
        {
                width=w;
                height=h;
                depth=d;
        }

        // constructor used when no dimensions are specified
        Box()
        {
                width=-1;
                height=-1;
                depth=-1;
        }

       // constructor used when cube is created
       Box(double len)
       {
                width = height = depth = len;
        }

         // Compute and return volume
     double volume()
        {
                return width*height*depth;
        }
  }
```

```
class ConsOverloadDemo
{
public static void main(String arg[])
{
    // declare, allocate and initialize Box objects
    Box mybox1=new Box(10,20,15);
    Box mybox2=new Box();
    Box mybox3=new Box(7);
    double vol;

    // Get volume of first box
    vol=mybox1.volume();
    System.out.println("Volume of Box1 is "+vol);

    // Get volume of second box
    vol=mybox2.volume();
    System.out.println("Volume of Box2 is "+vol);

    // Get volume of cube
    vol=mybox2.volume();
    System.out.println("Volume of Cube is "+vol);
}
}
```

Output:

Volume of Box1 is 3000.0
Volume of Box2 is -1.0
Volume of the cube is 343.0

As we can see, the proper overloaded constructor is called based upon the parameters specifiedwhen **new** is executed.

CONSTRUCTOR CHAINING:

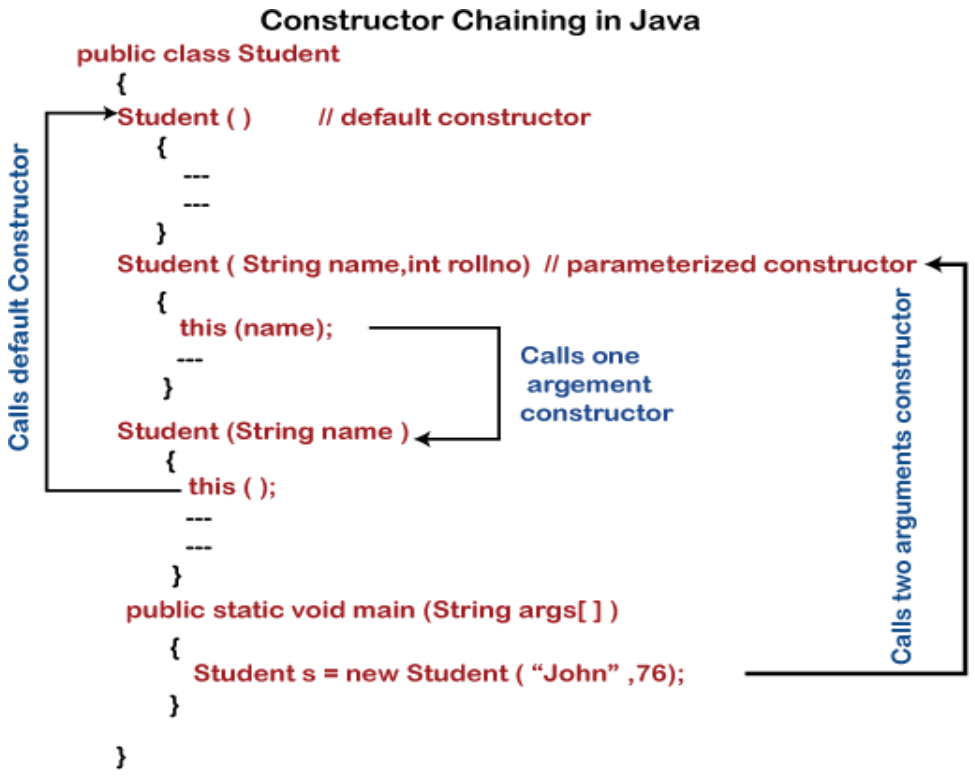
Constructor chaining is the process of calling one constructor of a class from another constructor of the same class or another class using the current object of the class.

- It occurs through inheritance.

Ways to achieve Constructor Chaining:

We can achieve constructor chaining in two ways:

- **Within the same class:** If we want to call the constructor from the same class, then we use **this** keyword.
- **From the base class:** If we want to call the constructor that belongs to different classes (parent and child classes), we use the **super** keyword to call the constructor from the base class.



Rules of Constructor Chaining:

- ✓ An expression that uses **this** keyword must be the first line of the constructor.
- ✓ Order does not matter in constructor chaining.
- ✓ There must exist at least one constructor that does not use this

Advantage:

- ✓ Avoids duplicate code while having multiple constructors.
- ✓ Makes code more readable

Example

```
class Shape
{
  int radius,length,breadth;

  Shape(int radius)
  {
    this.radius=radius;
  }
```

```
    Shape(int r,int l,int b)
    {
       this(r);
       length=l;
       breadth=b;
    }

    void areaCircle()
    {
       System.out.println("Area of Circle is "+(3.14*radius*radius));
    }
    void areaRectangle()
    {
       System.out.println("Area of Rectangle is "+(length*breadth));
    }
}
public class ConstructorChaining
{
    public static void main(String arg[])
    {
    Shape s1=new Shape(5,10,50);
    s1.areaCircle();
    s1.areaRectangle();
    }
}
```

Output:

Area of Circle is 78.5
Area of Rectangle is 500

1.13: ACCESS SPECIFIERS

Definition:

Access specifiers are used to specify the visibility and accessibility of a class constructors, member variables and methods.

Java classes, fields, constructors and methods can have one of four different accessmodifiers:

1. Public
2. Private
3. Protected
4. Default (package)

1. **Public (anything declared as public can be accessed from anywhere):**
   A variable or method declared/defined with the public modifier can be accessed anywhere in the program through its class objects, through its subclass objects andthrough the objects of classes of other packages also.

2. **Private (anything declared as private can't be seen outside of the class):**
   The instance variable or instance methods declared/initialized as private can be accessedonly by its class. Even its subclass is not able to access the private members.

3. **Protected (anything declared as protected can be accessed by classes in the samepackage and subclasses in the other packages):**
   The protected access specifier makes the instance variables and instance methods visibleto all the classes, subclasses of that package and subclasses of other packages.

4. **Default (can be accessed only by the classes in the same package):**
   The default access modifier is friendly. This is similar to public modifier except only theclasses belonging to a particular package knows the variables and methods.

	PRIVATE	DEFAULT	PROTECTED	PUBLIC
Same class	Yes	Yes	Yes	Yes
Same package Subclass	No	Yes	Yes	Yes
Same package Non-subclass	No	Yes	Yes	Yes
Different package Subclass	No	No	Yes	Yes
Different package Non-subclass	No	No	No	Yes

Example: Illustrating the visibility of access specifiers:

Z:\MyPack\FirstClass.java

```
package MyPack;

public class FirstClass
{
   public String i="I am public variable";
   protected String j="I am protected variable";
```

```
    private String k="I am private variable";
    String r="I dont have any modifier";
}
```

Z:\MyPack2\SecondClass.java

```
package MyPack2;
import MyPack.FirstClass;
class SecondClass extends FirstClass {
    void method()
    {
        System.out.println(i);     // No Error: Will print "I am public variable".
       System.out.println(j);     // No Error: Will print "I am protected variable".
       System.out.println(k);   // Error: k has private access in FirstClass
       System.out.println(r);   // Error: r is not public in FirstClass; cannot be accessed
                                  //            from outside package
    }

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

Output:

```
I am public variable
I am protected variable

Exception in thread "main" java.lang.RuntimeException: Uncompilable  source code - k
has private access in MyPack.FirstClass
```

1.14: "static" MEMBERS:

Static Members are data members (variables) or methods that belong to a static or non-static class rather than to the objects of the class. Hence it is not necessary to create object of that class to invoke static members.

- ✓ The static can be:
    1. variable (also known as class variable)

2. method (also known as class method)
3. block
4. nested class

❖ **Static Variable:**

- ✓ **When a member variable is declared with the static keyword, then it is called static variable and it can be accessed before any objects of its class are created, and without reference to any object.**
- ✓ **Syntax** to declare a static variable:

  [access_spefier] static data_type instance_variable;
- ✓ When a static variable is loaded in memory (static pool) it creates only a single copy of static variable and shared among all the objects of the class.
- ✓ A static variable can be accessed outside of its class directly by the class name anddoesn‘t need any object.

  Syntax : <class-name>.<variable-name>

Advantages of static variable

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

❖ **Static Method:**

If a method is declared with the static keyword , then it is known as static method.

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

o **Syntax: (defining static method)**

```
[access_specifier] static Return_type method_name(parameter_list)
{
  // method body
}
```

o Syntax to access static method:

```
<class-name>.<method-name>
```

- ✓ The most common example of a **static** member is **main( )**. **main( )** is declared as **static** because it must be called before any objects exist.

- Methods declared as **static** have several restrictions:
  - ✓ They can only directly call other **static** methods.
  - ✓ They can only directly access **static** data.
  - ✓ They cannot refer to **this** or **super** in any way.

❖ **Static Block:**

Static block is used to initialize the static data member like constructors helps toinitialize instance members and it gets executed exactly once, when the class is first loaded.

- It is executed before main method at the time of class loading in JVM.

- **Syntax:**

```
class classname
{
   static
   {
      // block of statements
   }
}
```

The following example shows a class that has a **static** method, some **static** variables, and a **static**
initialization block:

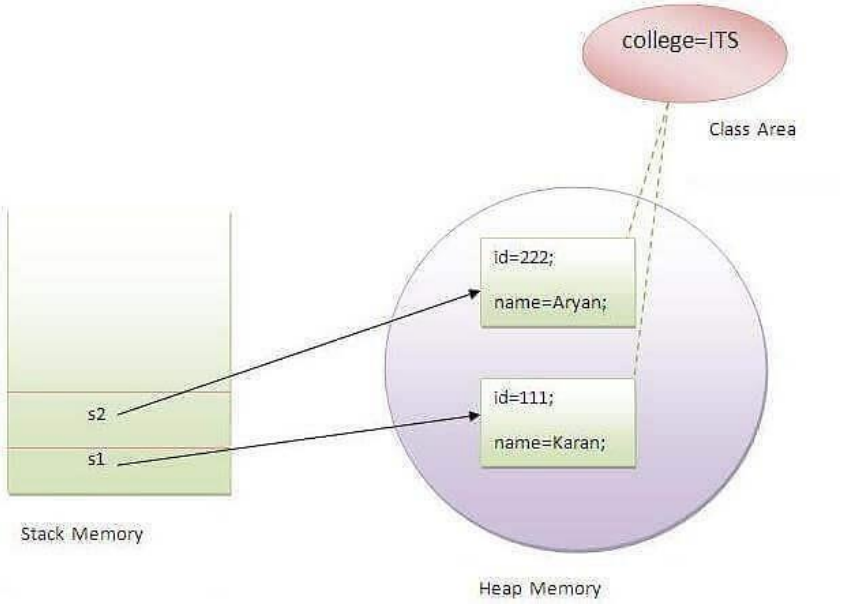
// Demonstrate static variables, methods, and blocks.

```
1.  class Student
2.  {
3.  int rollno;
4.  String name;
5.  static String college = "ITS";
6.  //static method to change the value of static variable
7.  static void change(){
8.  college = "BBDIT";
9.  }
10. //constructor to initialize the variable
11. Student(int r, String n){
12. rollno = r;
13. name = n;
14. }
15. //method to display values
16. void display()
17. {
18. System.out.println(rollno+" "+name+" "+college);
19. }
20. }
21. //Test class to create and display the values of object
22. public class TestStaticMembers
```

```
23.         {
24.         static
25.         {
26.         System.out.println(—*** STATIC MEMBERS – DEMO ***||);
27.         }
28.
29.         public static void main(String args[])
30.         {
31.         Student.change(); //calling change method
32.         //creating objects
33.         Student s1 = new Student(111,"Karan");
34.         Student s2 = new Student(222,"Aryan");
35.         Student s3 = new Student(333,"Sonoo");
36.         //calling display method
37.         s1.display();
38.         s2.display();
39.         s3.display();
40.         }
41.         }
```

Here is the output of this program:

```
*** STATIC MEMBERS – DEMO ***
111         Karan           BBDIT
222         Aryan           BBDIT
333         Sonoo           BBDIT
```



1.15 : JavaDoc Comments

Definition:

Javadoc is a tool which comes with JDK and it is used for generating Java code documentation in HTML format from Java source code. Java documentation can be created as part of the source code.

Input: Java source files (.java)

- Individual source files
- Root directory of the source files

Output: HTML files documenting specification of java code

- One file for each class defined
- Package and overview files

HOW TO INSERT COMMENTS?

The javadoc utility extracts information for the following items:

- Packages
- Public classes and interfaces
- Public and protected methods
- Public and protected fields

Each comment is placed immediately above the feature it describes.

Format:

- ✓ A Javadoc comment precedes similar to a multi-line comment except that it beginswith a forward slash followed by two asterisks (/**) and ends with a */
- ✓ Each /** . . . */ documentation comment contains free-form text followed by tags.
- ✓ A tag starts with an @, such as @author or @param.
- ✓ The first sentence of the free-form text should be a summary statement.
- ✓ The javadoc utility automatically generates summary pages that extract thesesentences.
- ✓ In the free-form text, you can use HTML modifiers such as ... foremphasis, <code>...</code> for a monospaced —typewriter font, ... for strong emphasis, and even <img ...> to include an image.
- ✓ Example:

```
/**
This is a <b>doc</b> comment.
*/
```

TYPES OF COMMENTS:

1. ***Class Comments***

The class comment must be placed *after* any import statements, directly before the classdefinition.

Example:

```
import java.io.*;
/** class comments should be written here */Public class sample
{
....
}
```

2. ***Method Comments***

The method comments must be placed immediately before the method that it describes.

Tags used:

Tag	Description	Syntax
@param	It describes the method parameter	**@param name description**
@return	This tag describes the return value from a method with the exception void methods and constructors.	**@return description**
@throws	This tag describes the method that throws an exception.	**@throws class description**

Example:

```
/** adding two numbers
@param a & b are two numbers to be added
@return the result of addition
**/
public double add(int a,int b)
{
int c=a+b;
return c;
}
```

3. ***Field Comments***

Field comments are used to document public fields—generally that means static constants.

For example:

```
/**
* Account number
*/
public static final int acc_no = 101;
```

4. ***General Comments***

Tag	Description	Syntax
The following tags can be used in class documentation comments		
@author	This tag makes an —author entry. You can have multiple @author tags, one for each author.	**@author name**
@version	This tag makes a —version‖ entry. The text can be any description of the current version.	**@version text**
The following tags can be used in all documentation comments		
@since	This tag makes a —since‖ entry. The text can be any description of the version that introduced thisfeature. For example, @since version 1.7.1	**@since text**
@deprecated	This tag adds a comment that the class, method, or variable should no longer be used. The text should suggest a replacement. For example: @deprecated Use <code>setVisible(true)</code>instead	**@deprecated text**
Hyperlinks to other relevant parts of the javadoc documentation, or to external documents,with the @see and @link tags.		
@link	This tag place hyperlinks to other classes or methods anywhere in any of your documentationcomments.	**{@link package.class#feature label}**

@see	This tag adds a hyperlink in the —see also section. It can be used with both classes and methods. Here, reference can be one of the following: package.class#feature label label "text" Example: @see —Core java 2 @see Core Java	**@see reference**

COMMENT EXTRACTION

Here, *docDirectory* is the name of the directory where you want the HTML files to go.

Follow these steps:

1. Change to the directory that contains the source files you want to document.
2. To create the document API, you need to use the javadoc tool followed by java file name.There is no need to compile the javafile.

Here, docDirectory is the name of the directory where you want the HTML files to go.

Follow these steps:

1. Change to the directory that contains the source files you want to document.
2. Run the command

javadoc -d docDirectory nameOfPackage

for a single package. Or run

javadoc -d docDirectory nameOfPackage1 nameOfPackage2...

to document multiple packages.

If your files are in the default package, then instead run

javadoc -d docDirectory *.java

If you omit the -d docDirectory option, then the HTML files are extracted to the currentdirectory.

```
//Java program to illustrate frequently used
// Comment tags

/**
* <h1>Find average of three numbers!</h1>
* The FindAvg program implements an application that
* simply calculates average of three integers and Prints
* the output on the screen.
*
* @author  Pratik Agarwal
* @version 1.0
* @since  2017-02-18
*/
public class FindAvg
{
   /**
   * This method is used to find average of three integers.
   * @param numA This is the first parameter to findAvg method
   * @param numB  This is the second parameter to findAvg method
   * @param numC  This is the third parameter to findAvg method
   * @return int This returns average of numA, numB and numC.
   */
   public int findAvg(int numA, int numB, int numC)
   {
      return (numA + numB + numC)/3;
   }

   /**
   * This is the main method which makes use of findAvg method.
   * @param args Unused.
   * @return Nothing.
   */

   public static void main(String args[])
   {
      FindAvg obj = new FindAvg();
      int avg = obj.findAvg(10, 20, 30);

      System.out.println("Average of 10, 20 and 30 is :" + avg);
   }
}
```

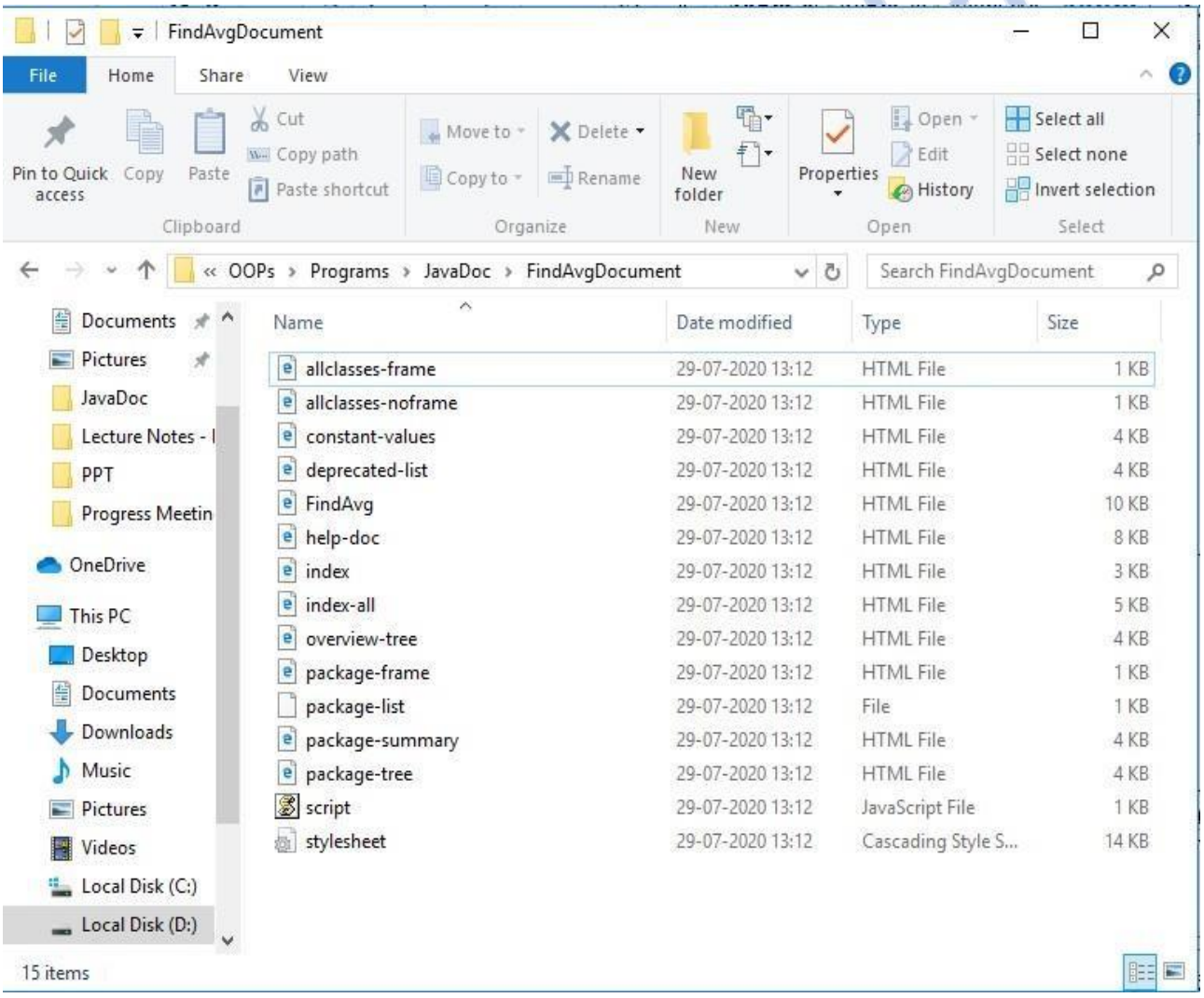
Example:

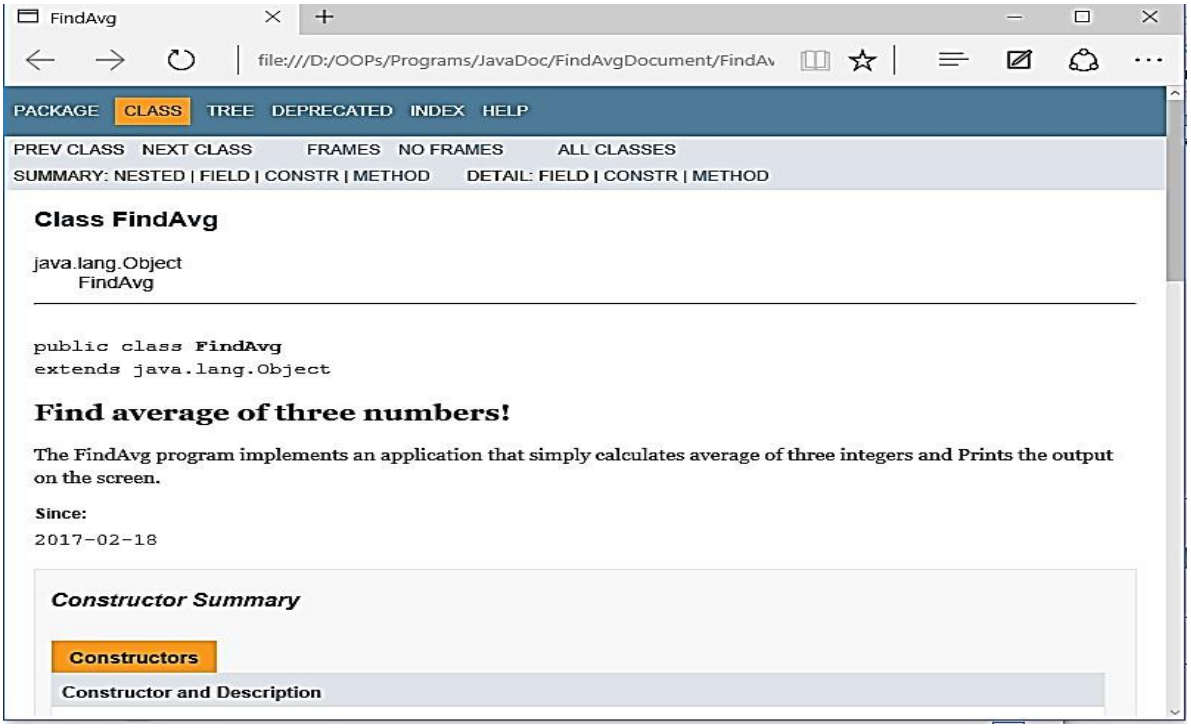
OUTPUT:

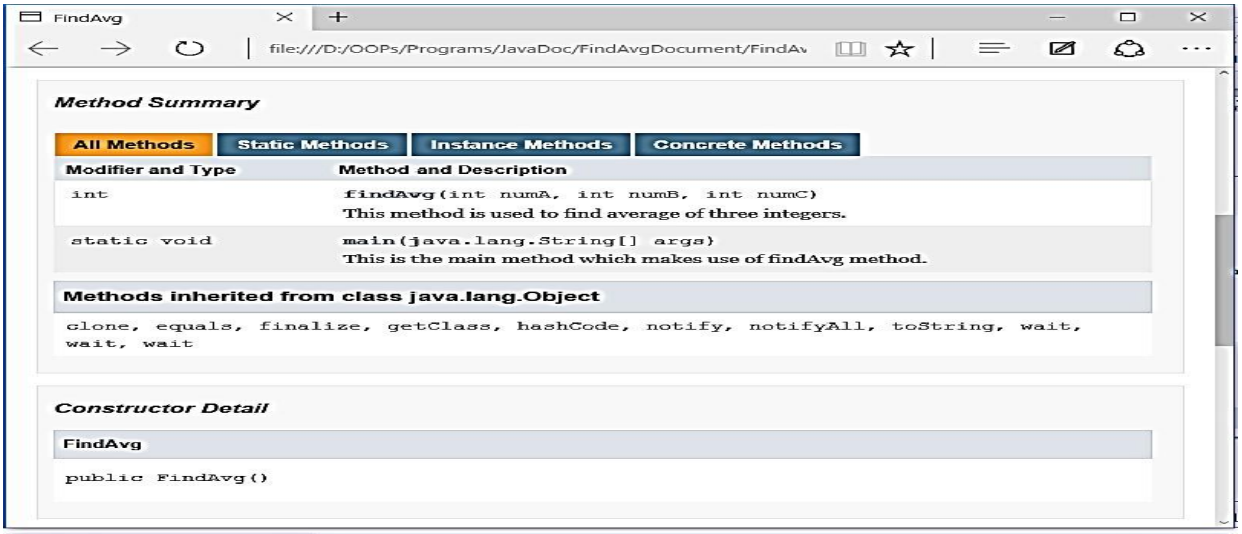
D:\OOPs\Programs\JavaDoc>javadoc -d FindAvgDocument FindAvg.java

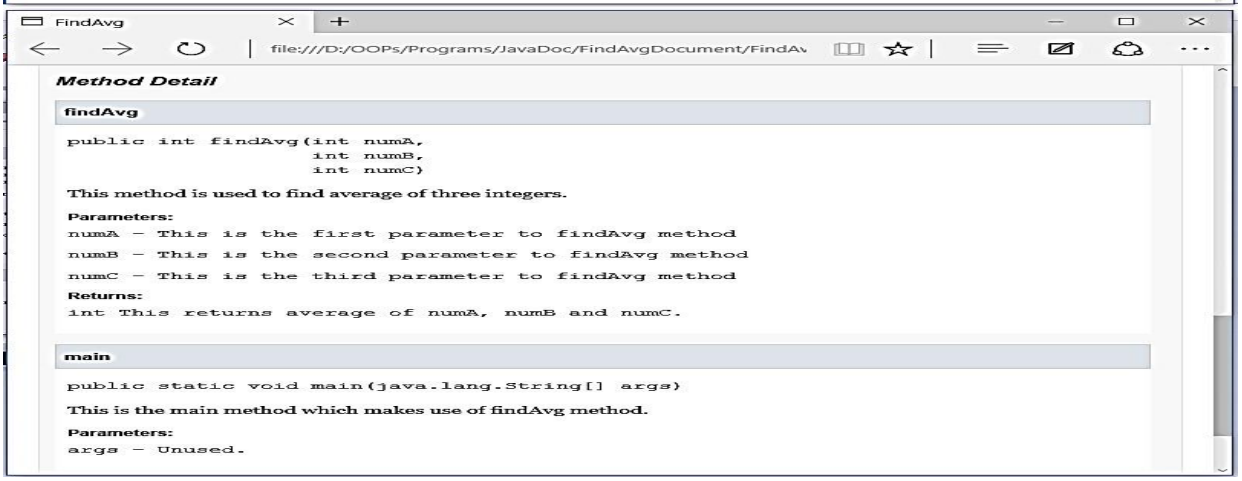
Loading source file FindAvg.java...
Constructing Javadoc information...
Creating destination directory: "FindAvgDocument\"
Standard Doclet version 1.8.0_251

```
Building tree for all the packages and classes...
Generating  FindAvgDocument\FindAvg.html...
FindAvg.java:32: error: invalid use of @return
    * @return Nothing.
      ^
Generating FindAvgDocument\package-frame.html...
Generating FindAvgDocument\package-summary.html...
Generating FindAvgDocument\package-tree.html...
Generating FindAvgDocument\constant-values.html...
Building index for all the packages and classes...
Generating FindAvgDocument\overview-tree.html...
Generating FindAvgDocument\index-all.html...
Generating FindAvgDocument\deprecated-list.html...
Building index for all classes...
Generating FindAvgDocument\allclasses-frame.html...
Generating FindAvgDocument\allclasses-noframe.html...
Generating FindAvgDocument\index.html...
Generating FindAvgDocument\help-doc.html...
1 error
```




FindAvg
file:///D:/OOPs/Programs/JavaDoc/FindAvgDocument/FindAv
PACKAGE CLASS TREE DEPRECATED INDEX HELP
PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD
Class FindAvg
java.lang.Object
FindAvg
public class FindAvg
extends java.lang.Object
Find average of three numbers!
The FindAvg program implements an application that simply calculates average of three integers and Prints the output on the screen.
Since:
2017-02-18
Constructor Summary
Constructors
Constructor and Description


FindAvg
file:///D:/OOPs/Programs/JavaDoc/FindAvgDocument/FindAv
Method Summary
All Methods Static Methods Instance Methods Concrete Methods
Modifier and Type Method and Description
int findAvg(int numA, int numB, int numC)
This method is used to find average of three integers.
static void main(java.lang.String[] args)
This is the main method which makes use of findAvg method.
Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait
Constructor Detail
FindAvg
public FindAvg()


FindAvg
file:///D:/OOPs/Programs/JavaDoc/FindAvgDocument/FindAv
Method Detail
findAvg
public int findAvg(int numA,
int numB,
int numC)
This method is used to find average of three integers.
Parameters:
numA - This is the first parameter to findAvg method
numB - This is the second parameter to findAvg method
numC - This is the third parameter to findAvg method
Returns:
int This returns average of numA, numB and numC.
main
public static void main(java.lang.String[] args)
This is the main method which makes use of findAvg method.
Parameters:
args - Unused.

1.16 : Additional Topics
Comments, Literals, Keywords, Type Conversion, Garbage Collection, Command Line Arguments

1.16.1 : JAVA – COMMENTS

- Java comments are either explanations of the source code or descriptions of classes,interfaces, methods, and fields.
- They are usually a couple of lines written above or beside Java code to clarify what itdoes.
- Comments in Java do not show up in the executable program.
- The Java language supports three kinds of comments:

1. Line comment:

- ✓ When you want to make a one line comment type "//" and follow the two forward slasheswith your comment.
- ✓ **Syntax:** // text
- ✓ **Example:** // this is a single line comment
- ✓ The compiler ignores everything from // to the end of the line.

2. Block Comment:

- ✓ To start a block comment type "/*". Everything between the forward slash and asterisk, even if it's on a different line, will be treated as comment until the characters "*/" end the comment.
- ✓ **Syntax:** /* *text* */
- ✓ **Example:** /* it is a comment */ (or)

```
/* thisis a block
comment
*/
```

- ✓ The compiler ignores everything from /* to */.

3. Documentation Comment:

- ✓ This type of comment helps is generating the documentation automatically.
- ✓ **Syntax:** /** *documentation* */

The JDK javadoc tool uses doc comments when preparing automatically generated documentation. For more information on javadoc, see the Java tool documentation.

- ✓ **Example:**

```
/*
* Title: Conversion of Degrees
* Aim: To convert Celsius to Fahrenheit and vice versa
* Date: 31/08/2000
* Author: Tim
*/
```

1.16.2 : JAVA - CONSTANTS

- ✓ A constant is an identifier written in uppercase (convention and not a rule) that prevents
  its contents form being modified by the program during the execution.
- ✓ If an attempt is made to change the value, the compiler will give an error message.
- ✓ In Java, the keyword **final** is used to declare constants.
- ✓ The value of a final variable cannot change after it has been initialized.

final datatype variablename=value;

- ✓ **Syntax:**
- ✓ **Example:**final float PI=3.14f;

1.16.3 : JAVA - IDENTIFIERS

- ✓ Identifiers are names given to the variables, classes, methods, objects, labels, package and interface in our program.
- ✓ The name we are giving must be meaningful and it may have random length.
- ✓ The following rule must be followed while giving a name:
  1. The first character must not begin with a number.
  2. The identifier is formed with alphabets, number, dollar sign ($) and underscore (_).
  3. It should not be a reserved word.
  4. Space is not allowed in between the identifier name.

- ✓ **Example:**

```
String name = "Homer Jay Simpson";
int weight = 300;
double height = 6;
```

1.16.4 : JAVA – RESERVED WORDS (KEYWORDS)

- ✓ There are some words that you cannot use as object or variable names in a Java program. These words are known as reserved words; they are keywords that are already used by the syntax of the Java programming language.
- ✓ For example, if you try and create a new class and name it using a reserved word:

```
// you can't use finally as it's a reserved word!
        class finally {
        public static void main(String[] args)
        {
```

```
//class code..
}
}
```

- ✓ It will not compile, instead you will get the following error: <identifier> expected
- ✓ The table below lists all the words that are reserved:

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

1.16.5 : TYPE CONVERSIONS AND CASTING

Type Conversion is the task of converting one data type into another data type.

Two types of type conversion:

1. Implicit Type Conversion (or) Automatic Conversion
2. Explicit Type Conversion (or) Casitng

1. Implicit Type Conversion (or) Automatic Conversion;

If the two types are compatible, then Java will perform the conversion automatically. When one type of data is assigned to another type of variable, an ***Automatic type conversion (or) WideningConversion*** will take place if the following two conditions are met:

- Two types are compatible
- The destination type is larger than the source type

.

Example:

```
byte a=100;
int b=a;        // b is larger than a

long d=b;       // d is large than b
float e=b;      // e is larger than b

float sum=10;
int s=sum;      // s is smaller than sum, So we need to go for explicit conversion.
```

1. Explicit Type Conversion (or) Casting:

If the two types are compatible, a forced conversion of one type into another type is performed This forced conversion is called as Explicit Type Conversion. **Casting (or) narrowing conversion** is an operation which performs an explicit conversion between incompatible types.

Example: converting int to byte.

Syntax to perform "Cast":

(target-type) value;

Here,

Target-type = specifies the desired type to convert the specified value.

Example:

```
class conversion {
public static void main(String arg[])
{
byte b;
int i=257;
double d=323.142;

System.out.println("\nConversion of int to byte: ");
b=(byte) i;
System.out.println("i and b : "+i+" , "+b);

System.out.println("\nConversion of double to int: ");
i=(int) d;
System.out.println("d and i : "+d+" , "+i);

System.out.println("\nConversion of double to byte: ");
b=(byte) d;
```

```
System.out.println("d and b : "+d+" , "+b);

        // Automatic Type promotions in expressions
byte   r=40;
byte   s=50;
byte t=100;
int p=r * s / t;        // r*s exceeds the range of byte, so automatic type promotion take place.
System.out.println("Value of P = "+p);
s=s*2;          //Error! cannot assign int to a byte.
s=(byte)(s*2);          // Possible.
}
}
```

Output:

```
Conversion of int to byte:
i and b : 257 , 1
Conversion of int to byte:
d and i : 323.142 , 323
Conversion of int to byte:
d and b : 323.142 , 67
Value of P = 20
```

Type Promotions rules:

1. All **byte, short** and **char** values are promoted to int.
2. If one operand is **long,** the whole expression is promoted to **long.**
3. If one operand is **float,** the whole expression is promoted to **float.**
4. If any of the operand is **double,** the result is **double.**

1.16.6: GARBAGE COLLECTION

- ✓ Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation.
- ✓ In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator.

- ✓ Java takes a different approach;

Automatic Garbage Collection: The technique that accomplishes automatic deallocation ofmemory occupied by an unused object is called *garbage collection*.

It works like this:

- When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++.
- Garbage collection only occurs sporadically (if at all) during the execution of your program.

- **Finalization:**
  - ✓ Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed.
  - ✓ To handle such situations, Java provides a mechanism called *finalization*. By using finalization, you can define specific actions that will occur when an object is just about tobe reclaimed by the garbage collector.

- **Finalize() method:**

A **finalize()** method is a method that will be called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

Inside the **finalize( )** method, we will specify those actions that must be performed before anobject is destroyed.

The finalize( ) method has this general form:

```
protected void finalize( )
{
// finalization code here
}
```

Here, the keyword protected is a specifier that prevents access to finalize( ) by code definedoutside its class.

Example:

```
public class TestGarbage1
{
public void finalize()
{
```

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

Output:

object is garbage collectedobject is garbage collected

1.16.7: USING COMMAND LINE ARGUMENTS:

- ✓ Sometimes you will want to pass information into a program when you run it. This isaccomplished by passing *command-line arguments* to **main( )**.

- ✓ **A command-line argument is the information passed to the main() method that directly follows the program's name on the command line when it is executed.**

- ✓ To access the command-line arguments inside a Java program is quite easy—they are stored as strings in a **String** array passed to the **args** parameter of **main( )**.
- ✓ The first command-line argument is stored at **args[0]**, the second at **args[1]**, and so on.
- ✓ For example, the following program displays all of the command-line arguments that it iscalled with:

```
// Display all command-line arguments.

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

Try executing this program, as shown here:

>java CommandLine this is a test 100 -1

When you do, you will see the following output:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

Unit – 2: INHERITANCE, PACKAGES AND INTERFACES

UNIT II 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

Method Overloading is a feature in Java that allows a class to have **more than one methods having same name**, but with **different signatures** (Each method must have different number of parameters or parameters having different types and orders).

Advantage:

- ✓ Method Overloading increases the readability of the program.
- ✓ Provides the flexibility to use similar method with different parameters.

Three ways to overload a method

In order to overload a method, the argument lists of the methods must differ in either of these:

1. Number of parameters. (Different number of parameters in argument list)

For example: This is a valid case of overloading

add(int, int)
add(int, int, int)

2. Data type of parameters. (Difference in data type of parameters)

For example:

add(int, int)
add(int, float)

3. Sequence of Data type of parameters.

For example:

add(int, float)
add(float, int)

Rules for Method Overloading:

1. First and important rule to overload a method in java is to change method signature.
2. Return type of method is never part of method signature, so only changing the return type of method does not amount to method overloading.

Example: To find the Minimum of given numbers:

```
public class OverloadingCalculation1
{
  public static void main(String[] args)
 {
    int a = 11;
    int b = 6;
    int c = 3;
    double x = 7.3;
    double y = 9.4;

    int result1 = minFunction(a, b, c);
    double result2 = minFunction(x, y);
    double result3 = minFunction(a, x);

    System.out.println("Minimum("+a+","+b+","+c+") = " + result1);
    System.out.println("Minimum("+x+","+y+") = " + result2);
    System.out.println("Minimum("+a+","+x+") = " + result3);
  }

  public static int minFunction(int n1, int n2, int n3)
  {
    int min;
    int temp = n1<n2? n1 : n2;
    min = n3 < temp? n3 : temp;
    return min;
  }

  public static double minFunction(double n1, double n2)
  {
   double min;
```

```
    if (n1 > n2)
      min = n2;
    else
      min = n1;

    return min;
  }
public static double minFunction(int n1, double n2)
  {
    double min;
     if (n1 > n2)
       min = n2;
    else
      min = n1;

    return min;
  }
}
```

This would produce the following result:

```
Minimum(11,6,3) = 3
Minimum(7.3,9.4) = 7.3
Minimum(11,7.3) = 7.3
```

Note:-

Method overloading is not possible by changing the return type of the method because of ambiguity that may arise while calling the method with same parameter list with different return type.

Example:

```
class Add
{
   static int sum(int a, int b)
  {
      return a+b;
  }
  static float sum(int a, int b)
```

```
 {
    return a+b;
 }
public static void main(String arg[])
{
   System.out.println(sum(10,20));
   System.out.println(sum(15,25));
}
}
```

Output:

Compile by: javac TestOverloading3.java

Add.java:7: error: method sum(int,int) is already defined in class Add
static float sum(int a, int b)
^
1 error

Method Overloading and Type Promotion

Type Promotion: When a data type of smaller size is promoted to the data type of bigger size than this is called type promotion, for example: byte data type can be promoted to short, a short data type can be promoted to int, long, double etc.

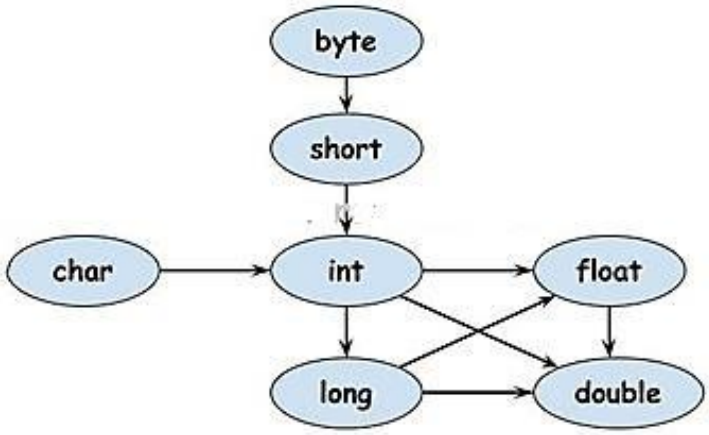
Type Promotion in Method Overloading:

One type is promoted to another implicitly if no matching datatype is found.

Type Promotion Table:

The **data type on the left side can be promoted to the any of the data type present in the right side of it**.

```
byte → short → int → long → double
short → int → long → float → double
int → long → float → double
float → double
long → float → double
char → int → long → float → double
```



Example: Method Overloading with Type Promotion:

```
class Overloading
{
     void sum(int a, float b)
   {
       System.out.println(a+b);
   }
     void sum(int a, int b, int c)
   {
       System.out.println(a+b+c);
   }

  public static void main(String args[])
{
        OverloadingCalculation1 obj=new OverloadingCalculation1();
        obj.sum(20,20);          //now second int literal will be promoted to float
        obj.sum(100,'A');        //Character literal will be promoted to float
        obj.sum(20,20,20);
 }
}
```

OUTPUT:

```
40.0
165.0
60
```

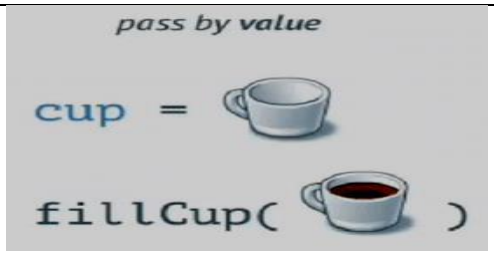
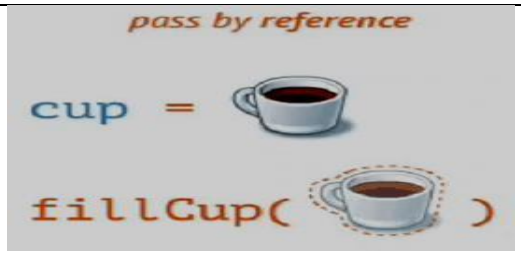
2.2: Objects as Parameters

Java is strictly pass-by-value. But the scenario may change when the parameter passed is of primitive type or reference type.

- If we pass a primitive type to a method, then it is called **pass-by-value** or call-by-value.
- If we pass an object to a method, then it is called **pass-by-reference** or call-by-reference.

Object as a parameter is a way to establish communication between two or more objects of the same class or different class as well.

Pass-by-value vs. Pass-by-reference:

Pass-by-value (Value as parameter)	Pass-by-reference (Object as parameter)
Only values are passes to the function parameters. So any modifications done in the formal parameter will not affect the value of actual parameter	Reference to the object is passed. So any modifications done through the object will affect the actual object.
Caller and Callee method will have two independent variables with same value.	Caller and Callee methods use the same reference for the object.
Callee method will not have any access to the actual parameter	Callee method will have the direct reference to the actual object
Requires more memory	Requires less memory
	

class CallByVal
{
void Increment(int count)
{
count=count+10;
}
}
public class CallByValueDemo {
public static void main(String arg[])
{
CallByVal ob1=new CallByVal();
int count=100;
System.out.println("Value of Count before method call = "+count);
	class CallByRef
	{
	int count=0;
	CallByRef(int c)
	{
	count=c;
	}
	static void Increment(CallByRef obj) {
	obj.count=obj.count+10;
	}
	
	public static void main(String arg[]) {
	CallByRef ob1=new CallByRef(10);
	System.out.println("Value of Count (Object 1) before

```
    ob1.Increment(count);
    System.out.println("Value of Count after
                        method call = "+count);
  }
}
```

OUTPUT:

```
Value of Count before method call = 100
Value of Count after method call = 100
```

```
                        method call = "+ob1.count);
    Increment(ob1);
    System.out.println("Value of Count (Object 1) after
                        method call = "+ob1.count);

  } }
```

OUTPUT:

```
Value of Count (Object 1) before method call = 10
Value of Count (Object 1) after method call = 20
```

Returning Objects:

In Java, a method can return any type of data. Return type may any primitive data type or class type (i.e. object). As a method takes objects as parameters, it can also return objects as return value.

Example:

```
class Add
{
  int num1,num2,sum;

  static Add calculateSum(Add a1,Add a2)
  {
    Add a3=new Add();
    a3.num1=a1.num1+a1.num2;
    a3.num2=a2.num1+a2.num2;
    a3.sum=a3.num1+a3.num2;
    return a3;
  }

  public static void main(String arg[])
  {
    Add ob1=new Add();
    ob1.num1=10;
    ob1.num2=15;

    Add ob2=new Add();
    ob2.num1=100;
    ob2.num2=150;

    Add ob3=calculateSum(ob1,ob2);
    System.out.println("Object 1 -> Sum = "+ob1.sum);
```

```
        System.out.println("Object 2 -> Sum = "+ob2.sum);
        System.out.println("Object 3 -> Sum = "+ob3.sum);
    }
}
```

OUTPUT:

Object 1 -> Sum = 0
Object 2 -> Sum = 0
Object 3 -> Sum = 275

2.3: STATIC, NESTED and INNER CLASSES

Definition:

> **An inner class is a class that is defined inside another class.**
> Inner classes let you make one class a member of another class. Just as classes have member variables and methods, a class can also have member classes.

Benefits:

1. Name control
2. Access control
3. Code becomes more readable and maintainable because it locally group related classes in one place.

Syntax: For declaring Inner classes

```
[modifier] class OuterClassName
{
      ---- Code -----
       [modifier] class InnerClassName
       {
       ---- Code ----
       }
}
```

- **Instantiating an Inner Class:**

  Two Methods:

  1. **Instantiating an Inner class from outside the outer class:**
     To instantiate an instance of an inner class, you must have an instance of the outer class.
     Syntax:
     OuterClass.InnerClass objectName=OuterObj.new InnerClass();

  2. **Instantiating an Inner Class from Within Code in the Outer Class:**
     From inside the outer class instance code, use the inner class name in the normal way:

  Syntax:

  InnerClassName obj=new InnerClassName();

Advantage of java inner classes

There are basically three advantages of inner classes in java. They are as follows:

1) Nested class **can access all the members (data members and methods) of outer class** including private.
2) Nested classes are used **to develop more readable and maintainable code**.
3) **Code Optimization**: It requires less code to write.

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)
  1. Member inner class
  2. Anonymous inner class
  3. Local inner class
- Static nested class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

1. Java Member inner class

A non-static class that is created inside a class but outside a method is called member inner class.

Syntax:

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

Java Member inner class example

In this example, we are creating msg() method in member inner class that is accessing the private data member of outer class.

```
1.  class TestMemberOuter1
2.  {
3.   private int data=30;
4.   class Inner
5.  {
6.    void msg()
7.  {
8.  System.out.println("data is "+data);
9.  }
10.  }
11.  public static void main(String args[])
12. {
13.   TestMemberOuter1 obj=new TestMemberOuter1();
14.   TestMemberOuter1.Inner in=obj.new Inner();
15.   in.msg();
16.  }
17. }
```

Output:

data is 30

2. Java Anonymous inner class

A class that have no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface. Java Anonymous inner class can be created by two ways:

1. Class (may be abstract or concrete).
2. Interface

Java anonymous inner class example using class

```
1.  abstract class Person
2.  {
3.    abstract void eat();
4.  }
5.  class TestAnonymousInner
6.  {
7.   public static void main(String args[])
8.  {
9.    Person p=new Person()
10. {
11.   void eat()
12. {
13.     System.out.println("nice fruits");
14. }
15.  };
16.  p.eat();
17.  }
18. }
```

Output:

nice fruits

Java anonymous inner class example using interface

```
1.  interface Eatable
2.  {
3.   void eat();
```

```
4.  }
5.  class TestAnnonymousInner1
6.  {
7.   public static void main(String args[])
8.  {
9.   Eatable e=new Eatable()
10.  {
11.  public void eat(){System.out.println("nice fruits");
12.  }
13.  };
14.  e.eat();
15.  }
16. }
```

Output:

```
nice fruits
```

3. Java Local inner class

A class i.e. created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Java local inner class example

```
1.  public class localInner1
2.  {
3.   private int data=30;//instance variable
4.   void display()
5.  {
6.  int value=50;
7.    class Local
8.  {
9.     void msg()
10.      {
11.       System.out.println(data);
12.       System.out.println(value);
13.      }
14.  }
```

```
15.    Local l=new Local();
16.    l.msg();
17.   }
18.   public static void main(String args[])
19.  {
20.    localInner1 obj=new localInner1();
21.    obj.display();
22.   }
23.  }
```

Output:

```
30
50
```

Rules for Java Local Inner class

1. Local inner class cannot be invoked from outside the method.
2. Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in local inner class.
3. Local variable can't be private, public or protected.

Properties:

1. Completely hidden from the outside world.
2. Cannot access the local variables of the method (in which they are defined), but the local variables has to be declared final to access.

4. Java static nested class

A static class i.e. created inside a class is called 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 outer class including private.
- Static nested class cannot access non-static (instance) data member or method.

Java static nested class example with instance method

```
1.    class TestOuter1
2.    {
3.      static int data=30;
4.      static class Inner
5.    {
6.       void msg()
```

```
7.      {
8.        System.out.println("data is "+data);
9.       }
10.     }
11.     public static void main(String args[])
12.    {
13.     TestOuter1.Inner obj=new TestOuter1.Inner();
14.     obj.msg();
15.     }
16.    }
```

Output:

```
data is 30
```

Java static nested class example with static method

If you have the static member inside static nested class, you don't need to create instance of static nested class.

```
1.  class TestOuter2{
2.   static int data=30;
3.   static class Inner
4.  {
5.    static void msg()
6.     {
7.      System.out.println("data is "+data);
8.     }
9.   }
10.  public static void main(String args[])
11. {
12.  TestOuter2.Inner.msg();//no need to create the instance of static nested
     class
13.  }
14. }
```

Output:

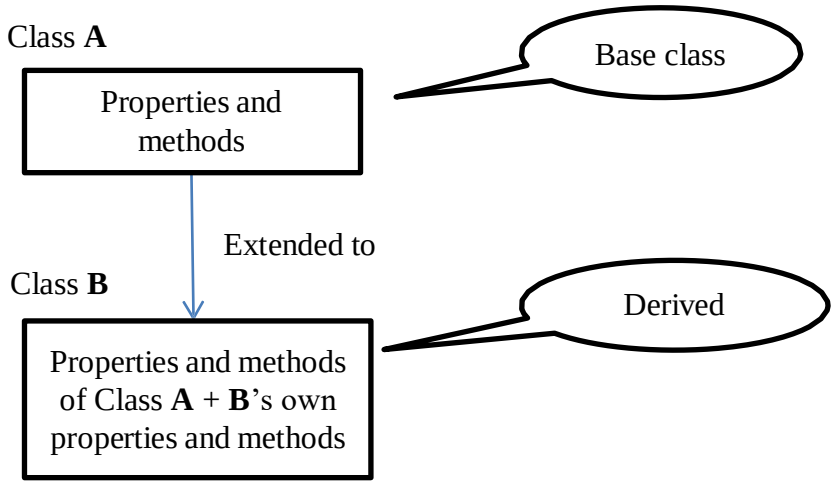
```
data is 30
```

2.4: Inheritance

Definition:

Inheritance is a process of deriving a new class from existing class, also called as "extending a class". When an existing class is extended, the new (inherited) class has all the properties and methods of the existing class and also possesses its own characteristics.

- ✓ The class whose property is being inherited by another class is called **"base class"** (or) **"parent class"** (or) **"super class"**.
- ✓ The class that inherits a particular property or a set of properties from the base class is called **"derived class"** (or) **"child class"** (or) **"sub class"**.



- ✓ Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

➢ **ADVANTAGES OF INHERITANCE:**

- **Reusability of Code:**
  - ✓ Inheritance is mainly used for code reusability (Code reusability means that we can add extra features to an existing class without modifying it).
- **Effort and Time Saving:**
  - ✓ The advantage of reusability saves the programmer time and effort. Since the main code written can be reused in various situations as needed.
- **Increased Reliability:**
  - ✓ The program with inheritance becomes more understandable and easily maintainable as the sub classes are created from the existing reliably working classes.

➢ **"extends" KEYWORD:**

- ✓ Inheriting a class means creating a new class as an extension of another class.
- ✓ The **extends** keyword is used to inherit a class from existing class.
- ✓ The general form of a **class** declaration that inherits a superclass is shown here:
- ✓ **Syntax:**

```
[access_specifier] class subclass_name extends superclass_name
{
        // body of class
}
```

Characteristics of Class Inheritance:

1. A class cannot be inherited from more than one base class. Java does not support
   the inheritance of multiple super classes into a single subclass.
2. Sub class can access only the non-private members of the super class.
3. Private data members of a super class are local only to that class. Therefore, they can't be accessed outside the super class, even sub classes can't access the private members.
4. Protected features in Java are visible to all subclasses as well as all other classes in the same package.

✓ **Example:**

```
class Vehicle
{
   String brand;
   String color;
}
class Car extends Vehicle
{
   int totalDoor;
}

class Bike extends Vehicle
{
}
```

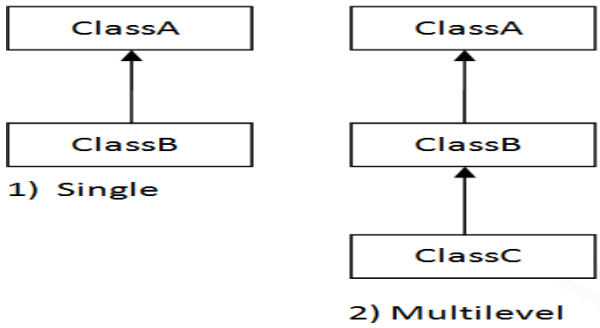
In the above example, Vehicle is the **super class** or base class that holds the common property of Car and Bike. Car and Bike is the **sub class** or derived class that inherits the property of class Vehicle **extends** is the keyword used to inherit a class.

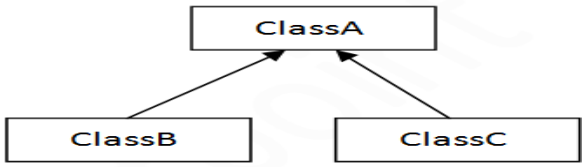
➢ TYPES OF INHERITACE:

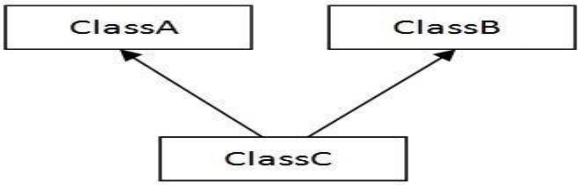
1. Single Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance

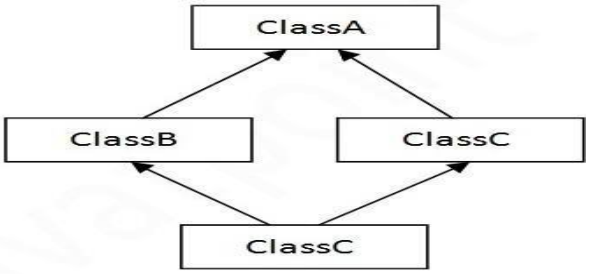
Note: The following inheritance types are not directly supported in Java.

4. Hierarchical Inheritance
5. Hybrid Inheritance

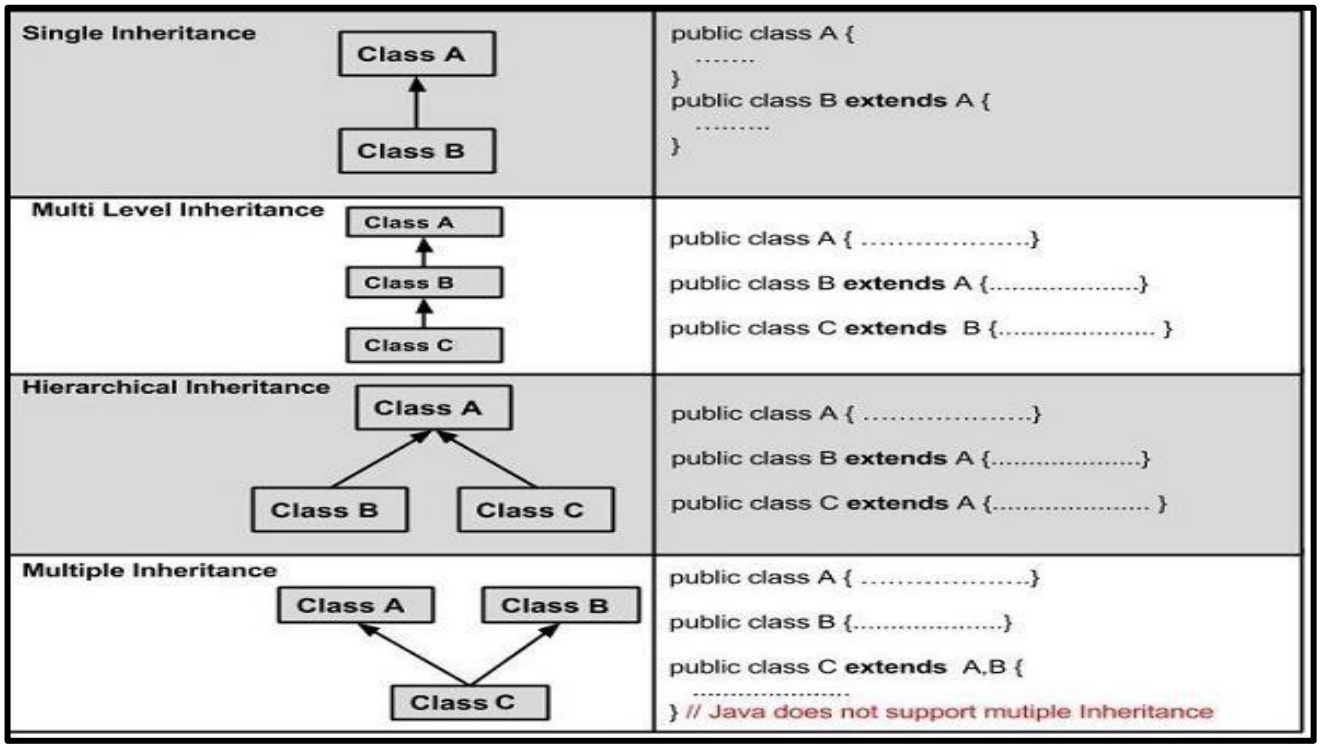








.



1. SINGLE INHERITANCE

The process of creating only one subclass from only one super class is known as **Single Inheritance.**

- ✓ Only two classes are involved in this inheritance.
- ✓ The subclass can access all the members of super class.

Example: Animal → Dog

```
1. class Animal
2. {
3.     void eat()
4.    {
5.          System.out.println("eating...");
6.     }
7. }
8. class Dog extends Animal
9. {
10.   void bark()
11.   {
12.       System.out.println("barking...");
13.   }
14.}
15.class TestInheritance
16.{
17.    public static void main(String args[])
18.    {
19.        Dog d=new Dog();
20.        d.bark();
21.        d.eat();
22.    }
23.}
```

Output:

```
$java TestInheritance
barking...
eating...
```

2. MULTILEVEL INHERITANCE:

- ✓ The process of creating a new sub class from an already inherited sub class is known as **Multilevel Inheritance**.
- ✓ Multiple classes are involved in inheritance, but one class extends only one.
- ✓ The lowermost subclass can make use of all its super classes' members.
- ✓ Multilevel inheritance is an indirect way of implementing multiple inheritance.
- ✓ **Example: Animal → Dog → BabyDog**

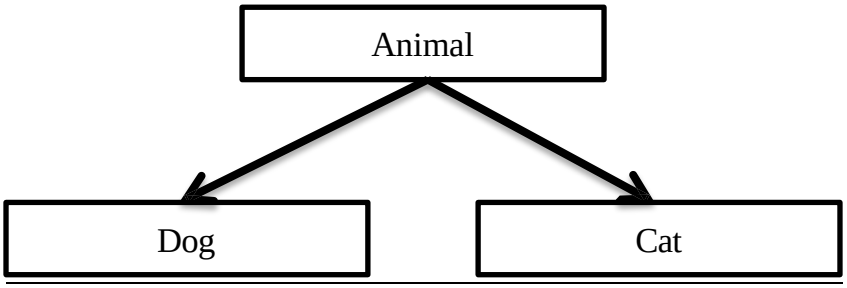
```
1.  class Animal
2.  {
3.    void eat()
4.    {
5.        System.out.println("eating...");
6.    }
7.  }
8.  class Dog extends Animal
9.  {
10.     void bark()
11.     {
12.       System.out.println("barking...");
13.     }
14.   }
15. class BabyDog extends Dog
16. {
17.    void weep()
18.    {
19.       System.out.println("weeping...");
20.    }
21. }
22. class TestInheritance2
23. {
24.   public static void main(String args[])  {
25.       BabyDog d=new BabyDog();
26.       d.weep();
27.       d.bark();
28.       d.eat();
29.   }
30. }
```

Output:

```
$java TestInheritance2
weeping...
barking...
eating..
```

3. HIERARCHICAL INHERITANCE

✓ The process of creating more than one sub classes from one super class is called **Hierarchical Inheritance.**



✓ **Example:**

```
1.  class Animal
2.  {
3.  void eat()
4.  {
5.  System.out.println("eating...");
6.  }
7.  }
8.  class Dog extends Animal
9.  {
10. void bark()
11. {
12. System.out.println("barking...");
13. }
14. }
15. class Cat extends Animal
16. {
17. void meow()
18. {
19. System.out.println("meowing...");
20. }
```

```
21.     }
22.     class TestInheritance3
23.     {
24.     public static void main(String args[])
25.     {
26.     Cat c=new Cat();
27.     c.meow();
28.     c.eat();
29.     //c.bark();//C.T.Error
30.     }
31.     }
```

Output:

```
meowing...
eating...
```

Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritances is not supported in java.

Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.

Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error now.

```
class A  {
    void msg()
   {
       System.out.println("Hello");
   }
}
class B  {
   void msg()
   {
        System.out.println("Welcome");
   }
}
class C extends A,B   // this is multiple inheritance which is ERROR
  {
        Public Static void main(String args[])
        {
           C obj=new C();
           obj.msg();//Now which msg() method would be i
nvoked?
        }
}
```

Output

```
Compile Time Error
```

Multiple Inheritance using Interface

```
interface Printable  {
    void print();
}

interface Showable  {
   void show();
}
class A implements Printable, Showable  {
       public void print() {
           System.out.println("Hello");
        }
       public void show() {
          System.out.println("Welcome");
       }
      public static void main(String args[]) {
          A obj = new A();
          obj.print();
          obj.show();
      }
}
```

Output:

```
Hello
Welcome
```

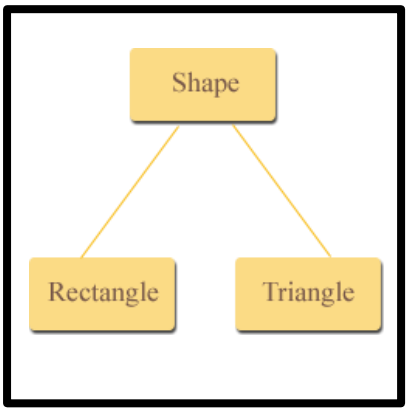
2.4.1: PROTECTED MEMBER

The private members of a class cannot be directly accessed outside the class. Only methods of that class can access the private members directly. However, sometimes it may be necessary for a subclass to access a private member of a superclass. If you make a private member public, then anyone can access that member. So, if a member of a superclass needs to be (directly) accessed in a subclass then you must declare that member **protected**.

Following table describes the difference

Modifier	Class	Package	subclass	World
public	Yes	Yes	Yes	Yes
private	Yes	No	No	No
protected	Yes	Yes	Yes	No

Following program illustrates how the methods of a subclass can directly access a protected member of the superclass.



Consider two kinds of shapes: **rectangles and triangles**. These two shapes have certain common properties height and a width (or base).

This could be represented in the world of classes with a **class Shapes** from which we would derive the two other ones : Rectangle and Triangle

Program : (Shape.java)

```
public class Shape
{
    protected double height;  // To hold height.
    protected double width;  //To hold width or base
```

```
    public void setValues(double height, double width)
   {
         this.height = height;
         this.width = width;
  }
}
```

Program : (Rectangle.java)

```
public class Rectangle extends Shape
{
     public double getArea()
    {
       return height * width; //accessing protected members
    }
}
```

Program : (Triangle.java)

```
public class Triangle extends Shape
{
            public double getArea()
           {
                 return height * width / 2; //accessing protected members
           }
}
```

Program : (TestProgram.java)

```
public class TestProgram
{
       public static void main(String[] args)
      {
           //Create object of Rectangle.
            Rectangle rectangle = new Rectangle();

          //Create object of Triangle.
          Triangle triangle = new Triangle();

          //Set values in rectangle object
         rectangle.setValues(5,4);
```

```
        //Set values in trianlge object
        triangle.setValues(5,10);

       // Display the area of rectangle.
        System.out.println("Area of rectangle : " +
        rectangle.getArea());

     // Display the area of triangle.
      System.out.println("Area of triangle : " +
      triangle.getArea());
    }
}
```

Output :

Area of rectangle : 20.0
Area of triangle : 25.0

2.4.2: CONSTRUCTORS IN SUB – CLASSES

In Java, constructor of base class with no argument gets automatically called in derived class constructor.

When Constructors are Called?

Constructors are called in order of derivation, from superclass to subclass. Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must be executed first.

Example:

```
class A
{
   A()
   {     System.out.println(" Inside A's Constructor");     }
}

class B extends  A
{
   B()
```

```
    {    System.out.println(" Inside B's Constructor");     }
}
class C  extends  B
{
    C()
    {    System.out.println(" Inside C's Constructor");     }
}
class CallingCons
{
    public static void main(String args[])
   {
        C objC=new C();
    }
}
```

Output:

```
Inside A's Constructor
Inside B's Constructor
Inside C's Constructor
```

Program Explanation:

In the above program, we have created three classes A, B and C using multilevel inheritance concept. Here, constructors of the three classes are called in the order of derivation. Since **super()** must be the first statement executed in subclass's constructor, this order is the same whether or not **super()** is used. If **super()** is not used, then the default or parameterless constructor of each superclass will be executed. When inheriting from another class, super() has to be called first in the constructor. If not, the compiler will insert that call. This is why super constructor is also invoked when a Sub object is created.

After compiler inserts the super constructor, the sub class constructor looks like the following:

```
B()
{
        super();
        System.out.println("Inside B's Constructor");
}
C()
{
        super();
```

```
        System.out.println("Inside C's Constructor");
}
```

2.5: "super" keyword

- ✓ Super is a special keyword that directs the compiler to invoke the superclass members. It is used to refer to the parent class of the class in which the keyword is used.
- ✓ **super keyword is used for the following three purposes**:
  1. To invoke superclass constructor.
  2. To invoke superclass members variables.
  3. To invoke superclass methods.

1. Invoking a superclass constructor:

- ✓ **super** as a standalone statement(ie. super()) represents a call to a constructor of the superclass.
- ✓ A subclass can call a constructor method defined by its superclass by use of the following form of **super**:

```
super();
or
super(parameter-list);
```

- ✓ Here, parameter-list specifies any parameters needed by the constructor in the superclass.
- ✓ **super( )** must always be the first statement executed inside a subclass constructor.
- ✓ The compiler implicitly calls the base class's no-parameter constructor or default constructor.
- ✓ If the superclass has parameterized constructor and the subclass constructor does not call superclass constructor explicitly, then the Java compiler reports an error.

2. Invoking a superclass members (variables and methods):

(i) Accessing the instance member variables of the superclass:

Syntax:

```
super.membervariable;
```

(ii) Accessing the methos of the superclass:

Syntax:

```
super.methodName();
```

This call is particularly necessary while calling a method of the super class that is overridden in the subclass.

✓ If a parent class contains a finalize() method, it must be called explicitly by the derived class's finalize() method.

```
super.finalize();
```

Example:

```
class A       // super class
{
  int i;
  A(String str)      //superclass constructor
  {
    System.out.println(" Welcome to "+str);
  }
  void show()       //superclass method
  {
    System.out.println(" Thank You!");
  }
}
class B extends A
{
  int i;     // hides the superclass variable 'i'.
  B(int a, int b)     // subclass constructor
  {
    super("Java Programming");      // invoking superclass constructor
    super.i=a;      //accessing superclass member variable
    i=b;
  }
  // Mehtod overriding
  @Override
  void show()
  {
    System.out.println(" i in superclass : "+super.i);
    System.out.println(" i in subclass : "+i);
    super.show();      // invoking superclass method
```

```
    }
}
public class UseSuper {
    public static void main(String[] args) {
        B objB=new B(1,2);      // subclass object construction
        objB.show();        // call to  subclass method show()
    }
}
```

Output:

```
Welcome to Java Programming
        i in superclass : 1
        i in subclass : 2
        Thank You!
```

Program Explanation:

In the above program, we have created the base class named **A** that contains a instance variable **'i'** and a method **show().** Class A contains a parameterized constructor that receives string as a parameter and prints that string. Class **B** is a subclass of **A** which contains a instance variable 'i' (hides the superclass variable 'i') and overrides the superclass method **show().** The subclass defines the constructor with two parameters a **and b.** The subclass constructor invokes the superclass constructor **super(String)** by passing the string "Java Programming" and assigns the value **a** to the superclass variable**(super.i=a)** and **b** to the subclass variable. The show() method of subclass prints the values of 'i' form both superclass and subclass & invokes the superclass method as **super.show().**

In the main class, object for subclass **B** is created and the object is used to invoke **show()** method of subclass.

2.6: METHOD OVERRIDING

The process of a subclass redefining a method contained in the superclass (with the same method signature) is called Method Overriding.

- ✓ When a method in a subclass has the same name and type signature as a method in its superclass, then the method in subclass is said to override a method in the superclass.

Example:

```
class Bank
{
int getRateOfInterest()// super class method
{
return 0;
}
}
class Axis extends Bank// subclass of bank
{
int getRateOfInterest()// overriding the superclass method
{
return 6;
}
}
class ICICI extends Bank// subclass of Bank
{
        int getRateOfInterest()// overriding the superclass method
{
return 15;
}
}
// Mainclass
class BankTest
{
public static void main(String[] a)
{
Axis a=new Axis();
ICICI i=new ICICI();
// following method call invokes the overridden method of subclass AXIS
System.out.println("AXIS: Rate of Interest = "+a.getRateOfInterest());

// following method call invokes the overridden method of subclass ICICI
System.out.println("ICICI: Rate of Interest = "+i.getRateOfInterest());
}
}
```

Output:

Z:\> java BankTest

AXIS: Rate of Interest = 6
ICICI: Rate of Interest = 15

- **RULES FOR METHOD OVERRIDING:**

  - ✓ The method signature must be same for all overridden methods.
  - ✓ Instance methods can be overridden only if they are inherited by the subclass.
  - ✓ A method declared final cannot be overridden.
  - ✓ A method declared static cannot be overridden but can be re-declared.
  - ✓ If a method cannot be inherited, then it cannot be overridden.
  - ✓ Constructors cannot be overridden.

- **ADVANTAGE OF JAVA METHOD OVERRIDING**

  - ✓ Method Overriding is used to provide specific implementation of a method that is already provided by its super class.
  - ✓ Method Overriding is used for Runtime Polymorphism

2.7: DYNAMIC METHOD DISPATCH

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Example that illustrate dynamic method dispatch:

```
class A {
void callme() {
System.out.println("Inside A's callme method");
}
}

class B extends A {
 //override callme()
void callme() {
System.out.println("Inside B's callme method");
} }
```

```
class C extends A
{
 //override callme()
void callme() {
System.out.println("Inside C's callme method");
}
}

class Dispatch
{
public static void main(String args[])
{
A a=new A();            //object of type A
B b=new B();            //object of type B
C c=new C();            //object of type C
A r;// obtain a reference of type A

r = a;      // r refers to an A object    // dynamic method dispatch
r.callme();// calls A's version of callme()

r = b;// r refers to an B object
r.callme();// calls B's version of callme()

r = c;// r refers to an C object
r.callme();// calls C's version of callme()
}
}
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

- **DIFFERENCE BETWEEN METHOD OVERLOADING AND METHOD OVERRIDING IN JAVA:**

	Method Overloading	**Method Overriding**
Definition	In Method Overloading, Methods of the same class shares the same name but each method must have different number of parameters or parameters having different types and order.	In Method Overriding, sub class have the same method with same name and exactly the same number and type of parameters and same return type as a super class.
Meaning	Method Overloading means more than one method shares the same name in the class but having different signature.	Method Overriding means method of base class is re-defined in the derived class having same signature.
Behavior	Method Overloading is to "add" or "extend" more to method's behavior.	Method Overriding is to "Change" existing behavior of method.
Overloading and Overriding is a kind of polymorphism. Polymorphism means "one name, many forms".		
Polymorphism	It is a **compile time polymorphism**.	It is a **run time polymorphism**.
Inheritance	It may or may not need **inheritance** in Method Overloading.	It always requires inheritance in Method Overriding.
Signature	In Method Overloading, methods must have **different signature**.	In Method Overriding, methods must have **same signature**.
Relationship of Methods	In Method Overloading, relationship is there between methods of same class.	In Method Overriding, relationship is there between methods of super class and sub class.
No. of Classes	Method Overloading does not require more than one class for overloading.	Method Overriding requires at least two classes for overriding.

Example	Class Add
	{
	int sum(int a, int b)
	{
	return a + b;
	}
	int sum(int a)
	{
	return a + 10;
	}
	}
		Class A // Super Class
		{
		void display(int num)
		{
		print num ;
		}
		}
		//Class B inherits Class A
		Class B //Sub Class
		{
		void display(int num)
		{
		print num ;
		}
		}

2.8: ABSTRACT CLASSES

- **Abstraction:**

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

- ✓ For example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.
- ✓ Abstraction lets you focus on what the object does instead of how it does it.

- **Ways to achieve Abstraction**

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

- **Abstract Classes:**

A class that is declared as abstract is known as **abstract class**. Abstract classes cannot be instantiated, but they can be subclassed.

✓ **Syntax to declare the abstract class:**

```
abstract class <class_name>
{
        Member variables;
        Concrete methods { }
        Abstract methods();
}
```

✓ Abstract classes are used to provide common method implementation to all the subclasses or to provide default implementation.

Properties of abstract class:

- abstract keyword is used to make a class abstract.
- Abstract class can't be instantiated.
- If a class has abstract methods, then the class also needs to be made abstract using abstract keyword, else it will not compile.
- Abstract classes can have both concrete methods and abstract methods.
- The subclass of abstract class must implement all the abstract methods unless the subclass is also an abstract class.
- A constructor of an abstract class can be defined and can be invoked by the subclasses.
- We can run abstract class like any other class if it has main() method.

Example:

```
abstract class GraphicObject {
   int x, y;
   ...
   void moveTo(int newX, int newY) {
      ...
   }
   abstract void draw();
   abstract void resize();
}
```

- **Abstract Methods:**

A method that is declared as abstract and does not have implementation is known as **abstract method.** It acts as placeholder methods that are implemented in the subclasses.

✓ **Syntax to declare a abstract method:**

```
abstract class classname
{

abstract return_type <method_name>(parameter_list);//no braces{}
        // no implementation required
                ……..
}
```

✓Abstract methods are used to provide a template for the classes that inherit the abstract methods.

Properties of abstract methods:

- The abstract keyword is also used to declare a method as abstract.
- An abstract method consists of a method signature, but no method body.
- If a class includes abstract methods, the class itself must be declared abstract.
- Abstract method would have no definition, and its signature is followed by a semicolon, not curly braces as follows:

```
public abstract class Employee {
  private String name;
  private String address;
  private int number;
  public abstract double computePay();
  //Remainder of class definition
}
```

- Any child class must either override the abstract method or declare itself abstract.

Write a Java program to create an abstract class named Shape that contains 2 integers and an empty method named PrintArea(). Provide 3 classes named Rectangle, Triangle and Circle such that each one of the classes extends the class Shape. Each one of the classes contain only the method PrintArea() that prints the area of the given shape.

```
abstract class shape
{
    int x, y;
    abstract void printArea();
}
```

```
class Rectangle extends shape
{
void printArea()
{
      System.out.println("Area of Rectangle is " + x * y);
}
}
class Triangle extends shape
{
void printArea()
{
System.out.println("Area of Triangle is " + (x * y) / 2);
}
}
class Circle extends shape
{
void printArea()
{
        System.out.println("Area of Circle is " + (22 * x * x) / 7);
}
}
class abs
{
public static void main(String[] args)
{
Rectangle r = new Rectangle();
r.x = 10;
r.y = 20;
r.printArea();

System.out.println(" ---------------------------------------------- ");

Triangle t = new Triangle();
t.x = 30;
t.y = 35;
t.printArea();

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

```
Circle c = new Circle();
c.x = 2;
c.printArea();

System.out.println(" ---------------------------------------------- ");
}
}
```

Output:

```
D:\>javac abs.java
D:\>java abs
Area of Rectangle is 200
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -
Area of Triangle is 525
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -
Area of Circle is 12
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -
```

2.9: final WITH INHERITANCE

What is final keyword in Java?

Final is a keyword or reserved word in java used for restricting some functionality. It can be applied to member variables, methods, class and local variables in Java.

- ✓ **final** keyword has three uses:
  1. For declaring variable – **to create a named constant.** A final variable cannot be changed once it is initialized.
  2. For declaring the methods – **to prevent method overriding**. A final method cannot be overridden by subclasses.
  3. For declaring the class – **to prevent a class from inheritance.** A final class cannot be inherited.

1. Final Variable:

Any variable either member variable or local variable (declared inside method or block) modified by final keyword is called final variable.

✓The final variables are equivalent to **const qualifier** in C++ and **#define** directive in C.

✓Syntax:

✓final data_type variable_name = value;

✓Example:

final int MAXMARKS=100;
final int PI=3.14;

✓The final variable can be assigned only once.

✓The value of the final variable will not be changed during the execution of the program. If an attempt is made to alter the final variable value, the java compiler will throw an error message.

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

```
1.  class Bike
2.  {
3.       final int speedlimit=90;//final variable
4.       void run( )
5.      {
6.         speedlimit=400;
7.      }
8.      public static void main(String args[])
9.     {
10.         Bike obj=new  Bike();
11.         obj.run();
12.    }
13.}
```

Output: Compile Time Error

NOTE: Final variables are by default read-only.

2. Final Methods:

- ✓ Final keyword in java can also be applied to methods.
- ✓ **A java method with final keyword is called final method and it cannot be overridden in sub-class.**
- ✓ If a method is defined with final keyword, it cannot be overridden in the subclass and its behaviour should remain constant in sub-classes.
- ✓ **Syntax:**

```
final return_type function_name(parameter_list)
{
// method body
}
```

✓ **Example of final method in Java:**

```
1.  class Bike
2.  {
3.     final void run()
4.    {
5.          System.out.println("running");
6.    }
7.  }
8.  class Honda extends Bike
9.  {
10.   void run()
11.  {
12.       System.out.println("running safely with 100kmph");
13.  }
14.  public static void main(String args[])
15. {
16.       Honda honda= new Honda();
17.       honda.run();
18.  }
19.}
```

Output:

```
D:\>javac Honda.java
Honda.java:9: error: run() in Honda cannot override run() in Bike
            void run()
                 ^
       overridden method is final
       1  error
```

3. Final Classes:

- ✓ **Java class with final modifier is called final class in Java and they cannot be sub-classed or inherited.**
- ✓ **Syntax:**

```
final class class_name
{
    // body of the class
}
```

- ✓ Several classes in Java are final e.g. String, Integer and other wrapper classes.

- ✓ **Example of final class in java:**

```
1. final class Bike
2. {
3. }
4. class Honda1 extends Bike
5. {
6.      void run()
7.     {
8.        System.out.println("running safely with 100kmph");
9.     }
10.    public static void main(String args[])
11.   {
12.       Honda1 honda= new Honda1();
13.       honda.run();
14.   }
15.   }
```

Output:

D:\>javac Honda.java
Honda.java:4: error: cannot inherit from final Bike class Honda extends Bike
^
1 error

Points to Remember:

1) A constructor cannot be declared as final.
2) Local final variable must be initializing during declaration.
3) All variables declared in an interface are by default final.
4) We cannot change the value of a final variable.
5) A final method cannot be overridden.
6) A final class cannot be inherited.
7) If method parameters are declared final then the value of these parameters cannot be changed.
8) It is a good practice to name final variable in all CAPS.
9) final, finally and finalize are three different terms. finally is used in exception handling and
10) finalize is a method that is called by JVM during garbage collection.

2.10: PACKAGES

Definition:

A Package can be defined as a collection of classes, interfaces, enumerations and annotations, providing access protection and name space management.

- ✓ Package can be categorized in two form:
  1. Built-in package
  2. user-defined package.

Packages	*Description*
Java.lang	It is a default package which contain primitive data type, displaying result on console screen, obtaining garbage collector etc.
java.io	It used for developing file handling applications, such as, opening the file in read or write mode, reading or writing the data, etc.
java.awt	This package is used for developing GUI (Graphic User Interface) components such as buttons, check boxes, scroll boxes, etc.
Java. applet	This package is used for developing browser oriented applications.
java.net	This package is used for developing client server applications.
java.util	Contains utility classes which implement data structures like Hash Table, Dictionary, etc.
java.sql	This package is used for retrieving the data from data base and performing various operations on data base.

Table: List of Built-in Packages

Advantage of Package:

- Package is used to categorize the classes and interfaces so that they can be easily maintained.
- Package provides access protection.
- Package removes naming collision.
- To bundle classes and interface
- The classes of one package are isolated from the classes of another package
- Provides reusability of code
- We can create our own package or extend already available package

2.10.1 : CREATING USER DEFINED PACKAGES:

Java package created by user to categorize their project's classes and interface are known as user-defined packages.

- ✓ When creating a package, you should choose a name for the package.
- ✓ Put a **package** statement with that name at the top of every source file that contains the classes and interfaces.
- ✓ The **package** statement should be the first line in the source file.
- ✓ There can be only one package statement in each source file

- ✓ **Syntax:**

```
package package_name.[sub_package_name];
public class classname
{  ……..
  ……..
}
```

- ✓ Steps involved in creating user-defined package:
    1. Create a directory which has the same name as the package.
    2. Include package statement along with the package name as the first statement in the program.
    3. Write class declarations.
    4. Save the file in this directory as "name of class.java".
    5. Compile this file using java compiler.

- ✓ Example:

```
package pack;
public class class1 {
public static void greet()
{  System.out.println("Hello");  }
}
```

To create the above package,

1. Create a directory called pack.
2. Open a new file and enter the code given above.
3. Save the file as class1.java in the directory.
4. A package called pack has now been created which contains one class class1.

2.10.2 : ACCESSING A PACKAGE (using "import" keyword):

- The import keyword is used to make the classes and interface of another package accessible to the current package.

Syntax:

import package1[.package2][.package3].classname or *;

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

✓ **Using packagename.***

- If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

✓ **Using packagename.classname**

- If you import package.classname then only declared class of this package will be accessible.

✓ **Using fully qualified name**

- If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

Example :

greeting.java (create a folder named "pack" in **F:** and save **)**

```
package pack;
public class greeting{
public static void greet()
{  System.out.println("Hello! Good Morning!");  }
}
```

FactorialClass.java (create a folder named "Factorial" inside **F:\pack** and save**)**

```
package Factorial;
public class FactorialClass
{
```

```
public int fact(int a)
{
if(a==1)
return 1;
else
return a*fact(a-1);
}
}
```

ImportClass.java (save the file in F:\ **)**

```
import java.lang.*;  // using import package.*
import pack.Factorial.FactorialClass; // using import package.subpackage.class;
import java.util.Scanner;
public class ImportClass
{
public static void main(String[] arg)
{
int n;
Scanner in=new Scanner(System.in);
System.out.println("Enter a Number: ");
n=in.nextInt();
pack.greeting p1=new pack.greeting();  // using fully qualified name
p1.greet();
FactorialClass fobj=new FactorialClass();
System.out.println("Factorial of "+n+" = "+fobj.fact(n));
System.out.println("Power("+n+",2) = "+Math.pow(n,2));
}
}
```

Output:

```
F:\>java ImportClass
Enter a Number:
5
Hello! Good Morning!
Factorial of 5 = 120
Power(5,2) = 25.0
```

2.10.3 : PACKAGES AND MEMBER ACCESS:

Access level modifiers determine whether other classes can use a particular field or invoke a particular method.

There are two levels of access control:

- **At the top level— public,** or package-private (no explicit modifier).
- **At the member level—public, private, protected,** or package-private (no explicit modifier).

Top Level access control:

- ✓ A class may be declared with the modifier public, in which case that class is visible to all classes everywhere.
- ✓ If a class has no modifier (the default, also known as package-private), it is visible only within its own package.

Member Level access control:

- ✓ **public** – if a member is declared with public, it is visible and accessible to all classes everywhere.
- ✓ **private** - The private modifier specifies that the member can only be accessed in its own class.
- ✓ **protected** - The protected modifier specifies that the member can only be accessed within its own package and, in addition, by a subclass of its class in another package.

The following table shows the access to members permitted by each modifier.

Access Levels				
Modifier	**Class**	**Package**	**Subclass**	**World**
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

The following figure shows the four classes in this example and how they are related.

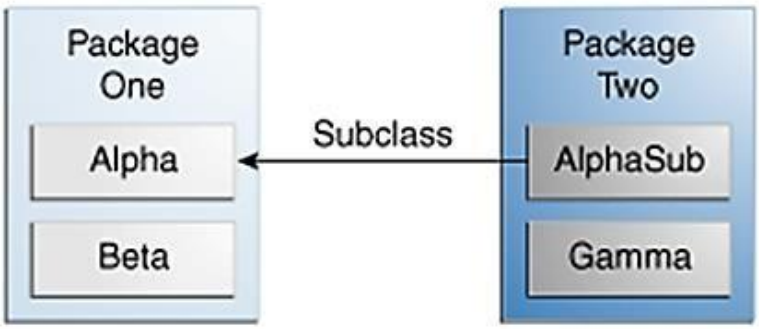


Figure: Classes and Packages of the Example Used to Illustrate Access Levels

The following table shows where the members of the Alpha class are visible for each of the access modifiers that can be applied to them.

Visibility				
Modifier	**Alpha**	**Beta**	**Alphasub**	**Gamma**
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Example:

Z:\MyPack\FirstClass.java

```
package MyPack;

public class FirstClass
{
   public String i="I am public variable";
   protected String j="I am protected variable";
   private String k="I am private variable";
   String r="I dont have any modifier";
}
```

Z:\MyPack2\SecondClass.java

```
package MyPack2;
import MyPack.FirstClass;
class SecondClass extends FirstClass {
   void method()
   {
      System.out.println(i);      // No Error: Will print "I am public variable".
     System.out.println(j);    // No Error: Will print “I am protected variable”.
     System.out.println(k);  // Error: k has private access in FirstClass
     System.out.println(r);  // Error: r is not public in FirstClass; cannot be accessed
                                //            from outside package
   }
```

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

Output:

I am public variable
I am protected variable

Exception in thread "main" java.lang.RuntimeException: Uncompilable source code - k has private access in MyPack.FirstClass

Visibility of the variables i,j,k and r in MyPack2				
Accessibility	**i**	**j**	**k**	**r**
Class	Y	Y	Y	Y
Package	Y	Y	N	N
Subclass	Y	Y	N	N
world	Y	N	N	N

Table: Accessibility of variables of MyyPack/FirstClass in MyPack2/SecondClass

2.11: INTERFACES

"interface" is a keyword which is used to achieve full abstraction. Using interface, we can specify what the class must do but not how it does.

Interfaces are syntactically similar to classes but they lack instance variable and their methods are declared without body.

Definition:

An interface is a collection of method definitions (without implementations) and constant values. It is a blueprint of a class. It has static constants and abstract methods.

- **Why use Interface?**

  There are mainly three reasons to use interface. They are given below.

  - It is used to achieve fully abstraction.

- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.
- Writing flexible and maintainable code.
- Declaring methods that one or more classes are expected to implement.

➢ **An interface is similar to a class in the following ways:**

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The bytecode of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

➢ **Defining Interfaces:**

An interface is defined much like a class. The keyword **"interface"** is used to define an interface.

Syntax to define interface:

```
[access_specifier] interface InterfaceName
{
Datatype VariableName1=value;
Datatype VariableName2=value;
.
.
Datatype VariableNameN=value;
returnType methodName1(parameter_list);
returnType methodName2(parameter_list);
.
.
returnType methodNameN(parameter_list);
}
```

Where,

Access_specifer : either **public** or none.

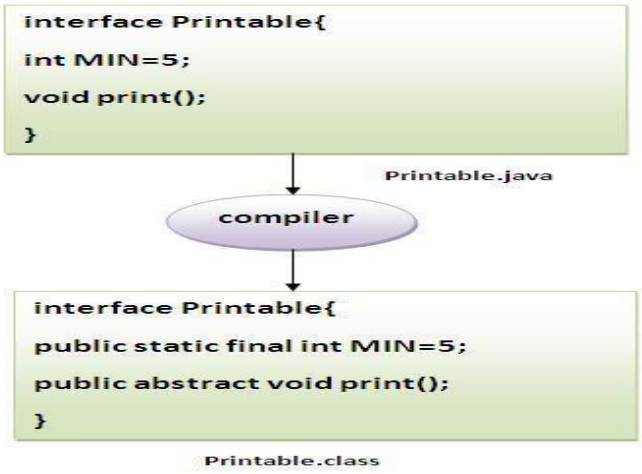
Name: name of an interface can be any valid java identifier.

Variables: They are implicitly **public, final and static**, meaning that they cannot be changed by the implementing class. They must be initialized with a constant value.

Methods: They are implicitly **public and abstract**, meaning that they must be declared without body and defined only by the implementing class.

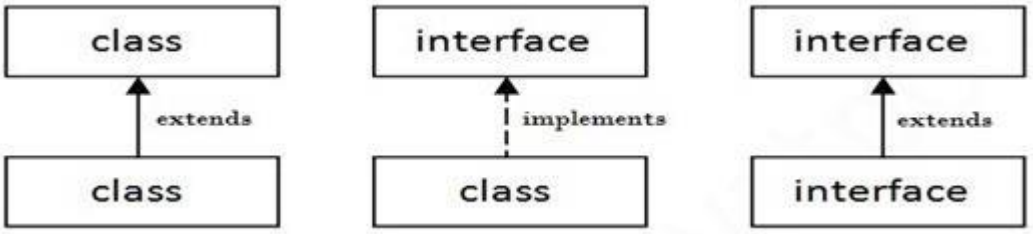
Note: The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

- ✓ In other words, **Interface fields are public, static and final by default, and methods are public and abstract.**



- ➢ **<u>Understanding relationship between classes and interfaces</u>**

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.



- ➢ **<u>Implementing Interfaces ("implements" keyword):</u>**

- ✓ Once an interface has been defined, one or more classes can implement that interface.
- ✓ A class uses the **implements** keyword to implement an interface.
- ✓ The implements keyword appears in the class declaration following the extends portion of the declaration.

- ✓ **Syntax:**

```
[access_specifier] class class_name [extends superclassName] implements
interface_name1, interface_name2…
{
//implementation code and code for the method of the interface
}
```

Rules:

1. If a class implements an interface, then it must provide implementation for all the methods defined within that interface.
2. A class can implement more than one interfaces by separating the interface names with comma(,).
3. A class can extend only one class, but implement many interfaces.
4. An interface can extend another interface, similarly to the way that a class can extend another class.
5. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

✓ **Example:**

```
/* File name : Super.java */
interface Super
 {
   final int x=10;
   void print();
}
/* File name : Sub.java */
class Sub implements Super
{
   int y=20;
   x=100                //ERROR; cannot change modify the value of final variable

  // defining the method of interface
   public void print()
   {
            System.out.println("X = "+x);
            System.out.println("Y = "+y);
   }
}
class sample
{
   public static void main(String arg[])
```

```
    {
      Sub SubObj=new Sub();
      SubObj.print();
      Super SupObj=new Sub(); //  interface variable referring to class object
      SupObj.print();
    }
  }
```

Output:

```
$java sample
X = 10
Y = 20
X = 10
Y = 20
```

- **<u>The rules for interfaces:</u>**

Member variables:

- Can be only public and are by default.
- By default are static and always static
- By default are final and always final

Methods:

- Can be only public and are by default.
- Cannot be static
- Cannot be Final

- **<u>When overriding methods defined in interfaces there are several rules to be followed:</u>**
  - The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
  - An implementation class itself can be abstract and if so interface methods need not be implemented.

- **<u>Properties of Interfaces:</u>**
  1. Interfaces are not classes. So the user can never use the new operator to instantiate an interface.
     Example: interface super {}
     X=new Super() // ERROR

2. The interface variables can be declared, even though the interface objects can't be constructed.

   Super x; // OK
3. An interface variable must refer to an object of a class that implements the interface.
4. The instanceOf() method can be used to check if an object implements an interface.
5. A class can extend only one class, but implement many interfaces.
6. An interface can extend another interface, similarly to the way that a class can extend another class.
7. All the methods in the interface are **public** and **abstract.**
8. All the variable in the interface are **public, static** and **final.**

- **<u>Extending Interfaces:</u>**
  - ✓ An interface can extend another interface, similarly to the way that a class can extend another class.
  - ✓ The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.
  - ✓ **Syntax:**

```
[accessspecifier] interface InterfaceName extends interface1, interface2,.....
{
Code for interface
}
```

Rule: When a class implements an interface that inherits another interface it must provide implementation for all the methods defined within the interface inheritance chain.

Example:

```
interface A
{
        void method1();
}
/* One interface can extend another interface. B now has two abstract methods */
interface B extends A
{
void method2();
}
```

```
// This class must implement all the methods of A and B

class MyClass implements B
{
public void method1() // overriding the method of interface A
{
System.out.println("—Method from interface: A—");
}
public void method2() // overriding the method of interface B
{
System.out.println("—Method from interface: B—");
}
public void method3() // instance method of class MyClass
{
System.out.println("—Method of the class : MyClass—");
}
public static void main(String[] arg)
{
MyClass obj=new MyClass();
Obj.method1();
Obj.method2();
Obj.method3();
}
}
```

Output:

F:\> java MyClass

--Method from Interface: A—
--Method from Interface: B—
--Method of the class: MyClass--

- **Difference between Class and Interface:**

Class	Interface
The class is denoted by a keyword **class**	The interface is denoted by a keyword **interface**
The class contains data members and methods. but the methods are defined in the class implementation. thus class contains an executable code	The interfaces may contain data members and methods but the methods are not defined. the interface serves as an outline for the class
By creating an instance of a class the class members can be accessed	you cannot create an instance of an interface
The class can use various access specifiers like public, private or protected	The interface makes use of only public access specifier
The members of a class can be constant or final	The members of interfaces are always declared as final

- **Difference between Abstract class and interface**

Abstract Class	Interface
Multiple inheritance is not possible; the class can inherit only one abstract class	Multiple inheritance is possible; The class can implement more than one interfaces
Members of abstract class can have any access modifier such as **public, private and protected**	Members of interface are **public** by default
The methods in abstract class may be **abstract method or concrete method**	The methods in interfaces are **abstract** by default
The method in abstract class **may or may not have implementation**	The methods in interface have **no implementation at all.** Only declaration of the method is given
Java abstract class is extended using the keyword **extends**	Java interface can be implemented by using the keyword **implements**
The member variables of abstract class can be **non-final**	The member variables of interface are **final** by default
Abstract classes **can have constructors**	Interfaces **do not have any constructor**
Only abstract methods need to be overridden.	All the method of an interface must be overridden.
Non-abstract methods can be static.	Methods cannot be static.

Example:
public abstract class Shape
{
public abstract void draw();
}
	Example:
	public interface Drawable
	{
	void draw();
	}

Example for Interface :

```
1.  interface Bank
2.  {
3.      float rateOfInterest();
4.  }
5.  class SBI implements Bank
6.  {
7.       public float rateOfInterest()
8.      {
9.          return 9.15f;
10.      }
11. }
12. class PNB implements Bank
13. {
14.      public float rateOfInterest()
15.     {
16.         return 9.7f;
17.     }
18. }
19. class TestInterface2
20. {
21.     public static void main(String[] args)
22.    {
23.        Bank b=new SBI();
24.        System.out.println("ROI:  "+b.rateOfInterest());
25.    }
26. }
```

Output:

```
ROI: 9.15
```

Unit – 3: EXCEPTION HANDLING AND MULTITHREADING

UNIT- 3 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

Definition:

An Exception is an event that occurs during program execution which disrupts the normal flow of a program. It is an object which is thrown at runtime.

Occurrence of any kind of exception in java applications may result in an abrupt termination of the JVM or simply the JVM crashes.

In Java, an exception is an **object** that contains:

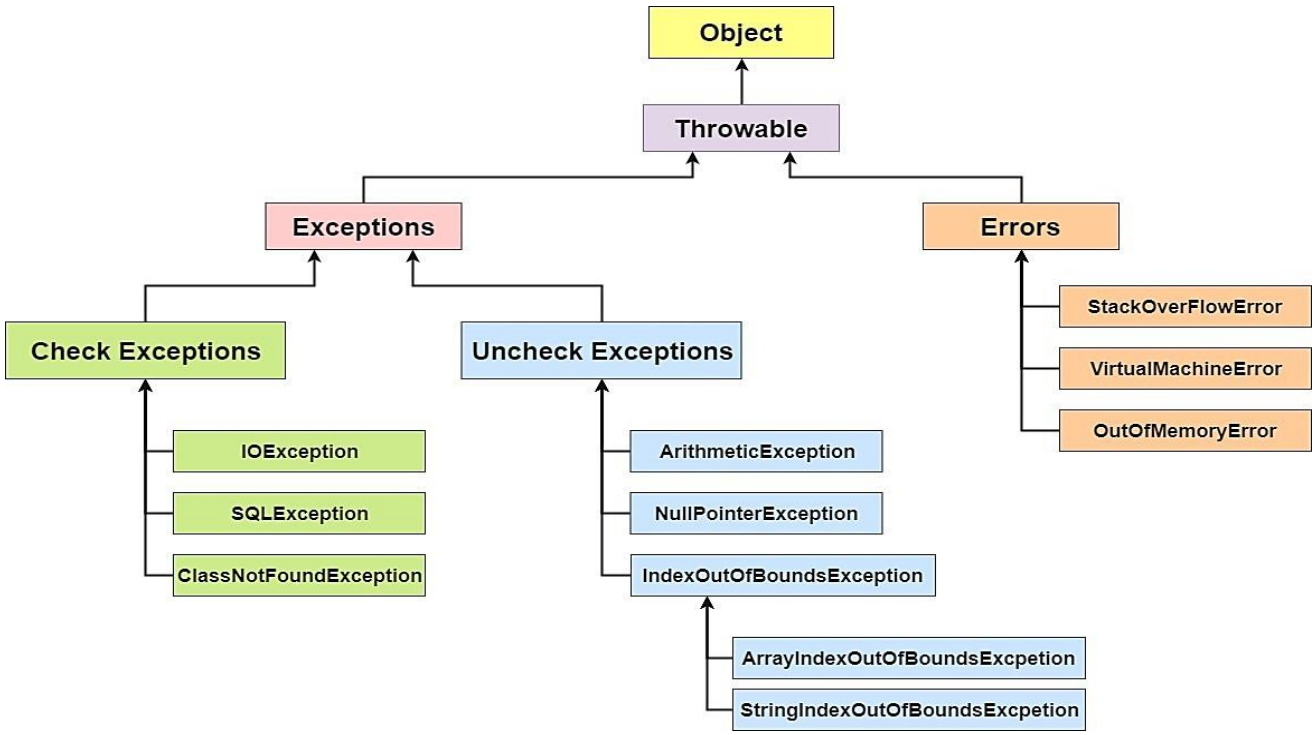
- Information about the error including its type
- The state of the program when the error occurred
- Optionally, other custom information

3.1.1: Exception Hierarchy

All exceptions and errors extend from a common java.lang.Throwable parent class.

The **Throwable** class is further divided into two classes:

1. **Exceptions** and
2. **Errors.**



Exceptions: Exceptions represents errors in the Java application program, written by the user. Because the error is in the program, exceptions are expected to be handled, either

- Try to recover it if possible
- Minimally, enact a safe and informative shutdown.

Sometimes it also happens that the exception could not be caught and the program may get terminated. Eg. **ArithmeticException**

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Errors: Errors represent internal errors of the Java run-time system which could not be handled easily. Eg. **OutOfMemoryError.**

DIFFERENCE BETWEEN EXCEPTION AND ERROR:

S.No.	Exception	Error
1.	Exceptions can be recovered	Errors cannot be recovered
2.	Exceptions are of type **java.lang.Exception**	Errors are of type **java.lang.Error**
3.	Exceptions can be classified into two types: a) Checked Exceptions b) Unchecked Exceptions	There is no such classification for errors. Errors are always unchecked.
4.	In case of Checked Exceptions, compiler will have knowledge of checked exceptions and force to keep try…catch block. Unchecked Exceptions are not known to compiler because they occur at run time.	In case of Errors, compiler won't have knowledge of errors. Because they happen at run time.
5.	Exceptions are mainly caused by the application itself.	Errors are mostly caused by the environment in which application is running.

6.	Examples: **Checked Exceptions:** SQLException, IOException **Unchecked Exceptions:** ArrayIndexOutOfBoundsException, NullPointerException	Examples: Java.lang.StackOverFlowError, java.lang.OutOfMemoryError

3.1.2: Exception Handling

What is exception handling?

Exception Handling is a mechanism to handle runtime errors, such as ClassNotFoundException, IOException, SQLException, RemoteException etc. by taking the necessary actions, so that normal flow of the application can be maintained.

Advantage of using Exceptions:

- Maintains the normal flow of execution of the application.
- Exceptions separate error handling code from regular code.
  - Benefit: Cleaner algorithms, less clutter
- Meaningful Error reporting.
- Exceptions standardize error handling.

JAVA EXCEPTION HANDLING KEYWORDS

Exception handling in java is managed using the following five keywords:

S.No.	Keyword	Description
1	**try**	A block of code that is to be monitored for exception.
2	**catch**	The catch block handles the specific type of exception along with the try block. For each corresponding try block there exists the catch block.
3	**finally**	It specifies the code that must be executed even though exception may or may not occur.
4	**throw**	This keyword is used to explicitly throw specific exception from the program code.
5	**throws**	It specifies the exceptions that can be thrown by a particular method.

- **try Block:**
  - ✓ The java code that might throw an exception is enclosed in try block. It must be used within the method and must be followed by either catch or finally block.

- ✓ If an exception is generated within the try block, the remaining statements in the try block are not executed.

- **catch Block:**
  - ✓ Exceptions thrown during execution of the try block can be caught and handled in a catch block.
  - ✓ On exit from a catch block, normal execution continues and the finally block is executed.

- **finally Block:**

  A finally block is always executed, regardless of the cause of exit from the try block, or whether any catch block was executed.
  - ✓ Generally finally block is used for freeing resources, cleaning up, closing connections etc.
  - ✓ Even though there is any exception in the try block, the statements assured by **finally** block are sure to execute.
  - ✓ **Rule:**
    - **For each try block there can be zero or more catch blocks, but only one finally block.**
    - **The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).**

The try-catch-finally structure(Syntax):

```
try {
   // Code block
}
catch (ExceptionType1 e1) {
   // Handle ExceptionType1 exceptions
}
catch (ExceptionType2 e2)  {
   // Handle ExceptionType2 exceptions
}
// ...
finally {
   // Code always executed after the
   // try and any catch block
}
```

Rules for try, catch and finally Blocks:

1) Statements that might generate an exception are placed in a try block.
2) Not all statements in the try block will execute; the execution is interrupted if an exception occurs
3) For each try block there can be zero or more catch blocks, but only one finally block.
4) The try block is followed by
   i. one or more catch blocks
   ii. or, if a try block has no catch block, then it must have the finally block
5) A try block must be followed by either at least one catch block or one finally block.
6) A catch block specifies the type of exception it can catch. It contains the code known as exception handler
7) The catch blocks and finally block must always appear in conjunction with a try block.
8) The order of exception handlers in the catch block must be from the most specific exception

Program without Exception handling: (Default exception handler):

```
class Simple
{
  public static void main(String args[])
{
    int data=50/0;

    System.out.println("rest of the code...");
}
}
```

Output:

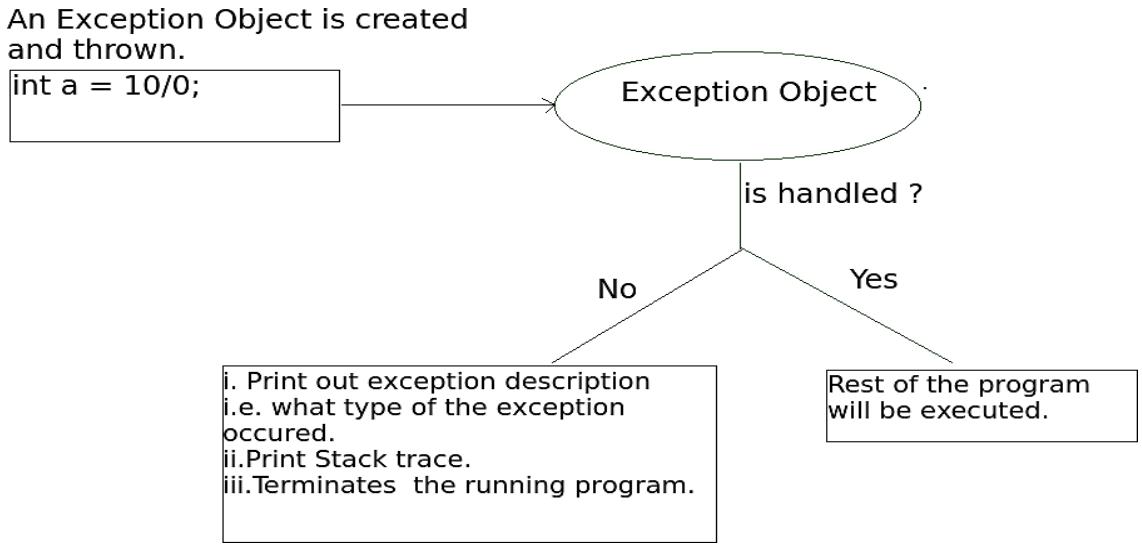
Exception in thread main java.lang.ArithmeticException:/ by zero

As displayed in the above example, rest of the code is not executed i.e. rest of the code... statement is not printed.

Program Explanation:

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.

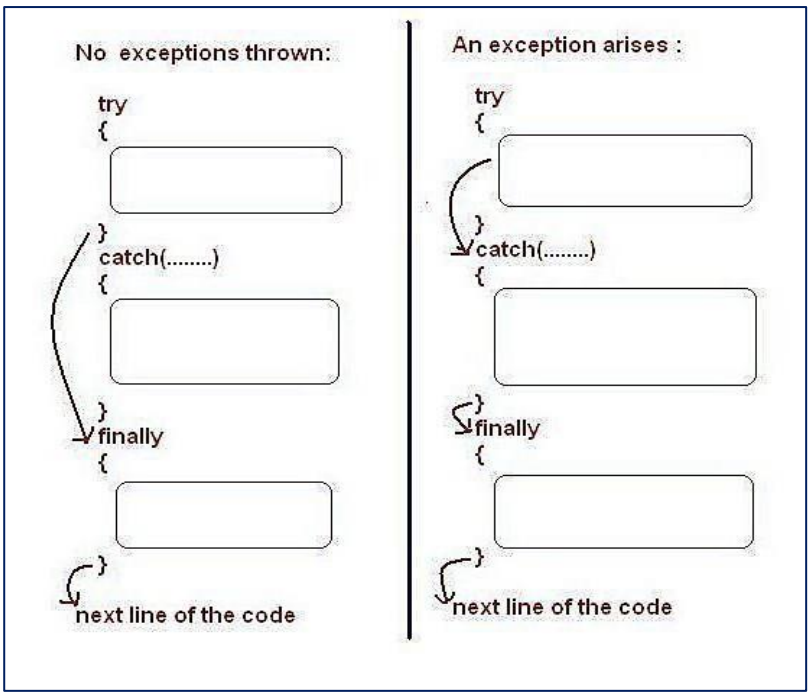


Example:

```
public class Demo
{
 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:

java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...



3.2: Multiple catch blocks

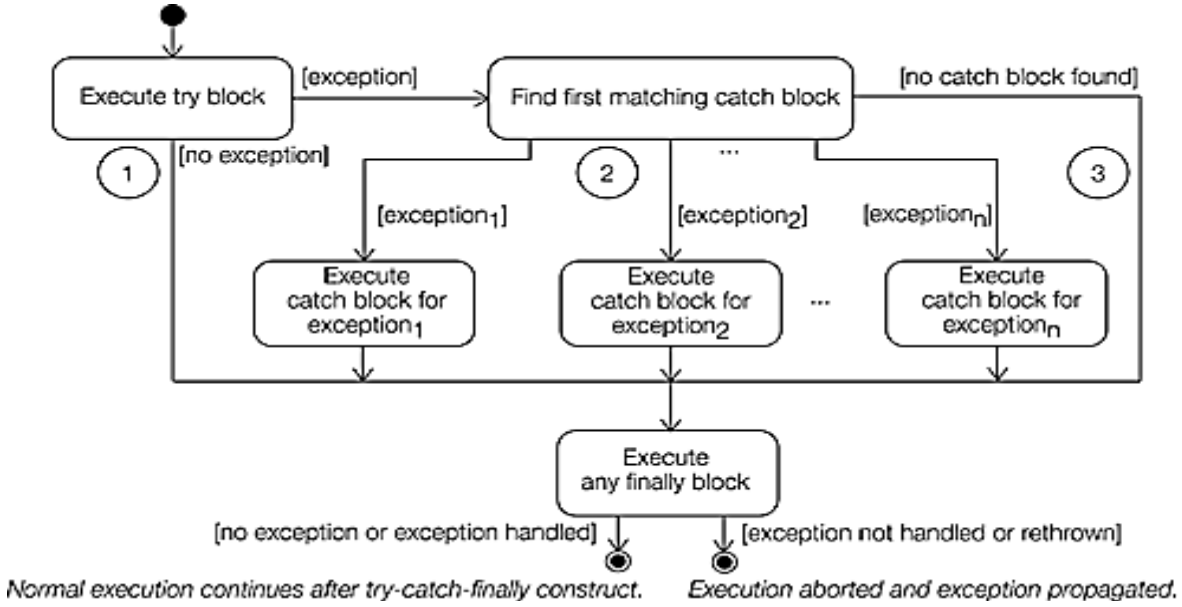
Multiple catch is used to handle many different kind of exceptions that may be generated while running the program. i.e more than one catch clause in a single try block can be used.

Rules:

- *At a time only one Exception can occur and at a time only one catch block is executed.*
- *All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception.*

Syntax:

```
try {
   // Code block
}
catch (ExceptionType1 e1) {
   // Handle ExceptionType1 exceptions
}
catch (ExceptionType2 e2)  {
   // Handle ExceptionType2 exceptions
}
```



Example:

```
public class MultipleCatchBlock2 {

public static void main(String[] args) {

  try
{
    int a[]= {1,5,10,15,16};
    System.out.println("a[1] = "+a[1]);
    System.out.println("a[2]/a[3] = "+a[2]/a[3]);
    System.out.println("a[5] = "+a[5]);
  }
  catch(ArithmeticException e)
 {
     System.out.println("Arithmetic Exception occurs");
 }
  catch(ArrayIndexOutOfBoundsException e)
 {
     System.out.println("ArrayIndexOutOfBounds Exception occurs");
 }
  catch(Exception e)
 {
     System.out.println("Parent Exception occurs");
 }
```

```
        System.out.println("rest of the code");
}
}
```

Output:

a[1] = 5
a[2]/a[3] = 0
ArrayIndexOutOfBounds Exception occurs
rest of the code

3.3: Nested Try Block

Definition: try block within a try block is known as nested try block.

Why use nested try block?

- ✓ Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.
- ✓ If an inner try statement does not have a matching catch statement for a particular exception, the control is transferred to the next try statement's catch handlers that for a matching catch statement.
- ✓ If none of the catch statement match, then the Java run-time system will handle the exception.

Syntax:

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

Example:

```
class NestedExcep
{
 public static void main(String[] args)
 {
  try
  {
   int arr[]={1,5,4,10};
   try
   {
    int x=arr[3]/arr[1];
```

```
      System.out.println("Quotient = "+x);
     }
     catch(ArithmeticException ae)
     {
      System.out.println("divide by zero");
     }
     arr[4]=3;
    }
    catch(ArrayIndexOutOfBoundsException e)
    {
     System.out.println("array index out of bound exception");
    }
    System.out.println("...End of Program...");
   }
  }
```

Output:

Quotient = 2
array index out of bound exception
...End of Program...

3.4: THROWING AND CATCHING EXCEPTIONS

Before catching an exception, it is must to throw an exception first. This means that there should be a code somewhere in the program that could catch exception thrown in the try block.

An exception can be thrown explicitly

1. Using the **throw** statement
2. Using the **throws** statement

1: Using the throw statement

- ✓ A program can explicitly throw an exception using the throw statement besides the implicit exception thrown.
- ✓ We can throw either checked, uncheked exceptions or custom(user defined) exceptions
- ✓ When **throw** statement is called:

1) It causes the termination of the normal flow of control of the program code and stops the execution of the subsequent statements.
2) It transfers the control to the nearest catch block handling the type of exception object thrown
3) If no such catch block exists, then the program terminates.

The general format of the **throw** statement is as follows:

throw <exception reference>;

The Exception reference must be of type Throwable class or one of its subclasses. A detail message can be passed to the constructor when the exception object is created.

Example:

```
1)  public class ThrowDemo
2)  {
3)  static void validate(int age)
4)  {
5)  if(age<18)
6)  throw new ArithmeticException("not valid");
7)  else
8)  System.out.println("welcome to vote");
9)  }
10) public static void main(String args[])
11) {
12) validate(13);
13) System.out.println("rest of the code...");
14) }
15) }
```

Output:

Exception in thread "main" java.lang.ArithmeticException: not valid
at ThrowDemo.validate(ThrowDemo.java:6)
at ThrowDemo.main(ThrowDemo.java:12)

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

2: Using throws keyword:

- ✓ The **throws** statement is used by a method to specify the types of exceptions the method throws.
- ✓ If a method is capable of raising an exception that it does not handle, the method must specify that the exception have to be handled by the calling method.
- ✓ This is done using the throws clause. The throws clause lists the types of exceptions that a method might throw.

Syntax:

Return-type method_name(arg_list) **throws** *exception_list*
{
// method body
}

Example:

```
1.  import java.util.Scanner;
2.  public class ThrowsDemo
3.  {
4.  static void divide(int num, int din) throws ArithmeticException
5.  {
6.  int result=num/din;
7.  System.out.println("Result : "+result);
8.  }
9.  public static void main(String args[])
10. {
11. int n,d;
12. Scanner in=new Scanner(System.in);
13. System.out.println("Enter the Numerator : ");
14. n=in.nextInt();
15. System.out.println("Enter the Denominator : ");
16. d=in.nextInt();
17. try
18. {
19. divide(n,d);
20. }
21. catch(Exception e)
22. {
23. System.out.println(" Can't Handle : divide by zero ERROR");
24. }
25. System.out.println(" ** Continue with rest of the code ** ");
26. }
27. }
```

Output:

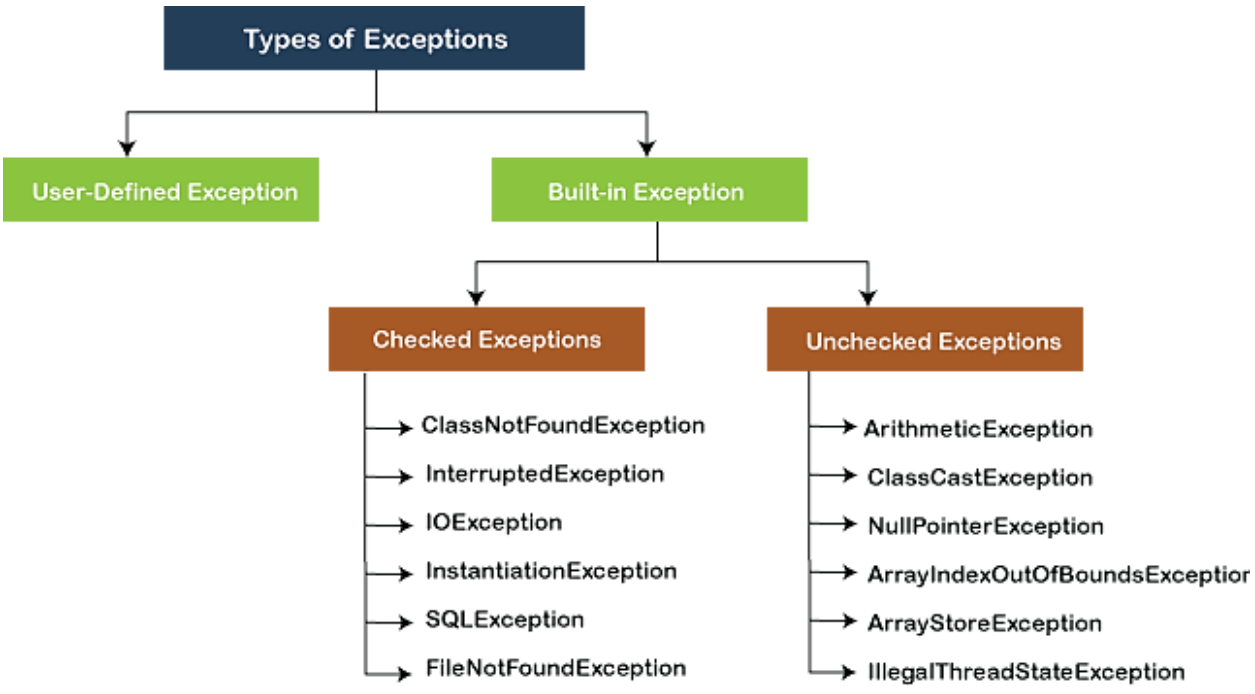
Enter the Numerator :
4

Enter the Denominator :
0
Can't Handle : divide by zero ERROR
**** Continue with rest of the code ****
Enter the Numerator :
6
Enter the Denominator :
2
Result : 3
**** Continue with rest of the code ****

Difference between throw and throws:

throw keyword	throws keyword
1) throw is used to explicitly throw an exception.	throws is used to declare an exception.
2) checked exception cannot be propagated without throws.	checked exception can be propagated with throws.
3) throw is followed by an instance.	throws is followed by class.
4) throw is used within the method.	throws is used with the method signature.
5)You cannot throw multiple exception	You can declare multiple exception e.g. **public void method()throws IOException,SQLException.**

3.5: Types of Exceptions



3.5.1 : Built-in Exceptions

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.

S. No.	Exception	Description
1.	**ArithmeticException**	Thrown when a problem in arithmetic operation is noticed by the JVM.
2.	**ArrayIndexOutOfBoundsException**	Thrown when you access an array with an illegal index.
3.	**ClassNotFoundException**	Thrown when you try to access a class which is not defined
4.	**FileNotFoundException**	Thrown when you try to access a non-existing file.
5.	**IOException**	Thrown when the input-output operation has failed or interrupted.
6.	**InterruptedException**	Thrown when a thread is interrupted when it is processing, waiting or sleeping
7.	**IllegalAccessException**	Thrown when access to a class is denied
8.	**NoSuchFieldException**	Thrown when you try to access any field or variable in a class that does not exist
9.	**NoSuchMethodException**	Thrown when you try to access a non-existing method.
10.	**NullPointerException**	Thrown when you refer the members of a null object
11.	**NumberFormatException**	Thrown when a method is unable to convert a string into a numeric format
12.	**StringIndexOutOfBoundsException**	Thrown when you access a String array with an illegal index.

A. Checked Exceptions:

- ✓ Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.
- ✓ Checked Exceptions forces programmers to deal with the exception that may be thrown.
- ✓ The compiler ensures whether the programmer handles the exception using try.. catch () block or not. The programmer should have to handle the exception; otherwise, compilation will fail and error will be thrown.

Example:

1. ClassNotFoundException
2. CloneNotSupportedException
3. IllegalAccessException,
4. MalformedURLException.
5. NoSuchFileException
6. NoSuchMethodException
7. IOException

Example Program: (Checked Exception)

`FileNotFoundException` is a checked exception in Java. Anytime, we want to read a file from filesystem, Java forces us to handle error situation where file may not be present in place.

Without try-catch

```
import java.io.*;

public class CheckedExceptionExample {
public static void main(String[] args)
{
    FileReader file = new FileReader("src/somefile.txt");
}
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
Unhandled exception type FileNotFoundException
```

To make program able to compile, you must handle this error situation in try-catch block. **Below given code will compile absolutely fine.**

With try-catch

```
import java.io.*;
public class CheckedExceptionExample {
public static void main(String[] args) {
try {
@SuppressWarnings("resource")
FileReader file = new FileReader("src/somefile.java");
System.out.println(file.toString());
}
catch(FileNotFoundException e){
System.out.println("Sorry...Requested resource not availabe...");
} }
}
```

Output:

```
Sorry...Requested resource not availabe...
```

B. Unchecked Exceptions(RunTimeException):

- ✓ The **unchecked** exceptions are just opposite to the **checked** exceptions.
- ✓ Unchecked exceptions are not checked at compile-time rather they are checked at runtime.
- ✓ The compiler doesn't force the programmers to either catch the exception or declare it in a throws clause.
- ✓ In fact, the programmers may not even know that the exception could be thrown.

Example:

1. ArrayIndexOutOfBoundsException
2. ArithmeticException
3. NullPointerException.

Example: Unchecked Exception

Consider the following Java program. It compiles fine, but it throws *ArithmeticException* when run. The compiler allows it to compile, because *ArithmeticException* is an unchecked exception.

```
class Main {
    public static void main(String args[]) {
        int x = 0;
        int y = 10;
        int z = y/x;
    }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Main.main(Main.java:5)
```

Example 1: NullPointer Exception

```
//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..
```

Example 2: NumberFormat Exception

```
// 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
```

3.5.2: USER-DEFINED EXCEPTIONS (CUSTOM EXCEPTIONS)

Exception types created by the user to describe the exceptions related to their applications are known as **User-defined Exceptions** or **Custom Exceptions.**

To create User-defined Exceptions:

1. Pick a self-describing ***Exception** class name.
2. Decide if the exception should be checked or unchecked.
   - ✓ Checked : **extends Exception**
   - ✓ Unchecked: **extends RuntimeException**
3. Define constructor(s) that call into super class constructor(s), taking message that can be displayed when the exception is raised.
4. Write the code that might generate the defined exception inside the try-catch block.
5. If the exception of user-defined type is generated, handle it using **throw clause** as follows:

   throw ExceptionClassObject;

Example:

The following program illustrates how user-defined exceptions can be created and thrown.

```
public class EvenNoException extends Exception
{
  EvenNoException(String str)
  {
    super(str);  // used to refer the superclass constructor
  }

  public static void main(String[] args)
  {
      int arr[]={2,3,4,5};
      int rem;
      int i;
      for(i=0;i<arr.length;i++)
      {
       rem=arr[i]%2;
       try
       {
```

```
            if(rem==0)
            {
               System.out.println(arr[i]+" is an Even Number");
            }
            else
            {
               EvenNoException exp=new EvenNoException(arr[i]+" is
               not an Even Number");
               throw exp;
            }
         }
          catch(EvenNoException exp)
           {
            System.out.println("Exception thrown is "+exp);
           }
      } // for loop
   } // main()
} // class
```

Output:

2 is an Even Number
Exception thrown is EvenNoException: 3 is not an Even Number
4 is an Even Number
Exception thrown is EvenNoException: 5 is not an Even Number

Program Explanation:

In the above program, the **EvenNumberException** class is created which inherits the **Exception** super class. Then the constructor is defined with the call to the super class constructor. Next, an array **arr** is created with four integer values. In the **main(),** the array elements are checked one by one for even number. If the number is odd, then the object of EvenNumberException class is created and thrown using **throw** clause. The EvenNumberException is handled in the catch block.

Comparison Chart - final Vs. finally Vs. finalize

Basis for comparison	final	finally	finalize
Basic	Final is a "Keyword" and "access modifier" in Java.	Finally is a "block" in Java.	Finalize is a "method" in Java.
Applicable	Final is a keyword applicable to classes, variables and methods.	Finally is a block that is always associated with try and catch block.	finalize() is a method applicable to objects.
Working	(1) Final variable becomes constant, and it can't be reassigned. (2) A final method can't be overridden by the child class. (3) Final Class can not be extended.	A "finally" block, clean up the resources used in "try" block.	Finalize method performs cleans up activities related to the object before its destruction.
Execution	Final method is executed upon its call.	"Finally" block executes just after the execution of"try-catch" block.	finalize() method executes just before the destruction of the object.

Example	class FinalExample{
	public static void main(String[] args){
	final int x=100;
	x=200;//Compile Time Error
	}}
		class FinallyExample{
		public static void main(String[] args){
		try{
		int x=300;
		}catch(Exception e){System.out.println(e);}
		finally{
		System.out.println("finally block is executed");
		}
		}}
			class FinalizeExample{
			public void finalize(){System.out.println("finalize called");}
			public static void main(String[] args){
			FinalizeExample f1=new FinalizeExample();
			FinalizeExample f2=new FinalizeExample();
			f1=null;
			f2=null;
			System.gc();
			}}

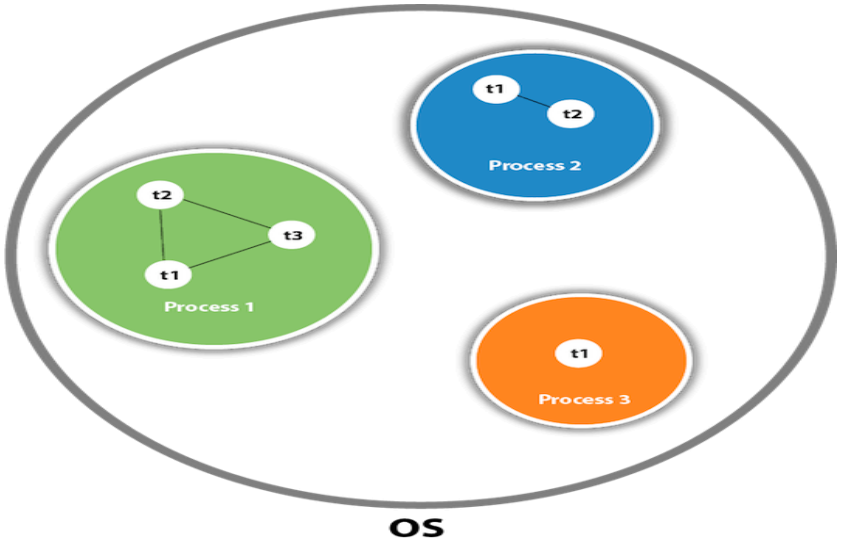
3.6: MULTITHREADED PROGRAMMING

3.6.1 : Introduction to Thread

Definition: Thread

A thread is a lightweight sub-process that defines a separate path of execution. It is the smallest unit of processing that can run concurrently with the other parts (other threads) of the same process.

- ✓ Threads are independent.
- ✓ If there occurs exception in one thread, it doesn't affect other threads.
- ✓ It uses a shared memory area.



- ✓ As shown in the above figure, a thread is executed inside the process.
- ✓ There is context-switching between the threads.
- ✓ There can be multiple processes inside the OS, and one process can have multiple threads.

DIFFERENCE BETWEEN THREAD AND PROCESS:

S.NO	PROCESS	THREAD
1)	Process is a heavy weight program	Thread is a light weight process
2)	Each process has a complete set of its own variables	Threads share the same data
3)	Processes must use IPC (Inter-Process Communication) to communicate with sibling processes	Threads can directly communicate with each other with the help of shared variables
4)	Cost of communication between	Cost of communication between

	processes is high.	threads is low.
5)	Process switching uses interface in operating system.	Thread switching does not require calling an operating system.
6)	Processes are independent of one another	Threads are dependent of one another
7)	Each process has its own memory and resources	All threads of a particular process shares the common memory and resources
8)	Creating & destroying processes takes more overhead	Takes less overhead to create and destroy individual threads

3.6.2 : MULTITHREADING

A program can be divided into a number of small processes. Each small process can be addressed as a single thread.

Definition: Multithreading

Multithreading is a technique of executing more than one thread, performing different tasks, simultaneously.

Multithreading enables programs to have more than one execution paths which executes concurrently. Each such execution path is a thread. For example, one thread is writing content on a file at the same time another thread is performing spelling check.

Advantages of Threads / Multithreading:

1. Threads are light weight compared to processes.
2. Threads share the same address space and therefore can share both data and code.
3. Context switching between threads is usually less expensive that between processes.
4. Cost of thread communication is low than inter-process communication.
5. Threads allow different tasks to be performed concurrently.
6. Reduces the computation time.
7. Through multithreading, efficient utilization of system resources can be achieved.

MULTITASKING

Definition: Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to maximize the utilization of CPU.

Multitasking can be achieved in two ways:

1) **Process-based Multitasking (Multiprocessing):-**
   - ❖ **It is a feature of executing two or more programs concurrently.**
   - ❖ For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor or visiting a web site.

2) **Thread-based Multitasking (Multithreading):-**
   - ❖ **It is a feature that a single program can perform two or more tasks simultaneously.**
   - ❖ For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

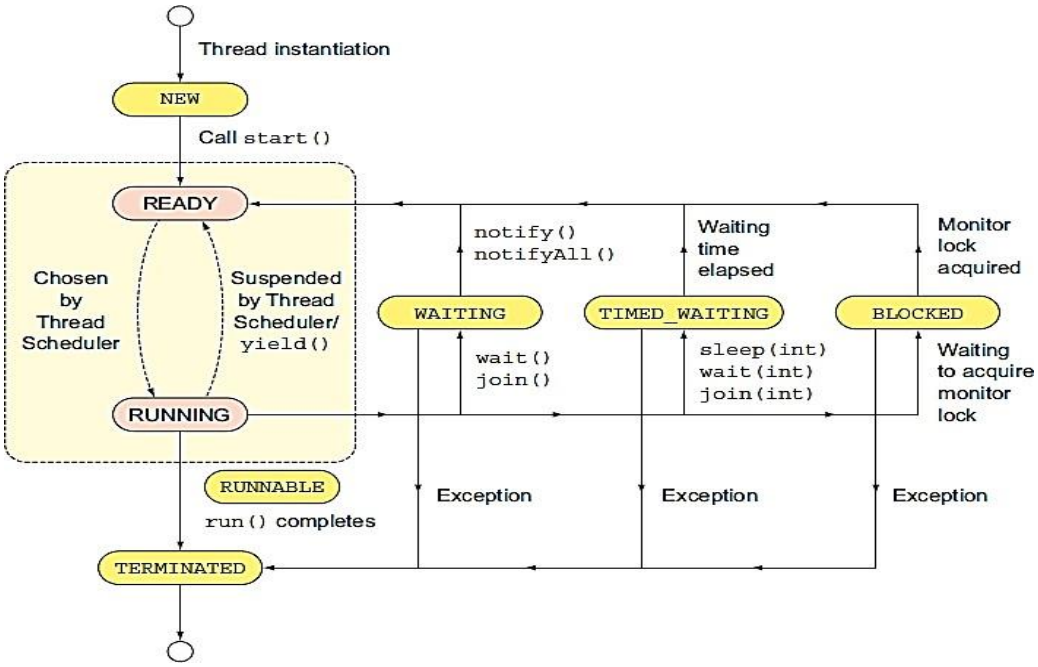
Differences between multi-threading and multitasking

Characteristics	Multithreading	Multitasking
Meaning	A process is divided into several different sub-processes called as threads, which has its own path of execution. This concept is called as multithreading.	The execution of more than one task simultaneously is called as multitasking.
Number of CPU	Can be one or more than one	One
Number of process being executed	Various components of the same process are being executed at a time.	One by one job is being executed at a time.
Number of users	Usually one.	More than one.
Memory Space	Threads are lighter weight. They share the same address space	Processes are heavyweight tasks that require their own separate address spaces.
Communication between units	Interthread communication is inexpensive	Interprocess communication is expensive and limited.
Context Switching	Context switching from one thread to the next is lower in cost.	Context switching from one process to another is also costly.

3.7: Thread Model / Thread Life Cycle (Different states of a Thread)

Different states, a thread (or applet/servlet) travels from its object creation to object removal (garbage collection) is known as life cycle of thread. A thread goes through various stages in its life cycle. At any time, a thread always exists in any one of the following state:

1. New State
2. Runnable State
3. Running State
4. Waiting/Timed Waiting/Blocked state
5. Terminated State/ dead state



1. New State:

A new thread begins its life cycle in the new state. It remains in this state until the program starts

the thread by calling **start()** method, which places the thread in the **runnable** state.

- ✓ A new thread is also referred to as a born thread.
- ✓ When the thread is in this state, only **start()** and **stop()** methods can be called. Calling any other methods causes an **IllegalThreadStateException**.
- ✓ Sample Code: **Thread myThread=new Thread();**

2. Runnable State:

After a newly born thread is started, the thread becomes runnable or running by calling the **run()** method.

- ✓ A thread in this state is considered to be executing its task.
- ✓ Sample code: **myThread.start();**
- ✓ The **start()** method creates the system resources necessary to run the thread, schedules the thread to run and calls the thread's run() method.

3. Running state:

- ✓ **Thread scheduler** selects thread to go from runnable to running state. In running state Thread starts executing by entering **run()** method.

- ✓ Thread scheduler selects thread from the runnable pool on basis of priority, if priority of two threads is same, threads are scheduled in unpredictable manner. Thread scheduler behaviour is completely unpredictable.
- ✓ When threads are in running state, **yield()** method can make thread to go in Runnable state.

4. Waiting/Timed Waiting/Blocked State :

❖ **Waiting State:**

Sometimes one thread has to undergo in waiting state because another thread starts executing. A runnable thread can be moved to a waiting state by calling the **wait**() method.

- ✓ A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- ✓ A call to **notify()** and **notifyAll()** may bring the thread from waiting state to runnable state.

❖ **Timed Waiting:**

A runnable thread can enter the timed waiting state for a specified interval of time by calling the **sleep()** method.

- ✓ After the interval gets over, the thread in waiting state enters into the runnable state.
- ✓ Sample Code:

```
try {
Thread.sleep(3*60*1000);// thread sleeps for 3 minutes
}
catch(InterruptedException ex) { }
```

❖ **Blocked State:**

When a particular thread issues an I/O request, then operating system moves the thread to

blocked state until the I/O operations gets completed.

- ✓ This can be achieved by calling **suspend()** method.
- ✓ After the I/O completion, the thread is sent back to the runnable state.

5. Terminated State:

A runnable thread enters the terminated state when,

(i) It completes its task (when the run() method has finished)

```
public void run() { }
```

(ii) Terminates (when the stop() is invoked) – **myThread.stop();**

A terminated thread cannot run again.

New : A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.
Runnable : After invocation of start() method on new thread, the thread becomes runnable.
Running : A thread is in running state if the thread scheduler has selected it.
Waiting : A thread is in waiting state if it waits for another thread to perform a task. In this stage the thread is still alive.
Terminated : A thread enter the terminated state when it complete its task.

THE "main" THREAD

The "main" thread is a thread that begins running immediately when a java program starts up. The "main" thread is important for two reasons:
1. It is the thread form which other child threads will be spawned.
2. It must be the last thread to finish execution because it performs various shutdown actions.

- ✓ Although the main thread is created automatically when our program is started, it can be controlled through a **Thread** object.
- ✓ To do so, we must obtain a reference to it by calling the method **currentThread()**.

Example:

```
class CurrentThreadDemo {
public static void main(String args[])
{
Thread t=Thread.currentThread();
System.out.println("Current Thread: "+t);

// change the name of the main thread
t.setName("My Thread");
System.out.println("After name change : "+t);

try {
for(int n=5;n>0;n--) {
System.out.println(n);
Thread.sleep(1000);// delay for 1 second
}
```

```
        } catch(InterruptedException e) {
              System.out.println("Main Thread Interrrupted");
        }
    }
    }
```

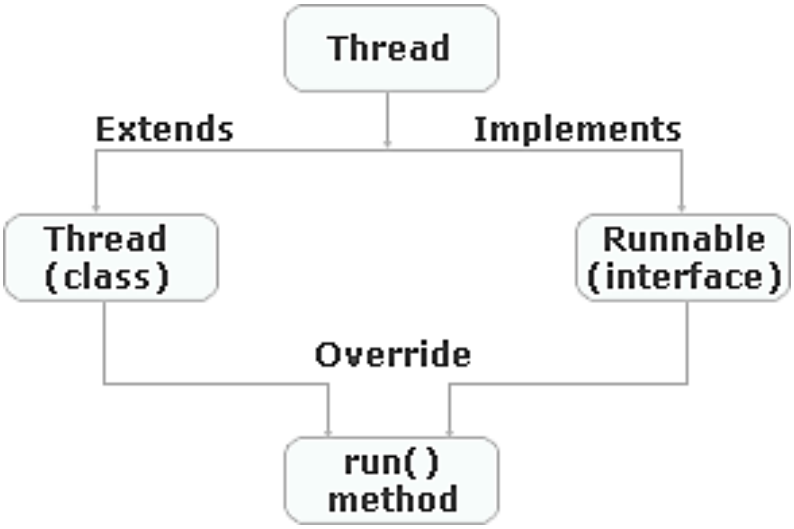
Output:

```
Current Thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

3.8: Creating Threads

We can create threads by instantiating an object of type **Thread.** Java defines two ways to create threads:

1. By implementing **Runnable** interface (**java.lang.Runnable)**
2. By extending the **Thread** class (**java.lang.Thread)**



1. Creating threads by implementing Runnable interface:

- The Runnable interface should be implemented by any class whose instances are intended to be executed as a thread.

- Implementing thread program using Runnable is preferable than implementing it by extending Thread class because of the following two reasons:
  1. If a class extends a Thread class, then it cannot extend any other class.
  2. If a class Thread is extended, then all its functionalities get inherited. This is an expensive operation.
- The Runnable interface has only one method that must be overridden by the class which implements this interface:

```
public void run()// run() contains the logic of the thread
{
// implementation code
}
```

- **Steps for thread creation:**

1. Create a class that implements **Runnable** interface. An object of this class is **Runnable** object.

```
public class MyThread implements Runnable
{
---
}
```

2. Override the **run()** method to define the code executed by the thread.
3. Create an object of type Thread by passing a Runnable object as argument.

```
Thread t=new Thread(Runnable threadobj, String threadName);
```

4. Invoke the **start()** method on the instance of the Thread class.

```
t.start();
```

- **Example:**

```
class MyThread implements Runnable
{

public void run()
{
for(int i=0;i<3;i++)
{
System.out.println(Thread.currentThread().getName()+" # Printing "+i);
try
{
Thread.sleep(1000);
}catch(InterruptedException e)
{
System.out.println(e);
```

```
    }
      }
}
}
public class RunnableDemo {
public static void main(String[] args)
{
MyThread obj=new MyThread();
MyThread obj1=new MyThread();
Thread t=new Thread(obj,"Thread-1");
t.start();
Thread t1=new Thread(obj1,"Thread-2");
t1.start();
}
}
```

Output:

```
Thread-0 # Printing 0
Thread-1 # Printing 0
Thread-1 # Printing 1
Thread-0 # Printing 1
Thread-1 # Printing 2
Thread-0 # Printing 2
```

2. Creating threads by extending Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

All the above constructors creates a new thread.

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.

3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public boolean isAlive():** tests if the thread is alive.
12. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
13. **public void suspend():** is used to suspend the thread(depricated).
14. **public void resume():** is used to resume the suspended thread(depricated).
15. **public void stop():** is used to stop the thread(depricated).
16. **public boolean isDaemon():** tests if the thread is a daemon thread.
17. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
18. **public void interrupt():** interrupts the thread.
19. **public boolean isInterrupted():** tests if the thread has been interrupted.
20. **public static boolean interrupted():** tests if the current thread has been interrupted.

- **<u>Steps for thread creation:</u>**

  1. Create a class that extends **java.lang.Thread** class.

  public class MyThread extends Thread

  {

  }

  2. Override the **run()** method in the sub class to define the code executed by the thread.

  3. Create an object of this sub class.

  MyThread t=new MyThread(String threadName);

  4. Invoke the **start()** method on the instance of the subclass to make the thread for running.

  start();

- **Example:**

```
class SampleThread extends Thread
{
public void run()
{
for(int i=0;i<3;i++)
{
System.out.println(Thread.currentThread().getName()+" # Printing "+i);
try
{
Thread.sleep(1000);
}catch(InterruptedException e)
{
System.out.println(e);
}
}
}
}

public class ThreadDemo {
public static void main(String[] args) {
SampleThread obj=new SampleThread();
obj.start();
SampleThread obj1=new SampleThread();
obj1.start();
}
}
```

Output:

```
Thread-0 # Printing 0
Thread-1 # Printing 0
Thread-1 # Printing 1
Thread-0 # Printing 1
Thread-0 # Printing 2
Thread-1 # Printing 2
```

3.9: THREAD PRIORITY

- ✓ Thread priority determines how a thread should be treated with respect to others.

- ✓ Every thread in java has some priority, it may be default priority generated by JVM or customized priority provided by programmer.
- ✓ Priorities are represented by a number between 1 and 10.
  1 – Minimum Priority5 – Normal Priority 10 – Maximum Priority
- ✓ Thread scheduler will use priorities while allocating processor. The thread which is having highest priority will get the chance first.
- ✓ **Thread scheduler** is a part of Java Virtual Machine (JVM). It decides which thread should execute first among two or more threads that are waiting for execution.
- ✓ It is decided based on the priorities that are assigned to threads. The thread having highest priority gets a chance first to execute.
- ✓ If two or more threads have same priorities, we can't predict the execution of waiting threads. It is completely decided by thread scheduler. It depends on the type of algorithm used by thread scheduler.
- ✓ Higher priority threads get more CPU time than lower priority threads.
- ✓ A higher priority thread can also preempt a lower priority thread. For instance, when a lower priority thread is running and a higher priority thread resumes (for sleeping or waiting on I/O), it will preempt the lower priority thread.
- ✓ If two or more threads have same priorities, we can't predict the execution of waiting threads. It is completely decided by thread scheduler. It depends on the type of algorithm used by thread scheduler.
- ✓ **3 constants defined in Thread class:**
  1. public static int MIN_PRIORITY
  2. public static int NORM_PRIORITY
  3. public static int MAX_PRIORITY
- ✓ Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.
- ✓ To set a thread's priority, use the **setPriority()** method.
- ✓ To obtain the current priority of a thread, use **getPriority()** method.

- ✓ **Example:**

```
class TestMultiPriority1 extends Thread{
 public void run(){
  System.out.println("running thread name is:"+Thread.currentThread().getNam
e());
  System.out.println("running thread priority is:"+
                                        Thread.currentThread().getPriority());

 }
```

```
public static void main(String args[]){
 TestMultiPriority1 m1=new TestMultiPriority1();
 TestMultiPriority1 m2=new TestMultiPriority1();
 m1.setPriority(Thread.MIN_PRIORITY);
 m2.setPriority(Thread.MAX_PRIORITY);
 m1.start();
 m2.start();
}
}
```

Output:

```
running thread name is:Thread-0
running thread priority is:10
running thread name is:Thread-1
running thread priority is:1
```

3.10: Thread Synchronization

Definition: Thread Synchronization

Thread synchronization is the concurrent execution of two or more threads that share critical resources.

When two or more threads need to use a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process of ensuring single thread access to a shared resource at a time is called **synchronization.**

Threads should be synchronized to avoid critical resource use conflicts. Otherwise, conflicts may arise when parallel-running threads attempt to modify a common variable at the same time.

✓ **Why use Synchronization**

The synchronization is mainly used to

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

✓ **Thread Synchronization**

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. Cooperation (Inter-thread communication in java)

✓ **Mutual Exclusive**

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

1. by synchronized method
2. by synchronized block

✓ **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.

1. Java 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.

Syntax to use synchronized method:

Access_modifier synchronized return_type method_name(parameters)
{ …….. }

Example of java 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
```

2. Synchronized block in java

- ✓ 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

1. **synchronized (object reference expression) {**
2. **//code block**
3. **}**

Example 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
```

Difference between synchronized method and synchronized block:

Synchronized method	Synchronized block
1. Lock is acquired on whole method. 2. Less preferred. 3. Performance will be less as compared to synchronized block.	1. Lock is acquired on critical block of code only. 2. Preferred. 3. Performance will be better as compared to synchronized method.

3.11: Inter-Thread Communication

Inter-Thread Communication or Co-operation is all about allowing synchronized threads to communicate with each other.

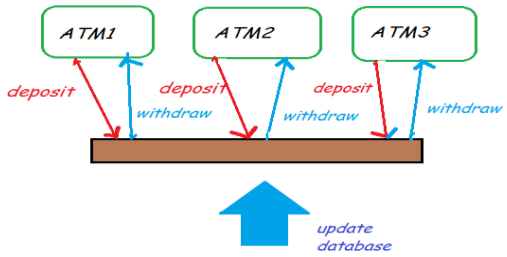
Definition: Inter-Thread Communication
Inter-thread communication is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

It is implemented by following methods of Object class and all these methods can be called only from within a synchronized context.

S.No.	Method & Description
1	**public final void wait() throws InterruptedException** Causes the current thread to wait until another thread invokes the notify().
2	**public final void wait(long timeout) throws InterruptedException** Causes current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed. **Parameters:** **timeout** – *the maximum time to wait in milliseconds.*
3	**public final void notify()** Wakes up a single thread that is waiting on this object's monitor.
4	**Public final void notifyAll()** Wakes up all the threads that called wait( ) on the same object.

Difference between wait() and sleep()		
Parameter	**wait()**	**sleep()**
Synchonized	wait should be called from synchronized context i.e. from block or method, If you do not call it using synchronized context, it will throw IllegalMonitorStateException	It need not be called from synchronized block or methods
Calls on	wait method operates on Object and defined in Object class	Sleep method operates on current thread and is in java.lang.Thread
Release of lock	wait release lock of object on which it is called and also other locks if it holds any	Sleep method does not release lock at all
Wake up condition	until call notify() or notifyAll() from Object class	Until time expires or calls interrupt()
static	wait is non-static method	sleep is static method

Example: The following program illustrates simple bank transaction operations with inter-thread communication:




```
class Customer{
int Balance=10000;

 synchronized void withdraw(int amount)
{
  System.out.println("going to withdraw..."+amount);

  if(Balance<amount)
  {
          System.out.println("Less balance;  Balance = Rs. "+Balance+"\nWaiting for
 deposit...\n");
          try
{
  wait();
                  }
 catch(Exception e){}
  }
```

```
  Balance-=amount;
  System.out.println("withdraw completed...");
}
 synchronized void deposit(int amount)
{
  System.out.println("going to deposit... Rs. "+amount);
  Balance+=amount;
  System.out.println("deposit completed... Balance = "+Balance);
  notify();
}
}

class ThreadCommn
{
  public static void main(String args[]) {
            Customer c=new Customer();
            new Thread()
            {
                        public void run(){c.withdraw(20000);}
            }.start();
            new Thread(){
                        public void run(){c.deposit(15000);}
            }.start();
  }
}
```

Output:

```
going to withdraw...20000
Less balance;  Balance = Rs. 10000
Waiting for deposit...

going to deposit... Rs. 15000
deposit completed... Balance = 25000
withdraw completed...
```

3.12: Suspending, Resuming and Stopping threads

The functions of Suspend, Resume and Stop a thread is performed using Boolean-type flags in a multithreading program. These flags are used to store the current status of the thread.

1. If the suspend flag is set to true, then run() will suspend the execution of the currently running thread.
2. If the resume flag is set to true, then run() will resume the execution of the suspended thread.
3. If the stop flag is set to true, then a thread will get terminated.

Example

```
class NewThread implements Runnable
{
   String name;     //name of thread
   Thread thr;
   boolean suspendFlag;
   boolean stopFlag;

   NewThread(String threadname)
   {
      name = threadname;
      thr = new Thread(this, name);
      System.out.println("New thread : " + thr);
      suspendFlag = false;
      stopFlag = false;
      thr.start();     // start the thread
   }

   /* this is the entry point for thread */
   public void run()
   {
      try
      {
         for(int i=1; i<10; i++)
         {
            System.out.println(name + " : " + i);
            Thread.sleep(1000);
```

```
            synchronized(this)
            {
               while(suspendFlag)
               {
                  wait();
               }
               if(stopFlag)
                  break;
            }
         }
      }
      catch(InterruptedException e)
      {
         System.out.println(name + " interrupted");
      }

      System.out.println(name + " exiting...");
   }

   synchronized void mysuspend()
   {
      suspendFlag = true;
   }

   synchronized void myresume()
   {
      suspendFlag = false;
      notify();
   }

    synchronized void mystop()
  {
   suspendFlag=false;
   stopFlag=true;
   notify();
   System.out.println("Thread "+name+" Stopped!!!");
  }

}
```

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

      NewThread obj1 = new NewThread("One");
      NewThread obj2 = new NewThread("two");

      try
      {
         Thread.sleep(1000);
         obj1.mysuspend();
         System.out.println("Suspending thread One...");
         Thread.sleep(1000);
         obj1.myresume();
         System.out.println("Resuming thread One...");

         obj2.mysuspend();
         System.out.println("Suspending thread Two...");
         Thread.sleep(1000);
         obj2.myresume();
         System.out.println("Resuming thread Two...");
         obj2.mystop();

      }
      catch(InterruptedException e)
      {
         System.out.println("Main thread Interrupted..!!");
      }

      System.out.println("Main thread exiting...");

   }
}
```

Output:

```
New thread : Thread[One,5,main]
New thread : Thread[two,5,main]
```

```
One : 1
two : 1
One : 2
Suspending thread One...
two : 2
two : 3
Resuming thread One...
One : 3
Suspending thread Two...
One : 4
Resuming thread Two...
two : 4
Thread two Stopped!!!
Main thread exiting...
two exiting...
One : 5
One : 6
One : 7
One : 8
One : 9
One exiting...
```

3.13: Wrappers

Wrappers

Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.

The table below shows the primitive type and the equivalent wrapper class:

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

3.13.1. Use of Wrapper classes

- ✓ **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- ✓ **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- ✓ **Synchronization:** Java synchronization works with objects in Multithreading.
- ✓ **java.util package:** The java.util package provides the utility classes to deal with objects.
- ✓ **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

Example:

```
//Java Program to convert all primitives into its corresponding
//wrapper objects and vice-versa
public class WrapperExample3{
public static void main(String args[]){
byte b=10;
short s=20;
int i=30;
long l=40;
float f=50.0F;
double d=60.0D;
char c='a';
boolean b2=true;

//Autoboxing: Converting primitives into objects
Byte byteobj=b;
Short shortobj=s;
Integer intobj=i;
Long longobj=l;
Float floatobj=f;
Double doubleobj=d;
Character charobj=c;
Boolean boolobj=b2;

//Printing objects
```

```
System.out.println("---Printing object values---");
System.out.println("Byte object: "+byteobj);
System.out.println("Short object: "+shortobj);
System.out.println("Integer object: "+intobj);
System.out.println("Long object: "+longobj);
System.out.println("Float object: "+floatobj);
System.out.println("Double object: "+doubleobj);
System.out.println("Character object: "+charobj);
System.out.println("Boolean object: "+boolobj);

//Unboxing: Converting Objects to Primitives
byte bytevalue=byteobj;
short shortvalue=shortobj;
int intvalue=intobj;
long longvalue=longobj;
float floatvalue=floatobj;
double doublevalue=doubleobj;
char charvalue=charobj;
boolean boolvalue=boolobj;

//Printing primitives
System.out.println("---Printing primitive values---");
System.out.println("byte value: "+bytevalue);
System.out.println("short value: "+shortvalue);
System.out.println("int value: "+intvalue);
System.out.println("long value: "+longvalue);
System.out.println("float value: "+floatvalue);
System.out.println("double value: "+doublevalue);
System.out.println("char value: "+charvalue);
System.out.println("boolean value: "+boolvalue);
}
}
```

Output

```
---Printing object values---
Byte object: 10
Short object: 20
Integer object: 30
Long object: 40
```

Float object: 50.0
Double object: 60.0
Character object: a
Boolean object: true
---Printing primitive values---
byte value: 10
short value: 20
int value: 30
long value: 40
float value: 50.0
double value: 60.0
char value: a
boolean value: true

3.14: Autoboxing

3.14. Autoboxing

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

Example:

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

Output

20 20 20

3.14.1. Unboxing

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

Example:

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

Output

3 3 3

A1: STACK TRACE ELEMENTS

A Stack Trace is a list of method calls from the point when the application was started to the current location of execution within the program. A Stack Trace is produced automatically by the Java Virtual Machine when an exception is thrown to indicate the location and progression of the program up to the point of the exception. They are displayed whenever a Java program terminates with an uncaught exception.

- ✓ We can access the text description of a stack trace by calling the **printStackTrace()** method of the **Throwable** class.
- ✓ The **java.lang.StackTraceElement is a** class where each element represents a single stack frame.
- ✓ We can call the **getStackTrace()** method to get an array of **StackTraceElement** objects that we want analyse in our program.

Class Declaration

Following is the declaration for **java.lang.StackTraceElement** class

public final class StackTraceElement extends **Object** implements **Serializable**

Class constructors

Constructor & Description
StackTraceElement(String declaringClass, String methodName, String fileName, int lineNumber) This creates a stack trace element representing the specified execution point.

Parameters:

- **declaringClass** – the fully qualified name of the class containing the execution point represented by the stack trace element.
- **methodName** – the name of the method containing the execution point represented by the stack trace element.
- **fileName** – the name of the file containing the execution point represented by the stack trace element, or null if this information is unavailable
- **lineNumber** – the line number of the source line containing the execution point represented by this stack trace element, or a negative number if this information is unavailable. A value of -2 indicates that the method containing the execution point is a native method.

Throws: NullPointerException – if declaringClass or methodName is null.

Methods in StackTraceElement class:

Method Name	Description
String getFileName()	Gets the name of the source file containing the execution point represented by the **StackTraceElement.**
int getLineNumber()	Gets the line number of the source file containing the execution point represented by the **StackTraceElement.**
String getClassName()	Gets the fully qualified name of the class containing the execution point represented by the **StackTraceElement.**
String getMethodName()	Gets the name of the method containing the execution point represented by the **StackTraceElement.**
boolean isNativeMethod()	Returns true if the execution point of the **StackTraceElement** is inside a native method.
String toString()	Returns a formatted string containing the class name, method name, file name and the line number, if available.

Example:

The following program for finding factorial(using recursion) prints the stack trace of a recursive factorial function.

```
import java.util.Scanner;

public class StackTraceTest
{
   public static int factorial(int n)
   {
      System.out.println(" Factorial ("+n+"):");
      Throwable t=new Throwable();
      StackTraceElement[] frames=t.getStackTrace();
      for(StackTraceElement f:frames)
      {
         System.out.println(f);
      }
      int r;
      if(n<=1)
         r=1;
      else
         r=n*factorial(n-1);
      System.out.println("return "+r);
      return r;
   }

   public static void main(String[] args)
   {
      Scanner in=new Scanner(System.in);
      System.out.println("Enter n: ");
      int n=in.nextInt();
      factorial(n);
   }
}
```

Output:

Enter n: 3

Factorial (3):
StackTraceTest.factorial(StackTraceTest.java:10)
StackTraceTest.main(StackTraceTest.java:30)
Factorial (2):
StackTraceTest.factorial(StackTraceTest.java:10)
StackTraceTest.factorial(StackTraceTest.java:20)
StackTraceTest.main(StackTraceTest.java:30)
Factorial (1):
StackTraceTest.factorial(StackTraceTest.java:10)
StackTraceTest.factorial(StackTraceTest.java:20)
StackTraceTest.factorial(StackTraceTest.java:20)
StackTraceTest.main(StackTraceTest.java:30)
return 1
return 2
return 6

A2: "assert" Keyword

Java assert **keyword is used to create assertions in Java, which enables us to test the assumptions about our program. For example, an assertion may be to make sure that an employee's age is positive number.**

Assertions are Boolean expressions that are used to test/validate the code. It is a statement in java that can be used to test your assumptions about the program.

- Assertion is achieved using "assert" keyword in java.
- While executing assertion, it is believed to be true. If it fails, JVM will throw an error named AssertionError. It is mainly used for testing purpose.

- **Following are the situations in which we can use the assertions:**

  - For making the program more readable and user friendly, the assert statements are used.
  - For validating the internal control flow and class invariant, the assertions are used.

- **Syntax of using Assertion:**

  There are two ways to use assertion.

  First way:

  1. **assert expression;**

Here the **Expression** is evaluated by the JVM and if any error occurs then **AssertionError** occurs.

Second way:

2. **assert expression1 : expression2;**

In this, Expression1 is evaluated and if it is false then the error message is displayed with the help of Expression2.

- **Assertion Enabling and Disabling:**

By default, assertions are disabled. They have to be enabled explicitly

For Enabling:

We can enable the assertions by running the java program with the -**enableassertions** (or) **-ea** option:

```
java –enableassertions AssertionDemo
        (or)
java –ea AssertionDemo
```

For Disabling: **-disableassertions** (or) **-da**

```
java –disableassertions AssertionDemo
        (or)
java –da AssertionDemo
```

When assertions are disabled, the class loader strips out the assertion code so that it won't slow excetuion.

Example:

```
// Java program to demonstrate syntax of assertion
import java.util.Scanner;

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

```
{
int value = 15;
assert value >= 20 : " Underweight";
System.out.println("value is "+value);
}
}
```

Output:

value is 15

After enabling assertions

Output:

Exception in thread "main" java.lang.AssertionError:
Underweight

- **Advantage of Assertions:**
  - It provides an effective way to detect and correct programming errors.

- **Where not to use Assertions**
  - Assertions should not be used to replace error messages
  - Do not use assertions for argument checking in public methods. Because if arguments are erroneous then that situation result in appropriate runtime exception such as **IllegalArgumentException, IndexOutOfBoundsException or NullPointerException.**

Unit – 4: I/O, GENERICS, STRING HANDLING

UNIT IV I/O, GENERICS, STRING HANDLING

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**.**
- ✓ These streams support all the types of objects, data-types, characters, files etc to fully execute the I/O operations.
- ✓ **The java.io package** contains all the classes required for input and output operations.

➢ **Java Input**

There are several ways to get input from the user in Java. To get input by using Scanner object, import Scanner class using:

import java.util.Scanner;

Then, we will create an object of Scanner class which will be used to get input from the user.

Scanner input = new Scanner (System.in);
int number = input.nextInt();

Example : Get Integer Input From the User

```
import java.util.Scanner;
class Input{
public static void main(String[] args){
    Scanner input =newScanner (System.in);
    System.out.print ("Enter an integer: ");
    int number =input.nextInt ();
```

```
        System.out.println ("You entered"+ number);
    }
    }
```

Output

Enter an integer: 23
You entered 23

- **Java Output**

Simply use System.out.println(), System.out.print() or System.out.printf() to send output to standard output (screen).system is a class and out is a public static field which accepts output data.

Example to output a line:

```
class Test
{
  public static void main(String[] args)
  {
        System.out.println("Java programming is interesting.");
  }
}
```

Output:

Java programming is interesting.

What's the difference between println (), print () and printf ()?

- print () - prints string inside the quotes.
- println () - prints string inside the quotes similar like print() method. Then the cursor moves to the beginning of the next line.
- printf () - it provides string formatting.

4.1.1: STREAMS

A **Stream** is a sequence of data or it is an abstraction that either produces or consumes information. In other simple words it is a flow of data from which you can read or write data to it. It's called a stream because it's like a stream of water that continues to flow.

➢ **PREDEFINED STREAMS:**

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

1) System.in: This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**. – It is an object of type InputStream.

2) System.out: This is used to output the data produced by the user's program and usually a computer screen is used to standard output stream and represented as **System.out**. - It is an object of type PrintStream

3) System.err: This is used to output the error data produced by the user's program and usually a computer screen is used to standard error stream and represented as **System.err**. - It is an object of type PrintStream

➢ **TYPES OF STREAMS:**

1. Byte Stream – Byte Streams provide a convenient means of handling input and output in terms of bytes. Byte streams are used when reading or writing binary data.

2. Character Stream – Character streams provide a convenient means of handling input or output in terms of characters. In some cases, character streams are more efficient than byte streams.

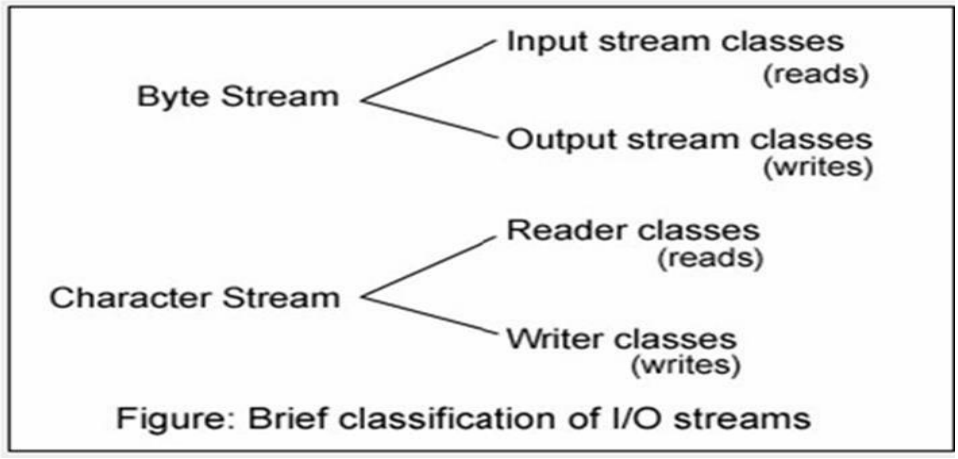


Figure: Brief classification of I/O streams

Some important Byte stream classes:

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
BufferedOutputStream	Used for Buffered Output Stream.
DataInputStream	Contains method for reading java standard datatype
DataOutputStream	An output stream that contain method for writing java standard data type
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that write to a file.
InputStream	Abstract class that describe stream input.
OutputStream	Abstract class that describe stream output.
PrintStream	Output Stream that contain print() and println() method

Some important Charcter stream classes.

Stream class	Description
BufferedReader	Handles buffered input stream.
BufferedWriter	Handles buffered output stream.
FileReader	Input stream that reads from file.
FileWriter	Output stream that writes to file.
InputStreamReader	Input stream that translate byte to character
OutputStreamReader	Output stream that translate character to byte.
PrintWriter	Output Stream that contain print() and println() method.
Reader	Abstract class that define character stream input
Writer	Abstract class that define character stream output

➢ INPUTSTREAM AND OUTPUTSTREAMS:

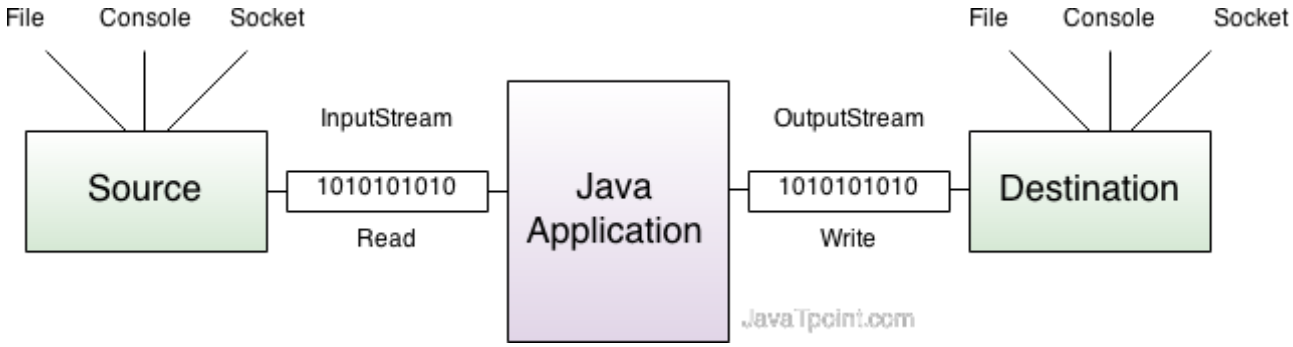
✓ **OutputStream**

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.

Working of Java OutputStream and InputStream by the figure given below.



OutputStream class

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

Commonly used methods of OutputStream class

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.

InputStream class

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

Commonly used methods of InputStream class

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.

1. FileInputStream and FileOutputStream (File Handling):

In Java, FileInputStream and FileOutputStream classes are used to read and write data in file. In another words, they are used for file handling in java.

- ✓ **FileOutputStream class**

Java FileOutputStream is an output stream for writing data to a file.

If you have to write primitive values then use FileOutputStream. Instead, for character-oriented data, prefer FileWriter. But you can write byte-oriented as well as character-oriented data.

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.
void close()	It is used to closes the file output stream.

Example of Java FileOutputStream class

```
1. import java.io.*;
2. class Test{
3.   public static void main(String args[]){
4.    try{
5.      FileOutputstream fout=new FileOutputStream("abc.txt");
6.      String s="java is my favourite language";
7.      byte b[]=s.getBytes();//converting string into byte array
8.      fout.write(b);
9.      fout.close();
10.     System.out.println("success...");
11.   }catch(Exception e){system.out.println(e);}
12.  }
13.}
```

Output:success...

✓ **FileInputStream class**

Java FileInputStream class obtains input bytes from a file.It is used for reading streams of raw bytes such as image data. For reading streams of characters, consider using FileReader.

It should be used to read byte-oriented data for example to read image, audio, video etc.

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.
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.

Example of FileInputStream class

```
1. import java.io.*;
2. class SimpleRead{
3.  public static void main(String args[]){
4.   try{
5.     FileInputStream fin=new FileInputStream("abc.txt");
6.     int i=0;
7.     while((i=fin.read())!=-1){
8.      System.out.println((char)i);
9.     }
10.    fin.close();
11.  }catch(Exception e){system.out.println(e);}
12. }
13.}
```

Output: java is my favourite language

2. BufferedOutputStream and BufferedInputStream

✓ **BufferedOutputStream class**

Java BufferedOutputStream class uses an internal buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

Constructor	Description
BufferedOutputStream(OutputStream os)	It creates the new buffered output stream which is used for writing the data to the specified output stream.
BufferedOutputStream(OutputStream os, int size)	It creates the new buffered output stream which is used for writing the data to the specified output stream with a specified buffer size.

Method	Description
void write(int b)	It writes the specified byte to the buffered output stream.
void write(byte[] b, int off, int len)	It write the bytes from the specified byte-input stream into a specified byte array, starting with the given offset
void flush()	It flushes the buffered output stream.

Example of BufferedOutputStream class:

In this example, we are writing the textual information in the BufferedOutputStream object which is connected to the FileOutputStream object. The flush() flushes the data of one stream and send it into another. It is required if you have connected the one stream with another.

```
1. import java.io.*;
2. class Test{
3.  public static void main(String args[])throws Exception{
4.    FileOutputStream fout=new FileOutputStream("f1.txt");
5.    BufferedOutputStream bout=new BufferedOutputStream(fout);
6.    String s="Java is my favourite language";
7.    byte b[]=s.getBytes();
8.    bout.write(b);
9.    bout.flush();
10.   bout.close();
11.   fout.close();
12.   System.out.println("success");
13. }
14.}
```

Output: success...

✓ **BufferedInputStream class**

Java BufferedInputStream class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.

Constructor	Description
BufferedInputStream(InputStream IS)	It creates the BufferedInputStream and saves it argument, the input stream IS, for later use.
BufferedInputStream(InputStream IS, int size)	It creates the BufferedInputStream with a specified buffer size and saves it argument, the input stream IS, for later use.

Method	Description
int available()	It returns an estimate number of bytes that can be read from the input stream without blocking by the next invocation method for the input stream.
int read()	It read the next byte of data from the input stream.

int read(byte[] b, int off, int ln)	It read the bytes from the specified byte-input stream into a specified byte array, starting with the given offset.
void close()	It closes the input stream and releases any of the system resources associated with the stream.
void reset()	It repositions the stream at a position the mark method was last called on this input stream.
void mark(int readlimit)	It sees the general contract of the mark method for the input stream.
long skip(long x)	It skips over and discards x bytes of data from the input stream.
boolean markSupported()	It tests for the input stream to support the mark and reset methods.

Example of Java BufferedInputStream

```
1. import java.io.*;
2. class SimpleRead{
3.  public static void main(String args[]){
4.   try{
5.     FileInputStream fin=new FileInputStream("f1.txt");
6.     BufferedInputStream bin=new BufferedInputStream(fin);
7.     int i;
8.     while((i=bin.read())!=-1){
9.      System.out.println((char)i);
10.   }
11.   bin.close();
12.   fin.close();
13.  }catch(Exception e){system.out.println(e);}
14. }
15.}
```

Output: Java is my favourite language

3. DataInputStream and DataOutputStream:

✓ **DataInputStream class**

DataInputStream class allows the programmer to read primitive data from the input source.

Method	Description
int read(byte[] b)	It is used to read the number of bytes from the input stream.
int readInt()	It is used to read input bytes and return an int value.
byte readByte()	It is used to read and return the one input byte.
char readChar()	It is used to read two input bytes and returns a char value.
double readDouble()	It is used to read eight input bytes and returns a double value.
boolean readBoolean()	It is used to read one input byte and return true if byte is non zero, false if byte is zero.
int skipBytes(int x)	It is used to skip over x bytes of data from the input stream.
void readFully(byte[] b)	It is used to read bytes from the input stream and store them into the buffer array.
void readFully(byte[] b, int off, int len)	It is used to read len bytes from the input stream.

✓ **DataOutputStream class**

The DataOutputStream stream let you write the primitives to an output source.

Example:

Following is the example to demonstrate DataInputStream and DataInputStream. This example reads 5 lines given in a file test.txt and converts those lines into capital letters and finally copies them into another file test1.txt.

Method	Description
int size()	It is used to return the number of bytes written to the data output stream.
void write(int b)	It is used to write the specified byte to the underlying output stream.
void writeChar(int v)	It is used to write char to the output stream as a 2-byte value.
void writeChars(String s)	It is used to write string to the output stream as a sequence of characters.

void writeByte(int v)	It is used to write a byte to the output stream as a 1-byte value.
void writeBytes(String s)	It is used to write string to the output stream as a sequence of bytes.
void writeInt(int v)	It is used to write an int to the output stream
void writeShort(int v)	It is used to write a short to the output stream.
void writeShort(int v)	It is used to write a short to the output stream.
void writeLong(long v)	It is used to write a long to the output stream.
void flush()	It is used to flushes the data output stream.

Test.txt

this is test 1 ,
this is test 2 ,
this is test 3 ,
this is test 4 ,
this is test 5 ,

test.java

```
import java.io.*;
public class Test{
  public static void main(String args[])throws IOException{
    DataInputStream d = new DataInputStream(new  FileInputStream("test.txt"));
    DataOutputStream out = new DataOutputStream(new FileOutputStream("test1.txt"));
    String count;
    while((count = d.readLine()) != null){
      String u = count.toUpperCase();
      System.out.println(u);
      out.writeBytes(u + "  ,");}
    d.close();
    out.close();
  }}
```

Output:

THIS IS TEST 1 ,
THIS IS TEST 2 ,
THIS IS TEST 3 ,
THIS IS TEST 4 ,
THIS IS TEST 5 ,

4. PrintStream

The **PrintStream** class provides methods to write data to another stream. The PrintStream class automatically flushes the data so there is no need to call flush() method. Moreover, its methods don't throw IOException.

Commonly used methods of PrintStream class:

There are many methods in PrintStream class. Let's see commonly used methods of PrintStream class:

- **public void print(boolean b):** it prints the specified boolean value.
- **public void print(char c):** it prints the specified char value.
- **public void print(char[] c):** it prints the specified character array values.
- **public void print(int i):** it prints the specified int value.
- **public void print(long l):** it prints the specified long value.
- **public void print(float f):** it prints the specified float value.
- **public void print(double d):** it prints the specified double value.
- **public void print(String s):** it prints the specified string value.
- **public void print(Object obj):** it prints the specified object value.
- **public void println(boolean b):** it prints the specified boolean value and terminates the line.
- **public void println(char c):** it prints the specified char value and terminates the line.
- **public void println(char[] c):** it prints the specified character array values and terminates the line.
- **public void println(int i):** it prints the specified int value and terminates the line.
- **public void println(long l):** it prints the specified long value and terminates the line.
- **public void println(float f):** it prints the specified float value and terminates the line.
- **public void println(double d):** it prints the specified double value and terminates the line.
- **public void println(String s):** it prints the specified string value and terminates the line./li>
- **public void println(Object obj):** it prints the specified object value and terminates the line.
- **public void println():** it terminates the line only.
- **public void printf(Object format, Object... args):** it writes the formatted string to the current stream.

- **public void printf(Locale l, Object format, Object... args):** it writes the formatted string to the current stream.
- **public void format(Object format, Object... args):** it writes the formatted string to the current stream using specified format.
- **public void format(Locale l, Object format, Object... args):** it writes the formatted string to the current stream using specified format.

Example of java.io.PrintStream class:

In this example, we are simply printing integer and string values.

```
1. import java.io.*;
2. class PrintStreamTest{
3.  public static void main(String args[])throws Exception{
4.    FileOutputStream fout=new FileOutputStream("mfile.txt");
5.    PrintStream pout=new PrintStream(fout);
6.    pout.println(1900);
7.    pout.println("Hello Java");
8.    pout.println("Welcome to Java");
9.    pout.close();
10.   fout.close();
11. }
12.}
```

Example of printf() method of java.io.PrintStream class:

Example of printing integer value by format specifier:

```
1. class PrintStreamTest{
2.  public static void main(String args[]){
3.    int a=10;
4.    System.out.printf("%d",a);//Note, out is the object of PrintStream class
5.  }
6. }
```

Output:10

➢ CHARACTER STREAMS (READER & WRITER):

Java IO's Reader and Writer work much like the InputStream and OutputStream with the exception that Reader and Writer are character based. They are intended for reading and writing text. The InputStream and OutputStream are byte based.

Reader class:

The **Java.io.Writer** class is a abstract class for writing to character streams.

Methods defined by Reader class:

Method	Description
abstract void close()	This method closes the stream and releases any system resources associated with it.
void mark(int numChars)	This method marks the present position in the stream.
boolean markSupported()	This method tells whether this stream supports the mark() operation.
int read()	This method reads a single character.
int read(char buffer[])	This method reads characters into an array.
abstract int read(char buffer[],int offset,int numChars)	This method reads characters into a portion of an array.
boolean ready()	This method tells whether this stream is ready to be read.
void reset()	This method resets the stream.
long skip(long numChars)	This method skips characters.

Writer class:

The Java.io.Writer class is a abstract class for writing to character streams

Methods defined by Writer class:

Method	Description
Writer append(char ch)	This method appends the specified character to this writer.
Writer append(CharSequence chars)	This method appends the specified character sequence to this writer.
Writer append(CharSequence chars, int begin, int end)	This method appends the specified character sequence to this writer.
abstract void close()	This method loses the stream, flushing it first.
abstract void flush()	This method flushes the stream.
void write(int ch)	This method writes a single character.
void write(char buffer[])	This method writes an array of characters.

1. Java FileWriter and FileReader (File Handling in java)

Java FileWriter and FileReader classes are used to write and read data from text files. These are character-oriented classes, used for file handling in java.
Java has suggested not to use the FileInputStream and FileOutputStream classes if you have to read and write the textual information.

➢ Java FileWriter class

Java FileWriter class is used to write character-oriented data to the file.
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
1) public void write(String text)	writes the string into FileWriter.
2) public void write(char c)	writes the char into FileWriter.
3) public void write(char[] c)	writes char array into FileWriter.
4) public void flush()	flushes the data of FileWriter.
5) public void close()	closes FileWriter.

➢ Java FileReader class

Java FileReader class is used to read data from the file. It returns data in byte format like FileInputStream class.

Constructors of FileWriter class

Constructor	Description
FileReader(String file)	It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.
FileReader(File file)	It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.

Methods of FileReader class

Method	Description
public int read()	returns a character in ASCII form. It returns -1 at the end of file.
public void close()	closes FileReader.

2. BufferedReader and BufferedWriter classes:

- **BufferedWriter class:**

This can be used for writing character data to the file.

Constructors

BufferedWriter bw = new BufferedWriter(writer w)
BufferedWriter bw = new BufferedWriter(writer r, int size)

BufferedWriter never communicates directly with the file. It should be communicate through some writer object only.

Important methods of BufferedWriter Class

void write(int ch) thorows IOException
void write(String s) throws IOException
void write(char[] ch) throws IOException
void newLine() for inserting a new line character.
void flush()
void close()

- **BufferedReader**

BufferedReader class can read character data from the file.

Constructors

1. BufferedReader br = new BufferedReader(Reader r)
2. BufferedReader br = new BufferedReader(Reader r, int buffersize)
3. BufferedReader never communicates directly with the file. It should Communicate through some reader object only.

Important methods of BufferedReader Class

1. int read()
2. int read(char [] ch)
3. String readLine(); - Reads the next line present in the file. If there is no nextline this method returns null.
4. void close()

Example for Java FileWriter and FileReader , BufferedReader and BufferedWriter classes:

```
import java.io.*;
class Simple{
public static void main(String args[]){
try{
        FileWriter fw=new FileWriter("d:/archana/abc.txt");
       BufferedWriter bw = new BufferedWriter(fw);
       bw.write(" Java");
        bw.close();
       fw.close();
        FileReader fr=new FileReader("d:/archana/abc.txt");
        BufferedReader br = new BufferedReader(fr);
         int i;
         while((i=br.read())!=-1)
         System.out.print((char)i);
         br.close();
         fr.close();
        }catch(Exception e){System.out.println(e);}
        System.out.println("success");
       } }
```

Output

Java
success

3. InputStreamReader and OutputStreamWriter classes:

- **OutputStreamWriter**

OutputStreamWriter behaves as a bridge to transfer data from character stream to byte stream. It uses default charset or we can specify charset for change in character stream to byte stream.

Constructors

1. OutputStreamWriter(OutputStream out)
2. OutputStreamWriter(OutputStream out, Charset cs)
3. OutputStreamWriter(OutputStream out, CharsetEncoder enc)
4. OutputStreamWriter(OutputStream out, String charsetName)

Important methods of OutputStreamWriter

1. void close()
2. void flush()
3. String getEncoding()
4. void write(int c)
5. void write(String str, int off, int len)

OutputStreamWriterDemo.java

```
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.Writer;
public class OutputStreamWriterDemo {
        public static void main(String[] args) {
    String str = "Hello World! \nThis is OutputStreamWriter Code Example."
    BufferedWriter bw = null;
    try {
     Writer w = new OutputStreamWriter(System.out);
     bw = new BufferedWriter(w);
     bw.write(str);
    } catch (IOException e) {
      e.printStackTrace();
    }finally {
       try {
          bw.close();
        } catch (IOException ex) {
          ex.printStackTrace();
        }
     }
  }
}
```

Output

Hello World!
This is OutputStreamWriter Code Example.

- **InputStreamReader**

InputStreamReader behaves as bridge from bytes stream to character stream. It also uses charset to decode byte stream into character stream.

Constructors

1. InputStreamReader(InputStream in_strm)
2. InputStreamReader(InputStream in_strm, Charset cs)
3. InputStreamReader(InputStream in_strm, CharsetDecoder dec)
4. InputStreamReader(InputStream in_strm, String charsetName)

Important methods of InputStreamReader

1. public boolean ready() – tells whether the character stream is ready to be read or not.
2. public void close() – closes InputStreamReader and releases all the Streams associated with it.
3. public int read() – returns single character after reading.
4. public String getEncoding() – returns the name of the character encoding being used by this stream.

InputStreamReaderDemo.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class InputStreamReaderDemo {
        public static void main(String[] args) {
                InputStreamReader isr = new InputStreamReader(System.in);
                BufferedReader br = new BufferedReader(isr);
                int a=0;
                int b=0;
                try {
                        System.out.println("Enter a number..");
                        a = Integer.parseInt(br.readLine());
                        System.out.println("Enter another number..");
                         b = Integer.parseInt(br.readLine());
                } catch (NumberFormatException e) {
                                e.printStackTrace();
                } catch (IOException e) {
                                e.printStackTrace();
                }
                System.out.println("you entered "+a+" and  "+b); }}
```

Output

Enter a number..
10
Enter another number..
14
you entered 10 and 14

4. PrintWriter Class

The **Java.io.PrintWriter** class prints formatted representations of objects to a text-output stream.

✓ **PrintWriter** defines several constructors. The one we will use is shown here:

PrintWriter(OutputStream *outputStream*, boolean *flushOnNewline*)

Here, *outputStream* is an object of type **OutputStream**, and *flushOnNewline* controls whether Java flushes the output stream every time a newline ('**\\n**') character is output. If *flushOnNewline* is **true**, flushing automatically takes place. If **false**, flushing is not automatic.

✓ **PrintWriter** supports the **print( )** and **println( )** methods for all types including **Object**. Thus, you can use these methods in the same way as they have been used with **System.out**. If an argument is not a simple type, the **PrintWriter** methods call the object's **toString( )** method and then print the result.

✓ To write to the console by using a **PrintWriter**, specify **System.out** for the output stream and flush the stream after each newline. For example, this line of code creates a **PrintWriter** that is connected to console output:

PrintWriter pw = new PrintWriter(System.out, true);

The following application illustrates using a **PrintWriter** to handle console output:

```
// Demonstrate PrintWriter
import java.io.*;
public class PrintWriterDemo {
public static void main(String args[]) {
PrintWriter pw = new PrintWriter(System.out, true);
pw.println("This is a string");
int i = -7;
pw.println(i);
double d = 4.5e-7;
pw.println(d); } }
```

The output from this program is shown here:

```
This is a string
-7
4.5E-7
```

4.2: READING AND WRITING CONSOLE

3 Ways to read input from console in Java

1. **Using Buffered Reader Class**

Advantages

The input is buffered for efficient reading.

Drawback:

The wrapping code is hard to remember.

2. **Using Scanner Class**

Advantages:

- Convenient methods for parsing primitives (nextInt(), nextFloat(), …) from the tokenized input.
- Regular expressions can be used to find tokens.

Drawback:

The reading methods are not synchronized

3. **Using Console Class**

Advantages:

- Reading password without echoing the entered characters.
- Reading methods are synchronized.
- Format string syntax can be used.

Drawback:

Does not work in non-interactive environment (such as in an IDE).

Java Console Class

The **Java Console class** is be used to get input from console. 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.

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

Java Console class declaration

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

```
1. import java.io.Console;
2. class ReadStringTest{
3. public static void main(String args[]){
4. Console c=System.console();
5. System.out.println("Enter your name: ");
```

```
6. String n=c.readLine();
7. System.out.println("Welcome "+n);
8. }
9. }
```

Output

```
Enter your name: abcd
Welcome abcd
```

Java Console Example to read password

```
1. import java.io.Console;
2. class ReadPasswordTest{
3. public static void main(String args[]){
4. Console c=System.console();
5. System.out.println("Enter password: ");
6. char[] ch=c.readPassword();
7. String pass=String.valueOf(ch);//converting char array into string
8. System.out.println("Password is: "+pass);
9. }
10.}
```

Output

```
Enter password:
Password is: 123
```

4.3: READING AND WRITING FILES

What is File Handling in Java?

- ✓ File handling in Java implies reading from and writing data to a file.
- ✓ The File class from the **java.io package**, allows us to work with different formats of files.
- ✓ In order to use the File class, you need to create an object of the class and specify the filename or directory name.

For example:

```
1)      // Import the File class
```

```
2)    import java.io.File
3)    // Specify the filename
4)    File obj = new File("filename.txt");
```

Java uses the concept of a stream to make I/O operations on a file.

The File class has many useful methods for creating and getting information about files. For example:

Method	Type	Description
canRead()	Boolean	Tests whether the file is readable or not
canWrite()	Boolean	Tests whether the file is writable or not
createNewFile()	Boolean	Creates an empty file
delete()	Boolean	Deletes a file
exists()	Boolean	Tests whether the file exists
getName()	String	Returns the name of the file
getAbsolutePath()	String	Returns the absolute pathname of the file
length()	Long	Returns the size of the file in bytes
list()	String[]	Returns an array of the files in the directory
mkdir()	Boolean	Creates a directory

➢ File Operations in Java

Basically, you can perform four operations on a file. They are as follows:

1) Create a File
2) Get File Information
3) Write To a File
4) Read from a File

1) Create a File

To create a file in Java, you can use the createNewFile() method. This method returns a boolean value: true if the file was successfully created, and false if the file already exists.

Example:

```
import java.io.File;
import java.io.IOException;

public class CreateFile {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      if (myObj.createNewFile()) {
        System.out.println("File created: " + myObj.getName());
      } else {
        System.out.println("File already exists.");
      }
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

The output will be:

```
File created: filename.txt
```

2) Write To a File

In the following example, we use the FileWriter class together with its write() method to write some text to the file we created in the example above. Note that when we are done writing to the file, we should close it with the close() method:

Example:

```
import java.io.FileWriter;   // Import the FileWriter class
import java.io.IOException;  // Import the IOException class to handle errors
public class WriteToFile {
  public static void main(String[] args) {
    try {
      FileWriter myWriter = new FileWriter("filename.txt");
      myWriter.write("Files in Java might be tricky, but it is fun enough!");
      myWriter.close();
```

```
      System.out.println("Successfully wrote to the file.");
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

Output:

Successfully wrote to the file.

3) Read a File

In the following example, we use the Scanner class to read the contents of the text file we created in the previous example:

Example:

```
import java.io.File;  // Import the File class
import java.io.FileNotFoundException;  // Import this class to handle errors
import java.util.Scanner; // Import the Scanner class to read text files
public class ReadFile {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      Scanner myReader = new Scanner(myObj);
      while (myReader.hasNextLine()) {
        String data = myReader.nextLine();
        System.out.println(data);
      }
      myReader.close();
    } catch (FileNotFoundException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }}}
```

Output:

Files in Java might be tricky, but it is fun enough!

4) Get File Information

To get more information about a file, use any of the File methods:

Example:

```
import java.io.File;  // Import the File class
public class GetFileInfo {
  public static void main(String[] args) {
    File myObj = new File("filename.txt");
    if (myObj.exists()) {
      System.out.println("File name: " + myObj.getName());
      System.out.println("Absolute path: " + myObj.getAbsolutePath());
      System.out.println("Writeable: " + myObj.canWrite());
      System.out.println("Readable " + myObj.canRead());
      System.out.println("File size in bytes " + myObj.length());
    } else {
      System.out.println("The file does not exist.");
    }
  }
}
```

Output:

```
File name: filename.txt
Absolute path: C:\Users\MyName\filename.txt
Writeable: true
Readable: true
File size in bytes: 0
```

4.4: Generic Programming

Generic programming is a style of computer programming in which algorithms are written in terms of "**to-be-specified-later"** types that are then instantiated when needed for specific types provided as parameters.

Generic programming refers to writing code that will work for many types of data.

NON-GENERICS:

In java, there is an ability to create generalized classes, interfaces and methods by operating through Object class.

Example:

```
class NonGen
{
   Object ob;
   NonGen(Object o)
   {
                                        ob=o;
   }
   Object getob()
   {
     return ob;
   }
   void showType()
   {
      System.out.println("Type of ob is "+ob.getClass().getName());
   }
}
public class NonGenDemo
{
   public static void main(String[] arg)
   {
      NonGen integerObj;
      integerObj=new NonGen(88);
      integerObj.showType();
      int v=(Integer)integerObj.getob(); // casting required
      System.out.println("Value = "+v);
      NonGen strObj=new NonGen("Non-Generics Test");
      strObj.showType();
      String str=(String)strObj.getob(); // casting required
      System.out.println("Vlaue = "+str);
   }
}
```

Output:

```
Type of ob is java.lang.Integer
Value = 88
Type of ob is java.lang.String
Vlaue = Non-Generics Test
```

Limitation of Non-Generic:

1) Explicit casts must be employed to retrieve the stored data.
2) Type mismatch errors cannot be found until run time.

Need for Generic:

1) It saves the programmers burden of creating separate methods for handling data belonging to different data types.
2) It allows the code reusability.
3) Compact code can be created.

Advantage of Java Generics (Motivation for Java Generics):

1) **Code Reuse:** We can write a method/class/interface once and use for any type we want.
2) **Type-safety :** We can hold only a single type of objects in generics. It doesn't allow to store other objects.
3) **Elimination of casts:** There is no need to typecast the object.

   The following code snippet without generics requires casting:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
```

   When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

4) **Stronger type checks at compile time:**

   A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);                    //Compile Time Error
```

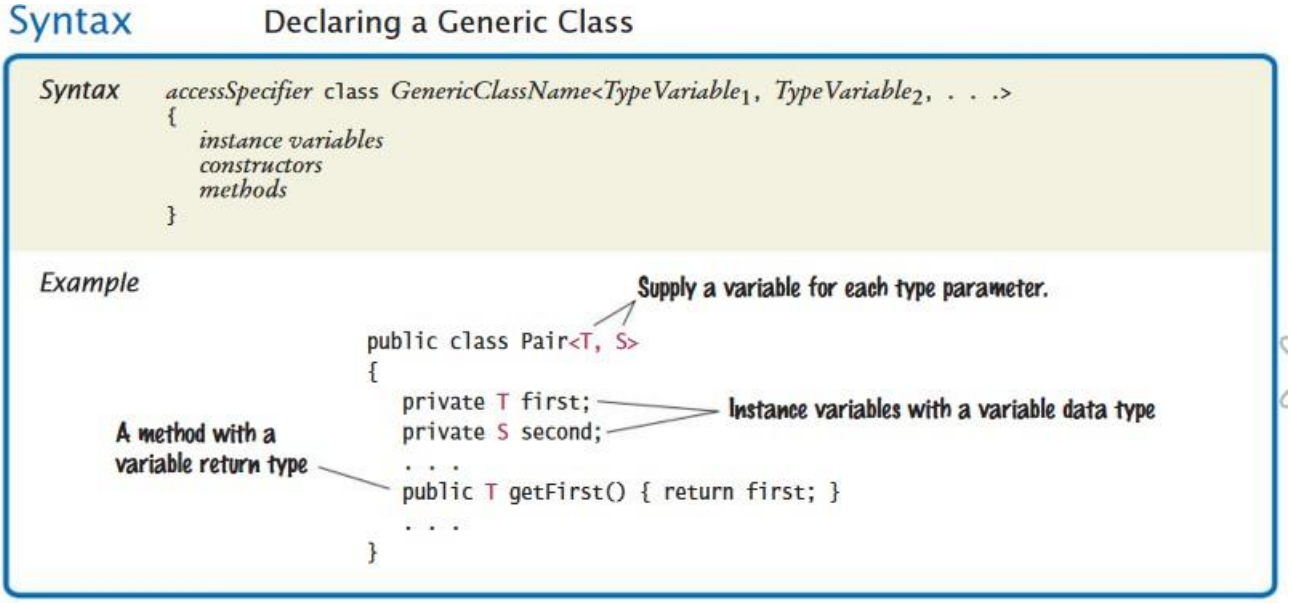
5) **Enabling programmers to implement generic algorithms.**

   By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

4.5: GENERIC CLASSES

A class that can refer to any type is known as generic class. Here, we are using T type parameter to create the generic class of specific type.

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.



Where, the type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the *type parameters* (also called *type variables*)

Example:

```
public class Pair<T, S>
{
. . .
}
```

Purpose: To define a generic class with methods and fields that depends on type variables.

Class reference declaration:

To instantiate this class, use the new keyword, as usual, but place **<type_parameter>** between the class name and the parenthesis:

class_name<type-arg-list> var-name=new class_name<type-arg-list>(cons-arg-list);

Type Parameter Naming Conventions:

- ✓ Type parameter is a place holder for a type argument.
- ✓ By convention, type parameter names are single, uppercase letters.

The most commonly used type parameter names are:

- ❖ E - Element (used extensively by the Java Collections Framework)
- ❖ K - Key
- ❖ N - Number
- ❖ T - Type
- ❖ V - Value
- ❖ S,U,V etc. - 2nd, 3rd, 4th types

Example: Generic class with single type parameter

```
class Gen <T>
{
T obj;
Gen(T x)
{
    obj= x;
}

T show()
{
    return obj;
}

void disp()
{
    System.out.println(obj.getClass().getName());
}
}

public class Test
{
public static void main (String[] args)
{
  Gen < String> ob = new Gen<>("java programming with Generics");
```

```
  ob.disp();
  System.out.println("value :  " +ob.show());

  Gen < Integer> ob1 = new Gen<>(550);
  ob1.disp();
  System.out.println("value  :" +ob1.show());
}
}
```

Output:

```
java.lang.String
value :  java programming with Generics
java.lang.Integer
value  :550
```

Example: Generic class with more than one type parameter

In Generic parameterized types, we can pass more than 1 data type as parameter. It works the same as with one parameter Generic type.

```
class Gen <T1,T2>
{
T1 obj1;
T2 obj2;
Gen(T1 o1,T2 o2)
{
  obj1 = o1;
  obj2 = o2;
}
 T1 get1()
{
  return obj1;
}
T2 get2()
{
     return obj2;
}
void disp()
{
   System.out.println(obj1.getClass().getName());
```

```
   System.out.println(obj2.getClass().getName());
}
}

public class Test
{
public static void main (String[] args)
{
  Gen < String, Integer> obj = new Gen<>("java programming with Generics",560);
  obj.disp();
  System.out.println("value 1 :  " +obj.get1());
  System.out.println("value 2:  "+obj.get2());

Gen < Integer, Integer> obje = new Gen<>(1000,560);
  obje.disp();
  System.out.println("value 1 :  " +obje.get1());
  System.out.println("value 2:  "+obje.get2());
}
}
```

Output:

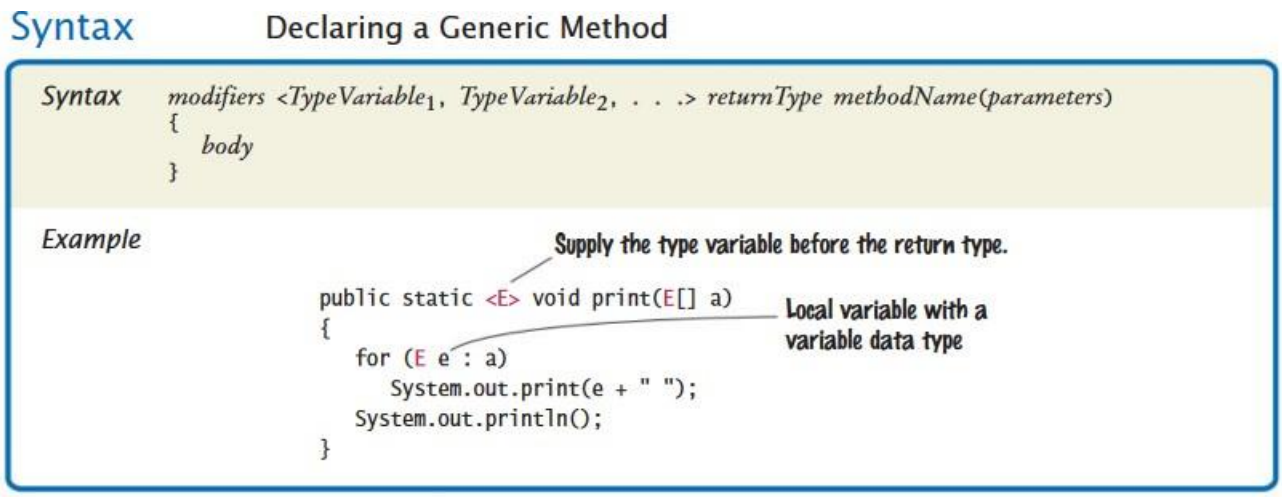
```
java.lang.String
java.lang.Integer
value 1 :  java programming with Generics
value 2:  560
java.lang.Integer
java.lang.Integer
value 1 :  1000
value 2:  560
```

4.6: GENERIC METHODS

A Generic Method is a method with type parameter. 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.

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.
- ✓ 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: (To iterate through the list and display the element using generic method)

```
class a < T >
{
        <T> void show(T[] el)
        {
          for(T x:el)
     System.out.println(x);
  }
 }
```

```
public class GenMethod
{
   public static void main(String arg[])
   {
                System.out.println("Integer array");
                a<Integer> o1=new a<Integer>();
                Integer[] ar={10,67,23};
                o1.show(ar);

                System.out.println("String array");
                a<String> o2=new a<String>();
                String[] ar1={"Hai","Hello","Welcome","to","Java programming"};
                o2.show(ar1);

              System.out.println("Boolean array");
              a<Boolean> o3=new a<Boolean>();
              Boolean[] ar2={true,false};
              o3.show(ar2);

              System.out.println("Double array");
              a<Double> o4=new a<Double>();
              Double[] ar3={10.234,67.451,23.90};
              o4.show(ar3);
   }
}
```

Output:

```
Integer array
10
67
23
String array
Hai
Hello
Welcome
to
Java  programming
```

Boolean array
true
false
Double array
10.234
67.451
23.9

4.7: GENERICS WITH BOUNDED TYPES

GENERICS WITH BOUNDED TYPE PARAMETERS:

Bounded Type Parameter is a type parameter with one or more bounds. The bounds restrict the set of types that can be used as type arguments and give access to the methods defined by the bounds.

For example, a method that operates on numbers might only want to accept instances of Number or its subclasses.

Syntax:

<T extends superclass>

Example:

The following example creates a generic class that contains a method that returns the average of array of any type of numbers. The type of the numbers is represented generically using Type Parameter.

```
public class GenBounds<T extends Number>
{
  T[]  nums;
  GenBounds(T[] obj)
  {
    nums=obj;
  }
  double average()
  {
    double sum=0.0;
    for(int i=0;i<nums.length;i++)
```

```
        sum+=nums[i].doubleValue();
      double avg=sum/nums.length;
      return avg;
   }
   public static void main(String[] args)
   {
      Integer inum[]={1,2,3,4,5};
      GenBounds<Integer> iobj=new GenBounds<Integer>(inum);
      System.out.println("Average of Integer Numbers : "+iobj.average());

      Double dnum[]={1.1,2.2,3.3,4.4,5.5};
      GenBounds<Double> dobj=new GenBounds<Double>(dnum);
      System.out.println("Average of Double Numbers : "+dobj.average());

       /* Error: java,lang.String not within bound
      String snum[]={"1","2","3","4","5"};
      GenBounds<String> sobj=new GenBounds<String>(snum);
      System.out.println("Average of Integer Numbers : "+iobj.average()); */
   }
}
```

Output:

```
F:\>java GenBounds
Average of Integer Numbers : 3.0
Average of Double Numbers : 3.3
```

Wild Card Arguments:

Question mark (?) is the wildcard in generics and represents an unknown type. The wildcard can be used as the type of a parameter, field, or local variable and sometimes as a return type.

Name	Syntax	Meaning
Wildcard with lower bound	`? extends B`	Any subtype of `B`
Wildcard with upper bound	`? super B`	Any supertype of `B`
Unbounded wildcard	`?`	Any type

Example: BOUNDED WILDCARDS:-

A bounded wildcard is a wildcard with either an upper or a lower bound.

The following program illustrates the use of wildcards with upper bound. In below method we can use all the methods of upper bound class Number.

```
import java.util.ArrayList;
import java.util.List;
public class GenericsWildcards
{
    public static void main(String[] args)
    {
        List<Integer> ints = new ArrayList<Integer>();
                    ints.add(3);
    ints.add(5);
    ints.add(10);
        double sum = sum(ints);
        System.out.println("Sum of ints="+sum);
    }

    // here Number is the upper bound for the type parameter
    public static double sum(List<? extends Number>  list)
    {
        double sum = 0;
        for(Number n : list)
    {
                    sum += n.doubleValue();
        }
        return sum;
    }
}
```

Output:

```
F:\>java GenericsBounds
Sum of ints=18.0
```

Example: UNBOUNDED WILDCARD:-

Sometimes we have a situation where we want our generic method to be working with all types; in this case unbounded wildcard can be used. The wildcard "?" simply matches any valid objects.

- ✓ Its same as using <? extends Object>.

```
import java.util.*;
public class GenUBWildcard
{
   public static void main(String[] args)
   {
     List<Integer> ints = new ArrayList<Integer>();
         ints.add(3);
        ints.add(5);
        ints.add(10);
      printData(ints);

     List<String> str = new ArrayList<String>();
         str.add("\nWelcome");
        str.add(" to ");
        str.add(" JAVA ");
      printData(str);
   }
   public static void printData(List<?> list)
   {
         for(Object obj : list)
           {
                 System.out.print(obj + "\n");
         }}}
```

Output:

```
F:\>java GenUBWildcard
3
5
10
Welcome
 to
 JAVA
```

4.8: RESTRICTIONS AND LIMITATIONS OF GENERICS

1) In Java, generic types are compile time entities. The runtime execution is possible only if it is used along with raw type.
2) Primitive type parameters are not allowed for generic programming.
   For example:
   Stack<int> is not allowed.
3) For the instances of generic class throw and catch keywords are not allowed.
   For example:
   public class Test<T> extends Exception
   {
   // code // Error: can't extend the Exception class
   }
4) Instantiation of generic parameter T is not allowed.
   For Example:
   new T(); // Error
   new T[10]; // Error
5) Arrays of parameterized types are not allowed.
   For Example:
   New Stack<String>[10]; // Error

4.9: STRINGS

Definition:

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

How to create String object?
There are two ways to create a String object:

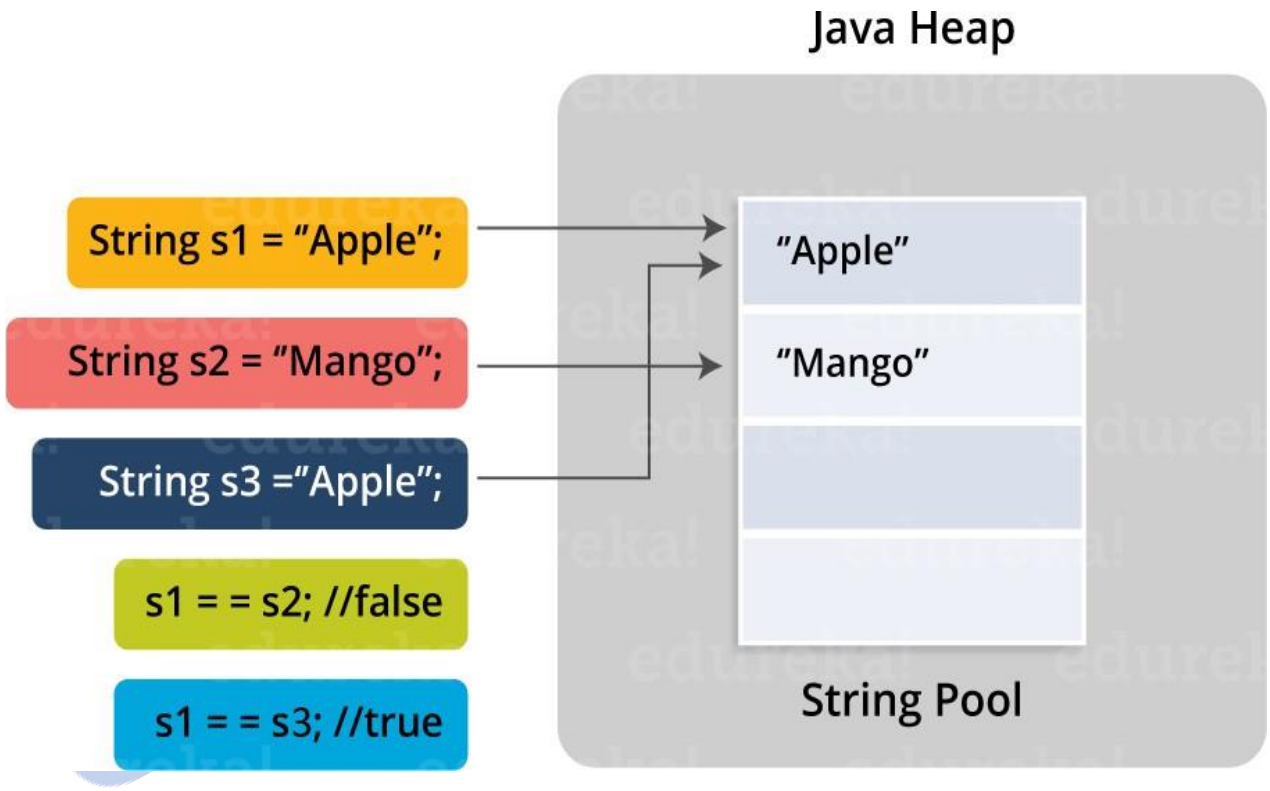
1. **By string literal**: Java String literal is created by using double quotes. For Example: String s="Welcome";

2. **By new keyword**: Java String is created by using a keyword "new".
   For example: String s=new String("Welcome");
   It creates two objects (in String pool and in heap) and one reference variable where the variable 's' will refer to the object in the heap.

Java String Pool:

Java String pool refers to collection of Strings which are stored in heap memory.
In this, whenever a new object is created,

1) String pool first checks whether the object is already present in the pool or not.
2) If it is present, then same reference is returned to the variable
3) else new object will be created in the String pool and the respective reference will be returned.



Refer to the diagrammatic representation for better understanding: In the above image, two Strings are created using literal i.e "Apple" and "Mango". Now, when third String is created with the value "Apple", instead of creating a new object, the already present object reference is returned.

Example: Creating Strings

```
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:
java
strings
example

Immutable String in Java

In java, **string objects are immutable**. Immutable simply means un-modifiable or unchangeable.

- ✓ Once string object is created its data or state can't be changed but a new string object is created.
- ✓ Let's try to understand the immutability concept by the example given below:

```
class Simple{
 public static void main(String args[]){
  String s="Sachin";
  s.concat(" Tendulkar");//concat() method appends the string at the end
  System.out.println(s);//will print Sachin because strings are immutable objec
ts
 }
}
```

Output: Sachin

4.10: METHODS

Methods of String class in Java

java.lang.String class provides a lot of methods to work on string. By the help of these methods, we can perform operations on string such as trimming, concatenating, converting strings etc.

Important methods of String class.

S.No.	Method	Description
1)	public boolean equals(Object anObject)	Compares this string to the specified object.
2)	public boolean equalsIgnoreCase(String another)	Compares this String to another String, ignoring case.
3)	public String concat(String str)	Concatenates the specified string to the end of this string.
4)	public int compareTo(String str)	Compares two strings and returns int
5)	public int compareToIgnoreCase(String str)	Compares two strings, ignoring case differences.
6)	public String substring(int beginIndex)	Returns a new string that is a substring of this string.
7)	public String substring(int beginIndex,int endIndex)	Returns a new string that is a substring of this string.
8)	public String toUpperCase()	Converts all of the characters in this String to upper case
9)	public String toLowerCase()	Converts all of the characters in this String to lower case.
10)	public String trim()	Returns a copy of the string, with leading and trailing whitespace omitted.
11)	public boolean startsWith(String prefix)	Tests if this string starts with the specified prefix.
12)	public boolean endsWith(String suffix)	Tests if this string ends with the specified suffix.

13)	public char charAt(int index)	Returns the char value at the specified index.
14)	public int length()	Returns the length of this string.
15)	public String intern()	Returns a canonical representation for the string object.
16)	public byte[] getBytes()	Converts string into byte array.
17)	public char[] toCharArray()	Converts string into char array.
18)	public static String valueOf(int i)	converts the int into String.
19)	public static String valueOf(long i)	converts the long into String.
20)	public static String valueOf(float i)	converts the float into String.
21)	public static String valueOf(double i)	converts the double into String.
22)	public static String valueOf(boolean i)	converts the boolean into String.
23)	public static String valueOf(char i)	converts the char into String.
24)	public static String valueOf(char[] i)	converts the char array into String.
25)	public static String valueOf(Object obj)	converts the Object into String.
26)	public void replaceAll(String firstString,String secondString)	Changes the firstString with secondString.

- **<u>String comparison in Java</u>**

  - ✓ We can compare two given strings on the basis of content and reference.
  - ✓ It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.
  - ✓ There are three ways to compare String objects:

1. By equals() method
2. By = = operator
3. By compareTo() method

1) By equals() method

equals() method compares the original content of the string.It compares values of string for equality.String class provides two methods:

- **public boolean equals(Object another){}** compares this string to the specified object.
- **public boolean equalsIgnoreCase(String another){}** compares this String to another String, ignoring case.

Example: equals() method

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

  String s1="Sachin";
  String s2="Sachin";
  String s3=new String("Sachin");
  String s4="Saurav";

  System.out.println(s1.equals(s2));//true
  System.out.println(s1.equals(s3));//true
  System.out.println(s1.equals(s4));//false
 }
}
```

Output:

```
true
true
false
```

Example: equalsIgnoreCase(String) method

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

  String s1="Sachin";
  String s2="SACHIN";

  System.out.println(s1.equals(s2));//false
  System.out.println(s1.equalsIgnoreCase(s3));//true
 }
}
```

Output:

```
false
 true
```

2) By == operator

The = = operator compares references not values.

Example: == operator:

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

  String s1="Sachin";
  String s2="Sachin";
  String s3=new String("Sachin");

  System.out.println(s1==s2);//true (because both refer to same instance)
  System.out.println(s1==s3);//false(because s3 refers to instance created in nonpoo
l)
 }

}
```

Output:

true
false

3) By compareTo() method:

compareTo() method compares values and returns an int which tells if the values compare less than, equal, or greater than.

Suppose s1 and s2 are two string variables.If:

- **s1 == s2** :0
- **s1 > s2** :positive value
- **s1 < s2** :negative value

Example: compareTo() method:

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

  String s1="Sachin";
  String s2="Sachin";
  String s3="Ratan";
```

```
  System.out.println(s1.compareTo(s2));//0
  System.out.println(s1.compareTo(s3));//1(because s1>s3)
  System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
 }
}
```

Output:

```
0
1
-1
```

- **<u>String Concatenation in Java</u>**

Concating strings form a new string i.e. the combination of multiple strings.

There are two ways to concat string objects:

1. By + (string concatenation) operator
2. By concat() method

1) By + (string concatenation) operator

String concatenation operator is used to add strings.For Example:

```
//Example of string concatenation operator

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

  String s="Sachin"+" Tendulkar";
  System.out.println(s);//Sachin Tendulkar
 }
}
```

Output: Sachin Tendulkar

The compiler transforms this to:

```
String s=(new StringBuilder()).append("Sachin").append(" Tendulkar).toString();
```

String concatenation is implemented through the StringBuilder(or StringBuffer) class and its append method. String concatenation operator produces a new string by appending the second operand onto the end of the first operand. The string concatenation operator can concat not only string but primitive values also. For

Example:

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

  String s=50+30+"Sachin"+40+40;
  System.out.println(s);//80Sachin4040
 }
}
```

Output: 80Sachin4040

Note: If either operand is a string, the resulting operation will be string concatenation. If both operands are numbers, the operator will perform an addition.

2) By concat() method

concat() method concatenates the specified string to the end of current string.

Syntax: public String concat(String another){}

Example of concat(String) method

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

  String s1="Sachin ";
  String s2="Tendulkar";

  String s3=s1.concat(s2);

  System.out.println(s3);//Sachin Tendulkar
 }
}
```

Output: Sachin Tendulkar

- **Example Program: Using all the methods of String class**

```
class Simple{
public static void main(String args[])
{
String s="Sachin Tendulkar";
System.out.println("Substring 1: "+s.substring(6));
System.out.println("Substring2: "+s.substring(0,6));
System.out.println("Uppercase: "+s.toUpperCase());
System.out.println("Lowercase: "+s.toLowerCase());
```

```
System.out.println("Trim: "+s.trim());
System.out.println("Start With: "+s.startsWith("Sa"));
System.out.println("End with: "+s.endsWith("n"));
System.out.println("Char at Position 0: "+s.charAt(0));
System.out.println("Char at Position 3: "+s.charAt(3))
System.out.println("Length: "+s.length());
String s2=s.intern();
System.out.println("Intern: "+s2);
System.out.println("Replace: "+s.replace('a','q'));
System.out.println("Index 1: "+s.indexOf('I'));
System.out.println("Index 2: "+s.indexOf('I',5));
 }
}
```

```
Substring 1: Tendulkar
Substring2: Sachin
Uppercase: SACHIN TENDULKAR
Lowercase: sachin tendulkar
Trim: Sachin Tendulkar
Start With: true
End with: false
Char at Position 0: S
Char at Position 3: h
Length: 16
Intern: Sachin Tendulkar
Replace: Sqchin Tendulkqr
Index 1: -1
Index 2: -1
```

4.11: STRING BUFFER CLASS

❖ StringBuffer CLASS

The StringBuffer class is used to created mutable (modifiable) string. The StringBuffer class is same as String except it is mutable i.e. it can be changed.
StringBuffer can be changed dynamically. String buffers are preferred when heavy modification of character strings is involved (appending, inserting, deleting, modifying etc).

Difference between String class and StringBuffer class:

S.No	String Class	StingBuffer Class
1	String objects are constants and immutable(cannot change the content)	StringBuffer objects are not constants and mutable(can change the content)
2	String class supports constant strings	StringBuffer class supports growable and modifiable strings
3	The methods in the String class are not synchronized	The methods of the StringBuffer class can be synchronized

Important Constructors of StringBuffer class

Constructor	Description
StringBuffer()	creates an empty string buffer with the initial capacity of 16. **Example:** StringBuffer s=**new** StringBuffer();
StringBuffer(String str)	creates a string buffer with the specified string. **Example:** StringBuffer s=**new** StringBuffer("Alice in Wonderland");
StringBuffer(int capacity)	creates an empty string buffer with the specified capacity as length. **Example:** StringBuffer s=**new** StringBuffer(20);

Important methods of StringBuffer class

Modifier and Type	Method	Description
public synchronized StringBuffer	append(String s)	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)	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)	is used to replace the string from specified startIndex and endIndex.
public synchronized StringBuffer	delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex.
public synchronized StringBuffer	reverse()	is used to reverse the string.
public int	capacity()	is used to return the current capacity.

public void	ensureCapacity(int minimumCapacity)	is used to ensure the capacity at least equal to the given minimum.
public char	charAt(int index)	is used to return the character at the specified position.
public int	length()	is used to return the length of the string i.e. total number of characters.
public String	substring(int beginIndex)	is used to return the substring from the specified beginIndex.
public String	substring(int beginIndex, int endIndex)	is used to return the substring from the specified beginIndex and endIndex.

Example:

```
public class StringBufferFunctionsDemo {

        public static void main(String[] args) {
//      Examples of Creation of Strings
        StringBuffer strBuf1 = new StringBuffer("Bobby");
        StringBuffer strBuf2 = new StringBuffer(100); //With capacity 100
        StringBuffer strBuf3 = new StringBuffer(); //Default Capacity 16
        System.out.println("strBuf1 : " + strBuf1);
        System.out.println("strBuf1 capacity : " + strBuf1.capacity());
        System.out.println("strBuf2 capacity : " + strBuf2.capacity());
        System.out.println("strBuf3 capacity : " + strBuf3.capacity());
        System.out.println("strBuf1 length : " + strBuf1.length());
        System.out.println("strBuf1 charAt 2 : " + strBuf1.charAt(2));
        //  A StringIndexOutOfBoundsException is thrown if the index is not valid.
        strBuf1.setCharAt(1, 't');
        System.out.println("strBuf1 after setCharAt 1 to t is : "+ strBuf1);
        System.out.println("strBuf1 toString() is : " + strBuf1.toString());
        strBuf3.append("beginner-java-tutorial");
        System.out.println("strBuf3 when appended with a String : "+ strBuf3.toString());
        strBuf3.insert(1, 'c');
        System.out.println("strBuf3 when c is inserted at 1 : "+ strBuf3.toString());
        strBuf3.delete(1, 'c');
        System.out.println("strBuf3 when c is deleted at 1 : "+ strBuf3.toString());
        strBuf3.reverse();
```

```
        System.out.println("Reversed strBuf3 : " + strBuf3);
        strBuf2.setLength(5);
        strBuf2.append("jdbc-tutorial");
        System.out.println("strBuf2 : " + strBuf2);
        //    We can clear a StringBuffer using the following line
        strBuf2.setLength(0);
        System.out.println("strBuf2 when cleared using setLength(0): "+ strBuf2);
        }
}
```

Output:

```
strBuf1 : Bobby
strBuf1 capacity : 21
strBuf2 capacity : 100
strBuf3 capacity : 16
strBuf1 length : 5
strBuf1 charAt 2 : b
strBuf1 after setCharAt 1 to t is : Btbby
strBuf1 toString() is : Btbby
strBuf3 when appended with a String : beginner-java-tutorial
strBuf3 when c is inserted at 1 : bceginner-java-tutorial
strBuf3 when c is deleted at 1 : b
Reversed strBuf3 : b
strBuf2 : jdbc-tutorial
strBuf2 when cleared using setLength(0):
```

Unit – 5: JAVAFX EVENT HANDLING, CONTROLS AND COMPONENTS

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.

5.1: Introduction to JavaFX

What is JavaFX?

JavaFX is a set of graphics and media packages that enable developers to design, create, test, debug, and deploy desktop applications and Rich Internet Applications (RIA) that operate consistently across diverse platforms. The applications built in JavaFX can run on multiple platforms including Web, Mobile, and Desktops.

Features of JavaFX:

Feature	Description
Java Library	It consists of many classes and interfaces that are written in Java.
FXML	FXML is the XML based Declarative markup language. The coding can be done in FXML to provide the more enhanced GUI to the user.
Scene Builder	Scene Builder generates FXML mark-up which can be ported to an IDE.
Web view	Web View uses WebKitHTML technology to embed web pages into the Java Applications.
Built in UI controls	Built-in controls are not dependent on operating system. The UI components are used to develop a full featured application.
CSS like styling	JavaFX code can be embedded with the CSS to improve the style and view of the application.
Swing interoperability	The JavaFX applications can be embedded with swing code using the Swing Node class. We can update the existing swing application with the powerful features of JavaFX.
Canvas API	Canvas API provides the methods for drawing directly in an area of a JavaFX scene.
Rich Set of APIs	JavaFX provides a rich set of API's to develop GUI applications.

Integrated Graphics Library	It is provided to deal with 2D and 3D graphics.
Graphics Pipeline	JavaFX graphics are based on Graphics rendered pipeline(prism). It offers smooth graphics which are hardware accelerated.
High Performance Media Engine	The media pipeline supports the playback of web multimedia on a low latency. It is based on a Gstreamer Multimedia framework.
Self-contained application deployment model	Self-Contained application packages have all of the application resources and a private copy of Java and JavaFX Runtime.

5.1.1 : JavaFX Application Structure:

A JavaFX application will have three major components namely

1) Stage
2) Scene and
3) Nodes

as shown in the following diagram.

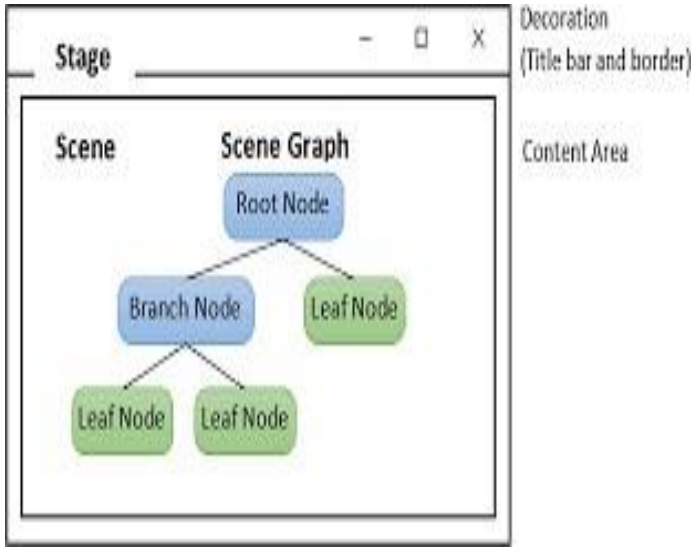


Figure: JavaFX App.lication Structure

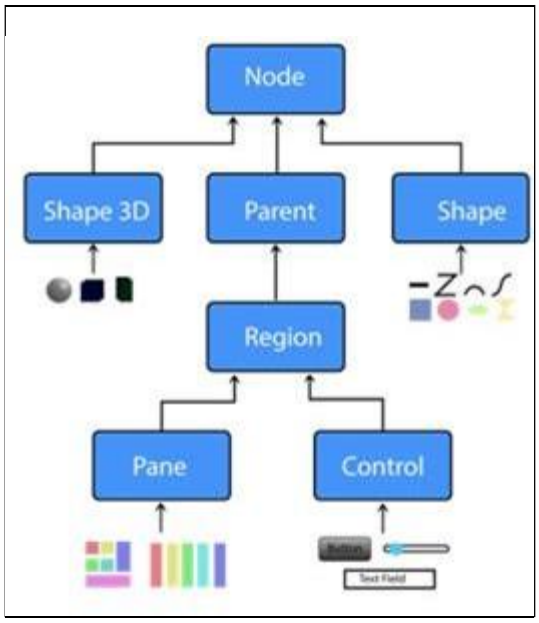


Figure: Scene Graph and Nodes

1) Stage

- ✓ Stage(a window) in a JavaFX application is similar to the Frame in a Swing Application. It acts like a container for all the JavaFX objects.
- ✓ Primary Stage is created internally by the platform. Other stages can further be created by the application.

- ✓ A stage has two parameters determining its position namely **Width** and **Height**. It is divided as Content Area and Decorations (Title Bar and Borders).
- ✓ There are five types of stages available –
  - o Decorated
  - o Undecorated
  - o Transparent
  - o Unified
  - o Utility
- ✓ We have to call the **show()** method to display the contents of a stage.

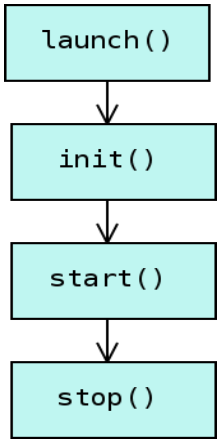
2) <u>Scene</u>

- ✓ A scene represents the physical contents of a JavaFX application. It contains all the contents of a scene graph.
- ✓ The class **Scene** of the package **javafx.scene** represents the scene object. At an instance, the scene object is added to only one stage.

3) <u>Scene Graph and Nodes</u>

- ✓ A **scene graph** is a tree-like data structure (hierarchical) representing the contents of a scene. In contrast, a **node** is a visual/graphical object of a scene graph.
- ✓ A node may include –
  - **Geometrical** (Graphical) objects (2D and 3D) such as – Circle, Rectangle, Polygon, etc.
  - **UI Controls** such as – Button, Checkbox, Choice Box, Text Area, etc.
  - **Containers** (Layout Panes) such as Border Pane, Grid Pane, Flow Pane, etc.
  - **Media elements** such as Audio, Video and Image Objects.
- ✓ A node is of three types –
  - o **Root Node** – The first Scene Graph is known as the Root node.
  - o **Branch Node/Parent Node** – the node with child nodes are known as branch/parent nodes. The parent nodes will be of the following types –
    - ➢ **Group** – A group node is a collective node that contains a list of children nodes. Whenever the group node is rendered, all its child nodes are rendered in order. Any transformation, effect state applied on the group will be applied to all the child nodes.
    - ➢ **Region** – It is the base class of all the JavaFX Node based UI Controls, such as Chart, Pane and Control.
    - ➢ **WebView** – This node manages the web engine and displays its contents.
  - o **Leaf Node** – The node without child nodes is known as the leaf node.

5.1.2: Lifecycle of a JavaFX Application:



The JavaFX Application class has three life cycle methods, which are –

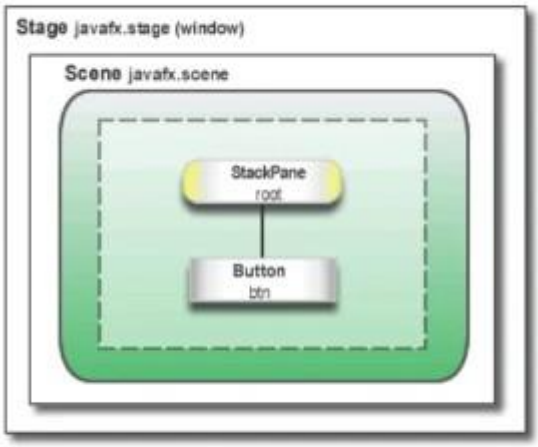
1) **launch()** - to launch JavaFX application.
2) **init()** – An empty method which can be overridden, but you cannot create a stage or scene in this method.
3) **start()** – The entry point method where the JavaFX graphics code is to be written.
4) **stop()** – An empty method which can be overridden, here we can write the logic to stop the application.

General Rules for writing JavaFX Application:

- ✓ A JavaFX Application must extend javafx.application.Application.
- ✓ The main() method should call Application.launch()
- ✓ The start() method is the main entry point for all JavaFX applications
  - Start() is called when a Stage is connected to the Operating System's window
- ✓ The content of the scene is represented as a hierarchical scene graph of nodes:
  - Stage is the top-level JavaFX Container
  - Scene is the container for all content

Minimal example

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World!");

        StackPane root = new StackPane();

        Button btn = new Button();
        btn.setText("Say 'Hello World'");

        root.getChildren().add(btn);

        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }
}
```



5.2 JavaFX Events

5.2.1 : Basic of JavaFX Events:

A GUI based applications are mostly driven by Events. Events are the actions that the user performs and the responses the application generates.

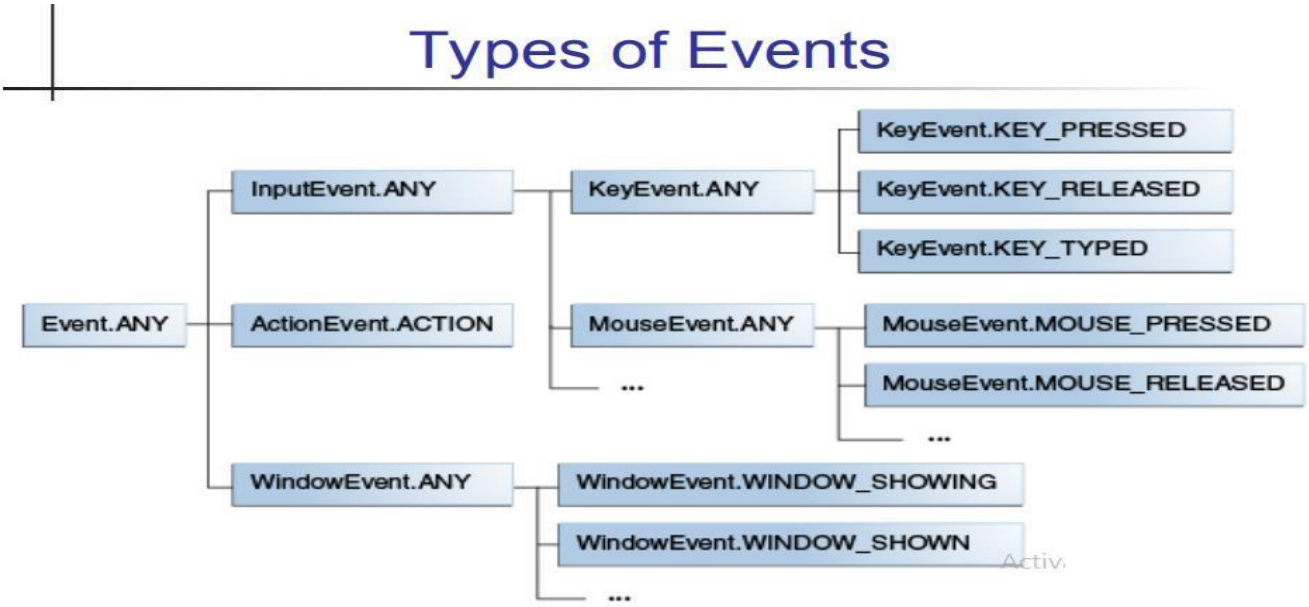
Example: Button clicks by user, key press on the application etc.

An event is a notification about a change. It encapsulates the state changes in the event source. Registered event filters and event handlers within the application receive the event and provide a response.

- ❖ JavaFX provides support to handle events through the base class "Event" which is available in the package javafx.event.

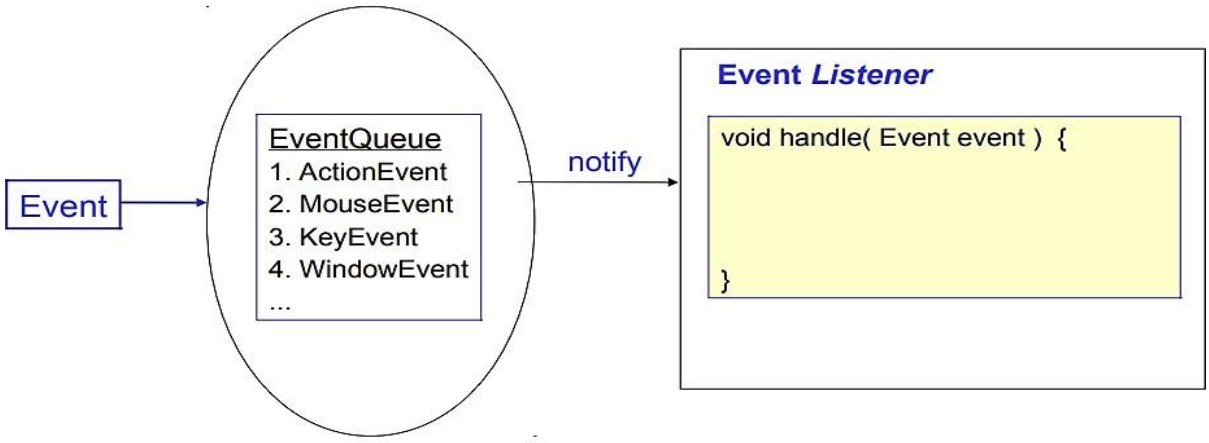
Examples of Events:

- **Action Event** — widely used to indicate things like when a button is pressed.
  Class:- ActionEvent
  Actions:- button pressed.
- **Mouse Event** — occurs when mouse is clicked
  Class:- MouseEvent
  Actions:- mouse clicked, mouse pressed, mouse released, mouse moved, mouse entered target, mouse exited target.
- **Drag Event** — occurs when the mouse is dragged.
  Class:- DragEvent
  Actions:- drag entered, drag dropped, drag entered target, drag exited target, drag over.
- **Key Event** — indicates that a keystroke has occurred.
  Class:- KeyEvent
  Actions:- Key pressed, key released and key typed.
- **Window Event:**
  Class:- WindowEvent
  Actions:- window hiding, window shown, window hidden, window showing.
- **Scroll Event** — indicates scrolling by mouse wheel, track pad, touch screen, etc...
- **TouchEvent** — indicates a touch screen action



5.2.2 : Event Handling:

Event handling is the mechanism that controls the event and decides what should happen, if an event occurs. It has the code which is known as Event Handler that is executed when an event occurs.



Event Handling in JavaFX is done by Event Filters and Event Handlers. They contain the event handling logic to process a generated event.
Every event in JavaFX has three properties:

1. Event source
2. Event target
3. Event type

S.N	Property	Description
1	**Event Source**	It denotes source of the event i.e. the origin which is responsible for generating the event.
2	**Event Target**	It denotes the node on which the event is created. It remains unaffected for the generated event. Event Target is the instance of any of the class that implements the java interface "EventTarget".
3	**Event Type**	It is the type of the event that is being generated. It is basically the instance of EventType class. Example: KeyEvent class contains KEY_PRESSED, KEY_RELEASED, and KEY_TYPED types.

Phases of Event Handling in JavaFX:

Whenever an event is generated, JavaFX undergoes the following phases:

1. Target Selection – Depends on the particular event type.
2. Route Construction – Specified by the event target.
3. Event Capturing – Event travels from the stage to the event target.
4. Event Bubbling – Event travel back from the target to the stage.

1. **Target Selection:**
   The first step to process an event is the selection of the event target. Event target is the node on which the event is created. Event target is selected based in the Event Type.

   - For key events, the target is the node that has key focus.
   - The node where the mouse cursor is located is the target for mouse events.

2. **Route Construction:**
   Usually, an event travels through the event dispatchers in order in the event dispatch chain. An Event Dispatch Chain is created to determine the default route of the event whenever an event is generated. It contains the path from the stage to the node on which the event is generated.

3. **Event Capturing:**
   In this phase, an event is dispatched by the root node and passed down in the Event Dispatch Chain to the target node.
   Event Handlers will not be invoked in this phase.
   If any node in the chain has registered the event filter for the type of event that occurred, then the filter on that node is called. When the filter completes, the

event is moved down to the next node in the Dispatch Chain. If no event filters consumes the event, then the event target receives and processes the generated event.

4. **Event Bubbling:**
   In this phase, a event returns from the target node to the root node along the event dispatch chain.
   Events handlers will be invoked in this phase.
   If any node in the chain has a handler for the generated event, that handler is executes. When the handler completes, the event is bubbled up in the chain. If the handler is not registered for a node, the event is returned to the bubbled up to next node in the route. If no handler in the path consumed the event, the root node consumes the event and completes the processing.

Three methods for Event Handling:

1. **Convenience Methods:**
   - ✓ setOnKeyPressed(eventHandler);
   - ✓ setOnMouseClicked(eventHandler);
2. **Event Handler/Filter Registration Methods:**
   - ✓ addEventHandler(eventType, eventHandler);
   - ✓ addEventFilter(eventType, eventFilter);
3. **Event Dispatcher Property (lambda expression).**

Event Filters:

- ❖ Event Filters provides the way to handle the events generated by the Keyboard Actions, Mouse Actions, Scroll Actions and many more event sources.
- ❖ They process the events during Event Capturing Phase.
- ❖ A node must register the required event filters to handle the generated event on that node. **handle()** method contains the logic to execute when the event is triggered.

- ❖ **Adding Event-Filter to a node:**
  To register the event filter for a node, **addEventFilter()** method is used.

  Syntax:

```
node.addEventFilter (<Event_Type>, new EventHandler<Event-Type>()
{
public void handle(Event-Type)
{
//Actual logic
```

```
});
```

Where,
First argument is the type of event that is generated.
Second argument is the filter to handle the event.

- **Removing Event-Filter:**
  We can remove an event filter on a node using **removeEventFilter()** method.

  Syntax:

```
node.removeEventFilter(<Input-Event>, filter);
```

Event Handlers:

- Event Filters provides the way to handle the events generated by the Keyboard Actions, Mouse Actions, Scroll Actions and many more event sources.
- They are used to handle the events during Event Bubbling Phase.
- A node must register the event handlers to handle the generated event on that node. **handle()** method contains the logic to execute when the event is triggered.

- **Adding Event-Handler to a node:**
  To register the event handler for a node, **addEventHandler()** method is used.

  Syntax:

```
node.addEventHandler (<Event_Type>, new EventHandler<Event-Type>()
{
public void handle(<Event-Type> e)
{
//Handling Code
});
```

Where,
First argument is the type of event that is generated.
Second argument is the filter to handle the event.

- **Removing Event-Filter:**
  We can remove an event handler on a node using **removeEventHandler()** method.

  Syntax:

```
node.removeEventHandler(<EventType>, handler);
```

A node can register for more than one Event Filters and Handlers.
The interface javafx.event.EventHanler must be implemented by all the event filters and event handlers.

5.3: Handling Key Events and Mouse Events

5.3.1 : HANDLING KEY EVENTS

Key Event – It is an input event that indicates the key stroke occurred on a node.

- ✓ It is represented by the class named KeyEvent.
- ✓ This event includes actions like key pressed, key released and key typed.

Types of Key Event in Java

1. KEY_PRESSED – When a key on the keyboard is pressed, this event will be triggered.
2. KEY_RELEASED – When the pressed key on the keyboard is released, this event will be executed.
3. KEY_TYPED – This event will be triggered when a Unicode character is entered

Methods in the KeyEvent class to get the key details

- KeyCode **getCode()** – This method returns the key information or the KeyCode enum constant linked with the pressed or released key.
- String **getText()** – This method returns a String description of the KeyCode linked with the KEY_PRESSED and KEY_RELEASED events.
- String **getCharacter()** – This method returns a string representing a character or a sequence of characters connected with the KEY_TYPED event.

Example:

```
/* Program to handle KeyTyped and KeyPressed Events.
   Whenever a key is pressed in TextFiled1, it will be displayed in TextFiled2.
   Whenever BackSpace key is pressed in TextFiled1, last character in TextFiled2 will
   be erased.
   If you attempt to type a character in TextField2, alert box will be displaying  */

import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.event.*;
import javafx.scene.*;
import javafx.scene.control.*;
```

```
import javafx.scene.layout.*;
import javafx.stage.Stage;
import javafx.scene.input.*;
import javafx.scene.control.Alert.*;

public class NewFXMain  extends Application {

   @Override
   public void start(Stage primaryStage)
{

      TextField tf1=new TextField();
      TextField tf2=new TextField();
      Label l1=new Label("Text Pressed : ");

      EventHandler<KeyEvent> handler1=new EventHandler<KeyEvent>()  {

         String str="",str1="";
         int d;
                  public void handle(KeyEvent event)
                  {
                     if(event.getCode()== KeyCode.BACK_SPACE)
                     {
                     str=str.substring(0,str.length()-1);
                     tf2.setText(str);
                     }
                     else
                     {
                     str+=event.getText();
                     tf2.setText(str);
                     }
                  }
               };

      EventHandler<KeyEvent> handler2=new EventHandler<KeyEvent>(){
         public void handle(KeyEvent event)
         {
            Alert a=new Alert(AlertType.WARNING);
            a.setContentText("Sorry! Dont Type Anything Here!!");
            a.show();
         }
      };

      tf1.setOnKeyPressed(handler1);
      tf2.setOnKeyTyped(handler2);
```

```
        GridPane root = new GridPane();
        root.addRow(1,tf1);
        root.addRow(2,l1);
        root.addRow(3,tf2);
        Scene scene = new Scene(root, 300, 250);
        primaryStage.setTitle("KeyEvent-Demo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

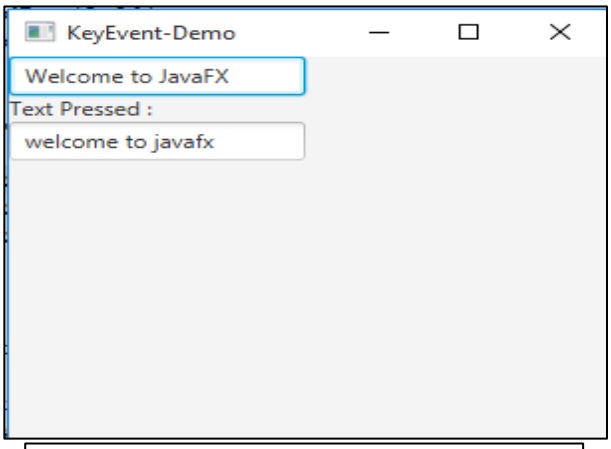


Figure 1: When a key is pressed in TextField 1

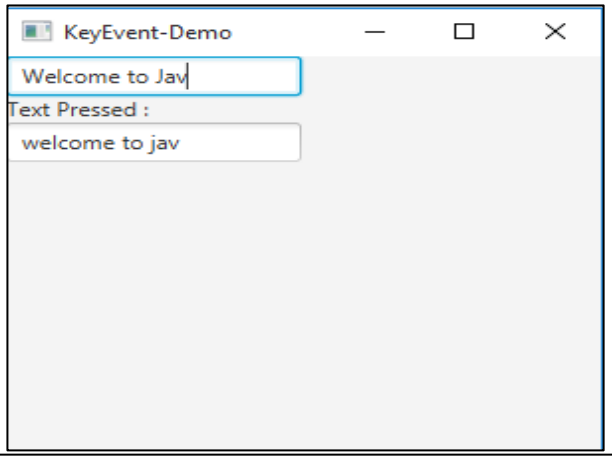
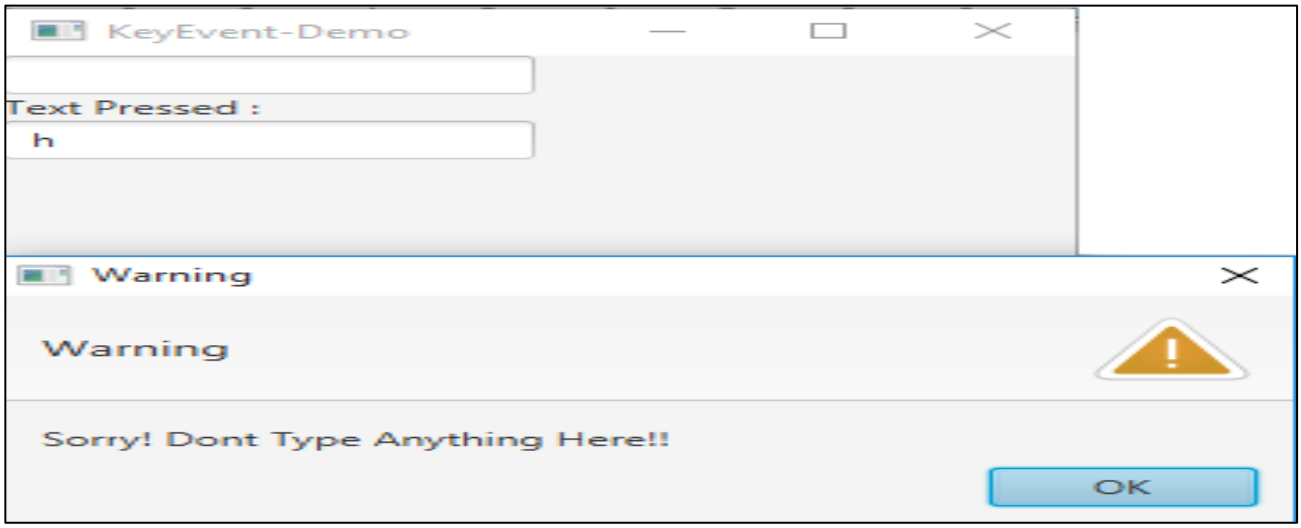


Figure 2: When backspace key is pressed in TextField 1



5.3.2 : HANDLING MOUSE EVENTS

JavaFX Mouse Events are used to handle mouse events. The MouseEvents works when you Clicked, Dragged, or Pressed and etc. An object of the MouseEvent class represents a mouse events.

Types of Mouse Events in JavaFX

- **ANY** – This mouse event type is known as the supertype of all mouse event types. If you want your node to receive all types of events. This event type would be used for your handlers.
- **MOUSE_PRESSED** – When you press a mouse button, this event is triggered. The MouseButton enum defines three constants that represent a mouse button: NONE, PRIMARY, and SECONDARY. The MouseEvent class's getButton() method returns the mouse button that is responsible for the event.
- **MOUSE_RELEASED** – The event is triggered if you pressed and released a mouse button in the same node.
- **MOUSE_CLICKED** – This event will occur when you pressed and released a node.
- **MOUSE_MOVED** – Simply move your mouse without pressing any mouse buttons to generate this type of mouse event.
- **MOUSE_ENTERED** – This event occurs when the mouse or cursor enters the target node.
- **MOUSE_EXITED** – This event occurs when the mouse or cursor leaves or moved outside the target node.
- **MOUSE_DRAGGED** – This event occurs when you move the mouse with a pressed mouse button to a target node.

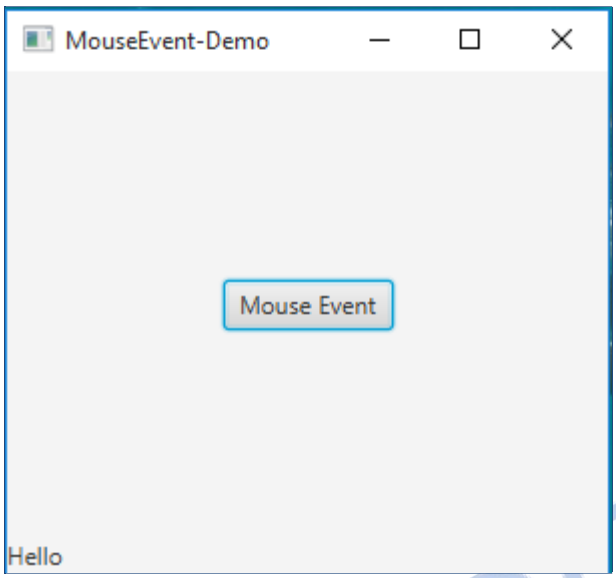
Example:

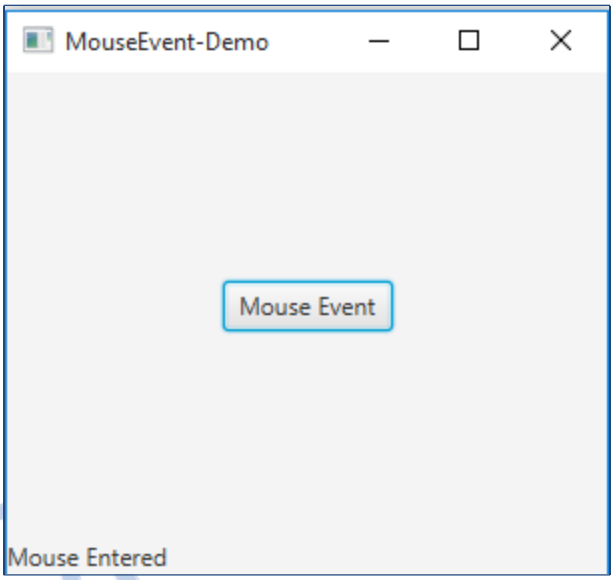
```
import javafx.application.Application;
import javafx.event.Event.*;
import javafx.scene.*;
import javafx.event.EventHandler;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.*;
import javafx.stage.Stage;
import javafx.scene.control.*;
import java.util.*;
```

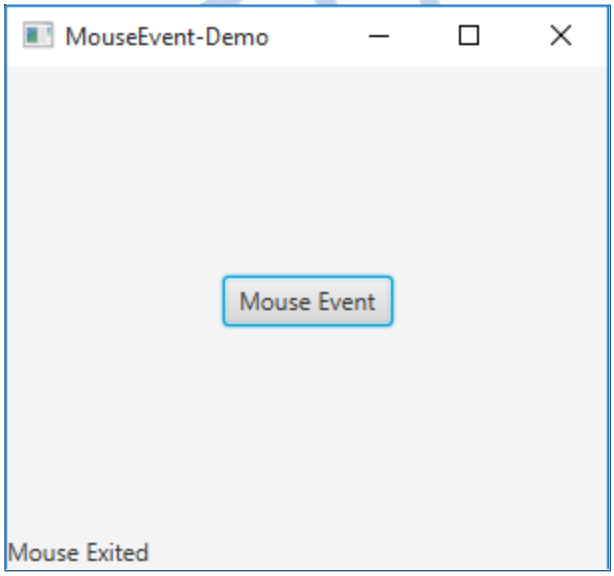
```
public class MouseEvents extends Application {

   @Override
   public void start(Stage primaryStage) {
      Button btn = new Button();
      Label status=new Label();

      btn.setText("Mouse Event");
      status.setText("Hello");
      btn.setOnMousePressed(new EventHandler<MouseEvent>() {
         public void handle(MouseEvent me) {
            status.setText("Mouse pressed");
         }
      });

      btn.setOnMouseEntered(e-> {
         status.setText("Mouse Entered");
      });

      btn.setOnMouseExited(e-> {
         status.setText("Mouse Exited");
      });

      btn.setOnMouseReleased(e-> {
         status.setText("Mouse Released");
      });
      BorderPane bp = new BorderPane();
      bp.setCenter(btn);
      bp.setBottom(status);

      Scene scene = new Scene(bp, 300, 250);
      scene.setOnMouseDragged(e-> {
         status.setText("Mouse Dragged");
      });

      primaryStage.setTitle("MouseEvent-Demo");
      primaryStage.setScene(scene);
      primaryStage.show();
   }
```

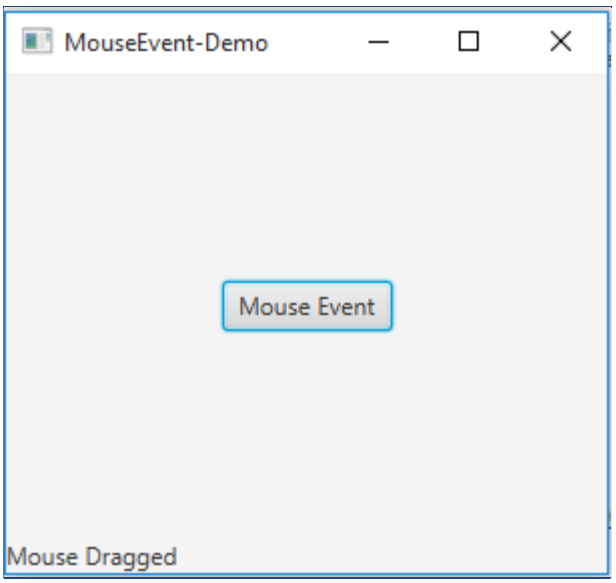
```
    public static void main(String[] args) {
        launch(args);
    }
}
```

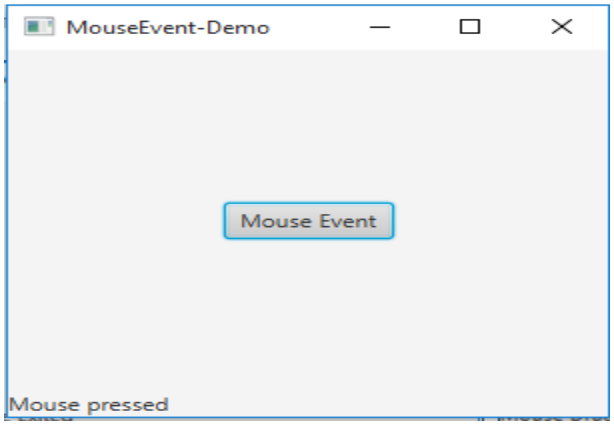
OUTPUT

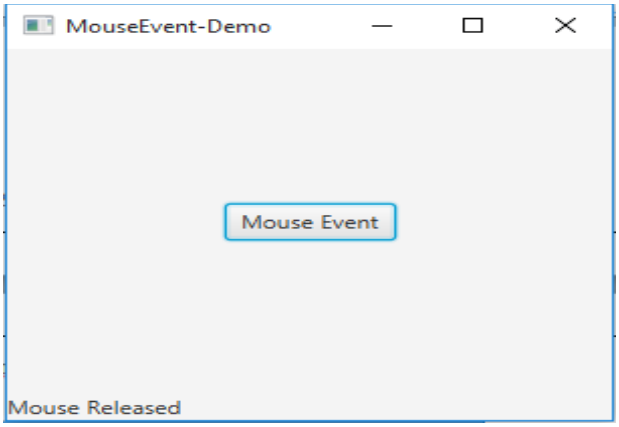








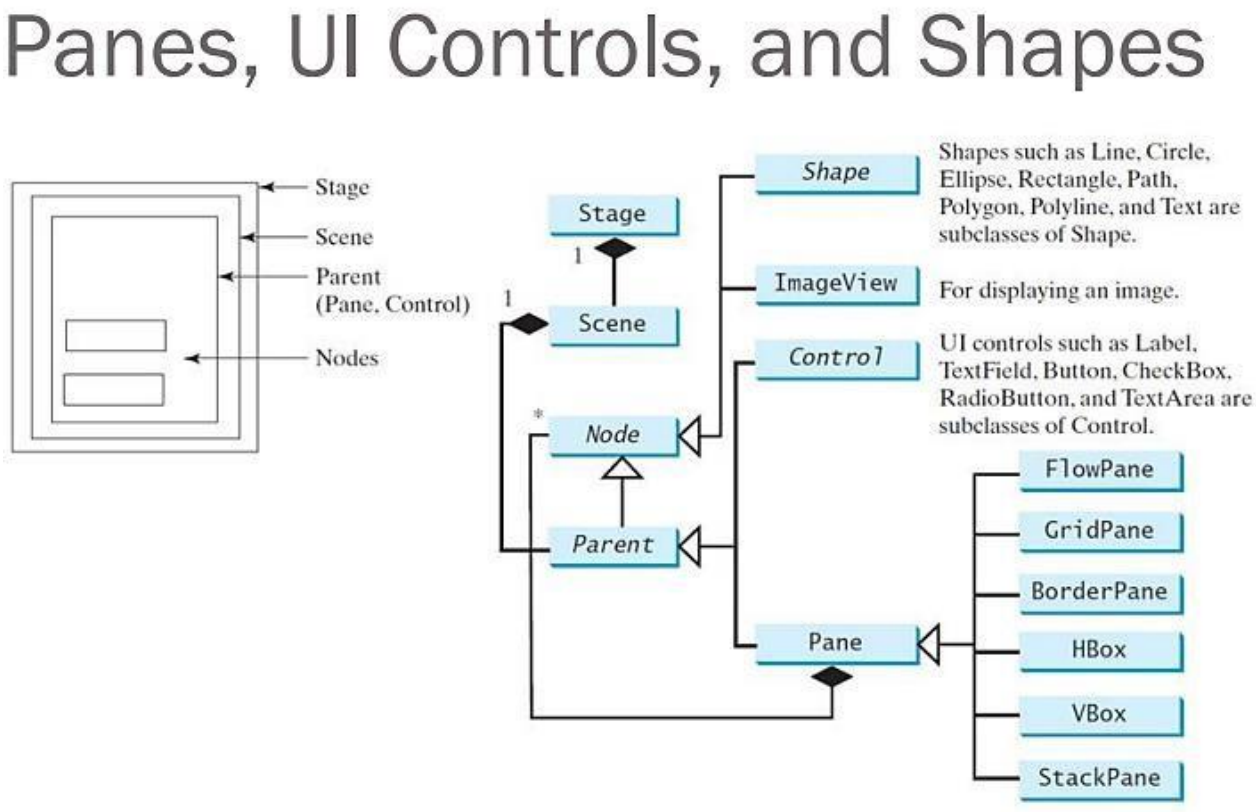




5.4: JavaFX UI Controls

Every user interface considers the following three main aspects –

1. UI elements – These are the core visual elements which the user eventually sees and interacts with.
2. Layouts – They define how UI elements should be organized on the screen.
3. Behavior – These are events which occur when the user interacts with UI elements.



❖ JavaFX provides several classes in the package **javafx.scene.control.**

Figure: JavaFX UI Controls

S. No.	UI Control	Description	Constructors
1.	Label	Component that is used to define a simple text on the screen. It is an not editable text control.	new Label() new Label(String S, Node n) new Label(String s)
2.	TextField	Used to get the input from the user in the form of text. Allows to enter a limited quantity of text.	New TextField()
3.	CheckBox	Used to get the kind of information from the user which contains various choices. User marked the checkbox either on (true) or off(false).	new CheckBox() new CheckBox(String s)
4.	RadioButton	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.	new RadioButton() new RadioButton(String s)
5.	Button	Component that controls the function of the application.	new Button() new Button(String s)
6.	ComboBox	Shows a list of items out of which user can select at most one item	new ComboBox new ComboBox(ObservableList i)
7.	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.	new ChoiceBox new ChoiceBox(ObservableList i)
8.	ListView	Enables users to choose one or more options from a predefined list of choices.	new ListView();
9.	ScrollPane	It provides a scrollable view of UI Elements. It is a container that has two scrollbars around the component it contains if the component is larger than the visible area of the ScrollPane. The scrollbars enable the user to scroll around the component shown inside the ScrollPane	new ScrollPane();
10.	ToggleButton	Special control having the ability to be selected. Basically, ToggleButton is rendered similarly to a Button but these two are the different types of Controls. A Button is a “command” button that invokes a function when clicked. But a ToggleButton is a control with a Boolean indicating whether it is selected.	new ToggleButton newToggleButton(String txt) new ToggleButton(String txt, Node graphic)

Selected User Actions and Handlers

User Action	*Source Object*	*Event Type Fired*	*Event Registration Method*
Click a button	`Button`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Press Enter in a text field	`TextField`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`RadioButton`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`CheckBox`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Select a new item	`ComboBox`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Mouse pressed	`Node, Scene`	`MouseEvent`	`setOnMousePressed(EventHandler<MouseEvent>)`
Mouse released			`setOnMouseReleased(EventHandler<MouseEvent>)`
Mouse clicked			`setOnMouseClicked(EventHandler<MouseEvent>)`
Mouse entered			`setOnMouseEntered(EventHandler<MouseEvent>)`
Mouse exited			`setOnMouseExited(EventHandler<MouseEvent>)`
Mouse moved			`setOnMouseMoved(EventHandler<MouseEvent>)`
Mouse dragged			`setOnMouseDragged(EventHandler<MouseEvent>)`
Key pressed	`Node, Scene`	`KeyEvent`	`setOnKeyPressed(EventHandler<KeyEvent>)`
Key released			`setOnKeyReleased(EventHandler<KeyEvent>)`
Key typed			`setOnKeyTyped(EventHandler<KeyEvent>)`

Example : JavaFX program for Simple Registration form using UI Controls:

```
import javafx.application.Application;
import javafx.collections.*;

import javafx.geometry.Insets;
import javafx.geometry.Pos;

import javafx.scene.image.*;

import javafx.scene.Scene;
import javafx.scene.control.*;

import javafx.scene.layout.*;
import javafx.scene.text.Text;

import javafx.stage.Stage;

public class JavaFXControlDemo extends Application {
  @Override
  public void start(Stage stage)
{
```

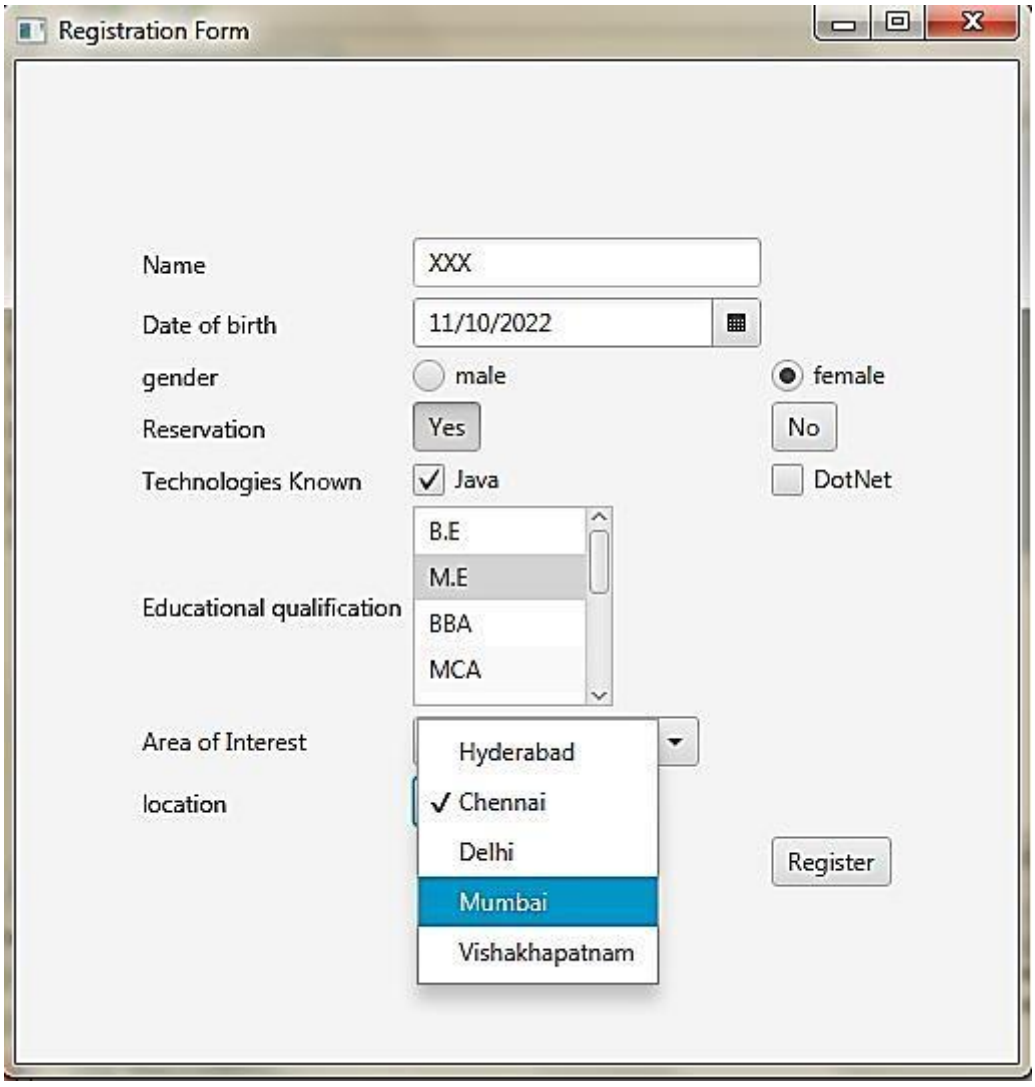
```
//Label for name
Text nameLabel = new Text("Name");

//Text field for name
TextField nameText = new TextField();

//Label for date of birth
Text dobLabel = new Text("Date of birth");

//date picker to choose date
DatePicker datePicker = new DatePicker();

//Label for gender
Text genderLabel = new Text("gender");

//Toggle group of radio buttons
ToggleGroup groupGender = new ToggleGroup();
RadioButton maleRadio = new RadioButton("male");
maleRadio.setToggleGroup(groupGender);
RadioButton femaleRadio = new RadioButton("female");
femaleRadio.setToggleGroup(groupGender);

//Label for reservation
Text reservationLabel = new Text("Reservation");

//Toggle button for reservation
ToggleButton yes = new ToggleButton("Yes");
ToggleButton no = new ToggleButton("No");
ToggleGroup groupReservation = new ToggleGroup();
yes.setToggleGroup(groupReservation);
no.setToggleGroup(groupReservation);

//Label for technologies known
Text technologiesLabel = new Text("Technologies Known");

//check box for education
CheckBox javaCheckBox = new CheckBox("Java");
javaCheckBox.setIndeterminate(false);

//check box for education
CheckBox dotnetCheckBox = new CheckBox("DotNet");
javaCheckBox.setIndeterminate(false);

//Label for education
```

```
        Text educationLabel = new Text("Educational qualification");

        //list View for educational qualification
        ObservableList<String> names = FXCollections.observableArrayList(
          "B.E","M.E","BBA","MCA", "MBA", "Vocational", "M.TECH", "Mphil",
"Phd");
        ListView<String> educationListView = new ListView<String>(names);
        educationListView.setMaxSize(100, 100);

educationListView.getSelectionModel().setSelectionMode(SelectionMode.MU
LTIPLE);

        Label interest=new Label("Area of Interest");
        ComboBox AoI=new ComboBox();
        AoI.getItems().addAll("Android App. Dev.", "IoS App. Dev.", "FUll Stack
Dev.", "Azure FrmWork", "AWS", "Web Dev.", "Ui/Ux Design");
        AoI.setVisibleRowCount(3);

        //Label for location
        Text locationLabel = new Text("location");

        //Choice box for location
        ChoiceBox locationchoiceBox = new ChoiceBox();
        locationchoiceBox.getItems().addAll
          ("Hyderabad", "Chennai", "Delhi", "Mumbai", "Vishakhapatnam");

        //Label for register
        Button buttonRegister = new Button("Register");

        //Creating a Grid Pane
        GridPane gridPane = new GridPane();

        //Setting size for the pane
        gridPane.setMinSize(500, 500);

        //Setting the padding
        gridPane.setPadding(new Insets(10, 10, 10, 10));

        //Setting the vertical and horizontal gaps between the columns
        gridPane.setVgap(5);
        gridPane.setHgap(5);


        //Setting the Grid alignment
        gridPane.setAlignment(Pos.CENTER);
```

```
        //Arranging all the nodes in the grid
        gridPane.add(nameLabel, 0, 0);
        gridPane.add(nameText, 1, 0);

        gridPane.add(dobLabel, 0, 1);
        gridPane.add(datePicker, 1, 1);

        gridPane.add(genderLabel, 0, 2);
        gridPane.add(maleRadio, 1, 2);
        gridPane.add(femaleRadio, 2, 2);
        gridPane.add(reservationLabel, 0, 3);
        gridPane.add(yes, 1, 3);
        gridPane.add(no, 2, 3);

        gridPane.add(technologiesLabel, 0, 4);
        gridPane.add(javaCheckBox, 1, 4);
        gridPane.add(dotnetCheckBox, 2, 4);

        gridPane.add(educationLabel, 0, 5);
        gridPane.add(educationListView, 1, 5);

        gridPane.add(interest,0,6);
        gridPane.add(AoI,1,6);

        gridPane.add(locationLabel, 0, 7);
        gridPane.add(locationchoiceBox, 1, 7);

        gridPane.add(buttonRegister, 2, 8);


        Scene scene = new Scene(gridPane);

        //Setting title to the Stage
        stage.setTitle("Registration Form");

        //Adding scene to the stage
        stage.setScene(scene);

        //Displaying the contents of the stage
        stage.show();
}
```

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

OUTPUT:



5.5: Layouts – FlowPane – HBox and VBox – BorderPane – StackPane – GridPane.

In JavaFX, Layout defines the way in which the components are to be seen on the stage. It basically organizes the scene-graph nodes.

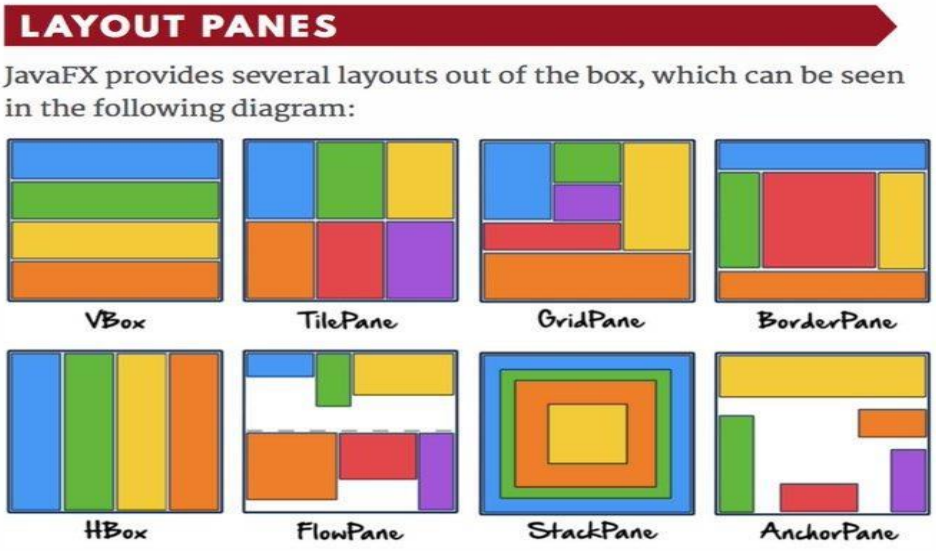
Layout Panes: Layout panes are containers which are used for flexible and dynamic arrangements of UI controls within a scene graph of a JavaFX application.

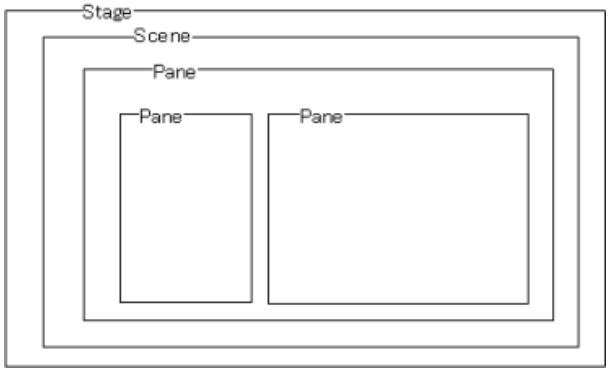
Package used: javaFX.scene.layout package

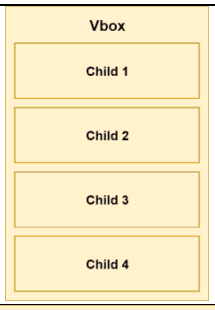
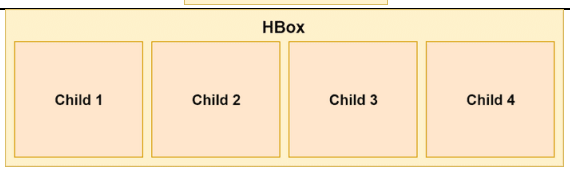
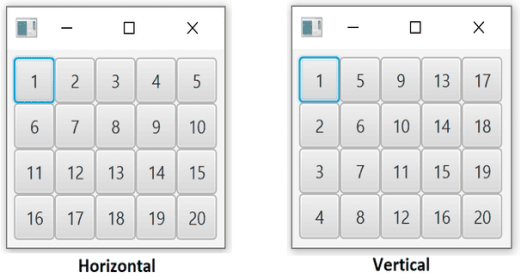
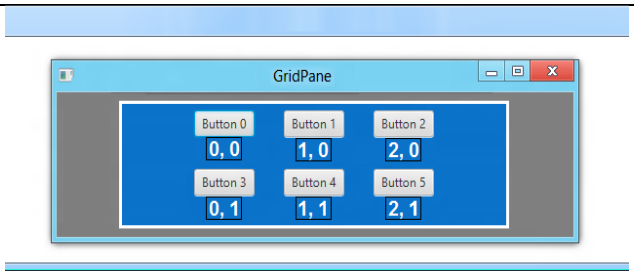
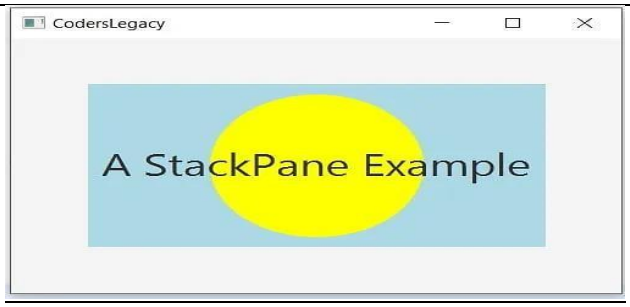
JavaFX provides various built-in Layouts that are

1. Pane
2. VBox
3. HBox
4. BorderPane
5. FlowPane
6. GridPane
7. StackPane.

JavaFX provides many types of panes for organizing nodes in a container:



Class	Description	Representation
Pane	Base class for layout panes. It containes the getChildren() method for returning a list of nodes in the pane.	

VBox	Places the nodes in a single column	
HBox	Places the nodes in a single row	
BorderPane	Places the nodes in the top, right, bottom, left and center regions	
FlowPane	Places the nodes row-by-row horizontally or column-by-column vertically	
GridPane	Places the nodes in the cells in a two-dimensional grid(like matrix)	
StackPane	Places the nodes on top of each other in the center of the pane	

Methods and Properties of different layouts:

<table>
<tr><th>Layout</th><th>Constructors</th><th>Methods/Properties</th></tr>
<tr><td>VBox</td><td>1. VBox() : creates layout with 0 spacing
2. Vbox(Double spacing) : creates layout with a spacing value of double type
3. Vbox(Double spacing, Node? children) : creates a layout with the specified spacing among the specified child nodes
4. Vbox(Node? children) : creates a layout with the specified nodes having 0 spacing among them</td><td>
<table>
<tr><th>Property</th><th>Description</th><th>Setter Methods</th></tr>
<tr><td>Alignment</td><td>This property is for the alignment of the nodes.</td><td>setAlignement(Double)</td></tr>
<tr><td>FillWidth</td><td>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.</td><td>setFillWidth(boolean)</td></tr>
<tr><td>Spacing</td><td>This property is to set some spacing among the nodes of VBox.</td><td>setSpacing(Double)</td></tr>
</table>
</td></tr>
<tr><td>HBox</td><td>1. new HBox() : create HBox layout with 0 spacing
2. new Hbox(Double spacing) : create HBox layout with a spacing value</td><td>
<table>
<tr><th>Property</th><th>Description</th><th>Setter Methods</th></tr>
<tr><td>Alignment</td><td>This represents the alignment of the nodes.</td><td>setAlignment(Double)</td></tr>
<tr><td>fillHeight</td><td>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.</td><td>setFillHeight(Double)</td></tr>
<tr><td>spacing</td><td>This represents the space between the nodes in the HBox. It is of double type.</td><td>setSpacing(Double)</td></tr>
</table>
</td></tr>
<tr><td>BorderPane</td><td>1. BorderPane() :- create the empty layout
2. BorderPane(Node Center):- create the layout with the center node
3. BorderPane(Node Center, Node top, Node right, Node bottom, Node left) :- create the layout with all the nodes</td><td>
<table>
<tr><th>Type</th><th>Property</th><th>Setter Methods</th><th>Description</th></tr>
<tr><td>Node</td><td>Bottom</td><td>setBottom()</td><td>Add the node to the bottom of the screen</td></tr>
<tr><td>Node</td><td>Centre</td><td>setCentre()</td><td>Add the node to the centre of the screen</td></tr>
<tr><td>Node</td><td>Left</td><td>setLeft()</td><td>Add the node to the left of the screen</td></tr>
<tr><td>Node</td><td>Right</td><td>setRight()</td><td>Add the node to the right of the screen</td></tr>
<tr><td>Node</td><td>Top</td><td>setTop()</td><td>Add the node to the top of the screen</td></tr>
</table>
</td></tr>
</table>

<table>
<tr>
<td>FlowPane</td>
<td>1. FlowPane()
2. FlowPane(Double Hgap, Double Vgap)
3. FlowPane(Double Hgap, Double Vgap, Node? children)
4. FlowPane(Node... Children)
5. FlowPane(Orientation orientation)
6. FlowPane(Orientation orientation, double Hgap, Double Vgap)</td>
<td>
<table>
<tr><th>Property</th><th>Description</th><th>Setter Methods</th></tr>
<tr><td>alignment</td><td>The overall alignment of the flowpane's content.</td><td>setAlignment(Pos value)</td></tr>
<tr><td>columnHalignment</td><td>The horizontal alignment of nodes within the columns.</td><td>setColumnHalignment(HPos Value)</td></tr>
<tr><td>hgap</td><td>Horizontal gap between the columns.</td><td>setHgap(Double value)</td></tr>
<tr><td>orientation</td><td>Orientation of the flowpane</td><td>setOrientation(Orientation value)</td></tr>
<tr><td>prefWrapLength</td><td>The preferred height or width where content should wrap in the horizontal or vertical flowpane.</td><td>setPrefWrapLength(double value)</td></tr>
<tr><td>rowValignment</td><td>The vertical alignment of the nodes within the rows.</td><td>setRowValignment(VPos value)</td></tr>
<tr><td>vgap</td><td>The vertical gap among the rows</td><td>setVgap(Double value)</td></tr>
</table>
</td>
</tr>
<tr>
<td>GridPane</td>
<td>Public GridPane(): creates a gridpane with 0 hgap/vgap.</td>
<td>
<table>
<tr><th>Property</th><th>Description</th><th>Setter Methods</th></tr>
<tr><td>alignment</td><td>Represents the alignment of the grid within the GridPane.</td><td>setAlignment(Pos value)</td></tr>
<tr><td>gridLinesVisible</td><td>This property is intended for debugging. Lines can be displayed to show the gidpane's rows and columns by setting this property to true.</td><td>setGridLinesVisible(Boolean value)</td></tr>
<tr><td>hgap</td><td>Horizontal gaps among the columns</td><td>setHgap(Double value)</td></tr>
<tr><td>vgap</td><td>Vertical gaps among the rows</td><td>setVgap(Double value)</td></tr>
</table>
</td>
</tr>
<tr>
<td>StackPane</td>
<td>1. StackPane()
2. StackPane(Node? Children)</td>
<td>
<table>
<tr><th>Property</th><th>Description</th><th>Setter Methods</th></tr>
<tr><td>alignment</td><td>It represents the default alignment of children within the StackPane's width and height</td><td>setAlignment(Node child, Pos value)</td></tr>
</table>
</td>
</tr>
</table>

Example: Program for Layouts, Menus and MenuBars, Event Handling

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class JavaFXApplication1 extends Application {
  @Override//from ww w . ja v a2s . co m
  public void start(Stage primaryStage) {
    // Create the top section of the UI
    Text tNumber1 = new Text("Number 1:");
    Text tNumber2 = new Text("Number 2:");
    Text tResult = new Text("Result:");
    TextField tfNumber1 = new TextField();
    TextField tfNumber2 = new TextField();
    TextField tfResult = new TextField();
    tfResult.setEditable(false);

    Menu me=new Menu("Edit");
    // create menuitems
     MenuItem m1 = new MenuItem("Set Default Value");
     MenuItem m2 = new MenuItem("Clear All values");

     // add menu items to menu
     me.getItems().add(m1);
     me.getItems().add(m2);

    Menu mc=new Menu("Bg_Color");
    MenuItem c1 = new MenuItem("Red");
    MenuItem c2 = new MenuItem("Green");

    mc.getItems().addAll(c1,c2);
    MenuBar mb = new MenuBar();
```

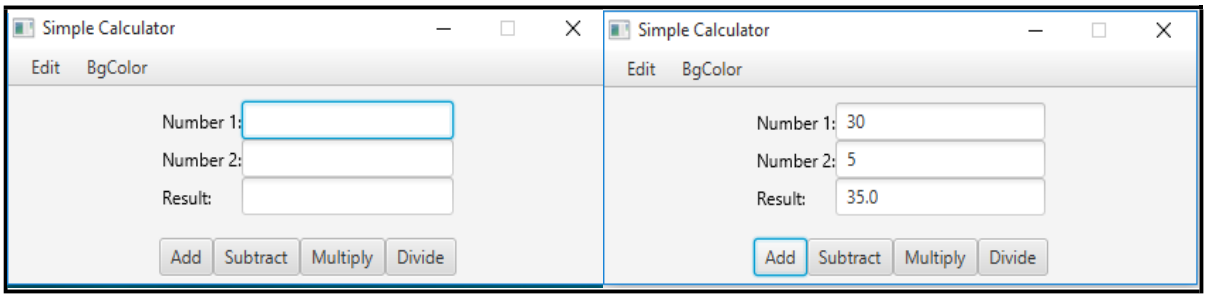


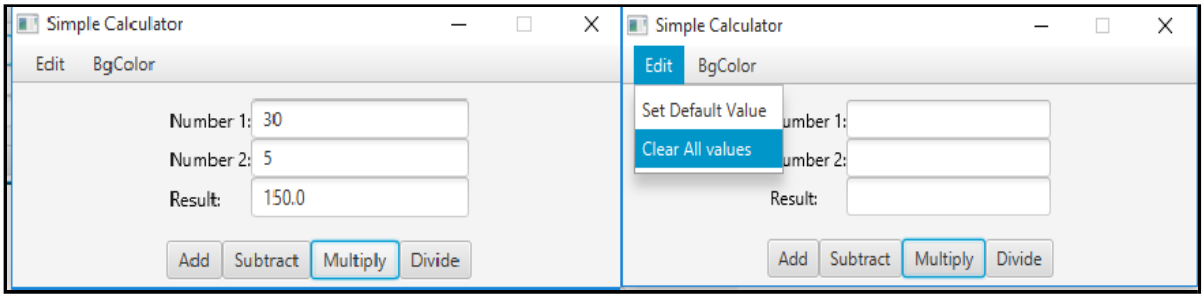
```
    // add menu to menubar
```

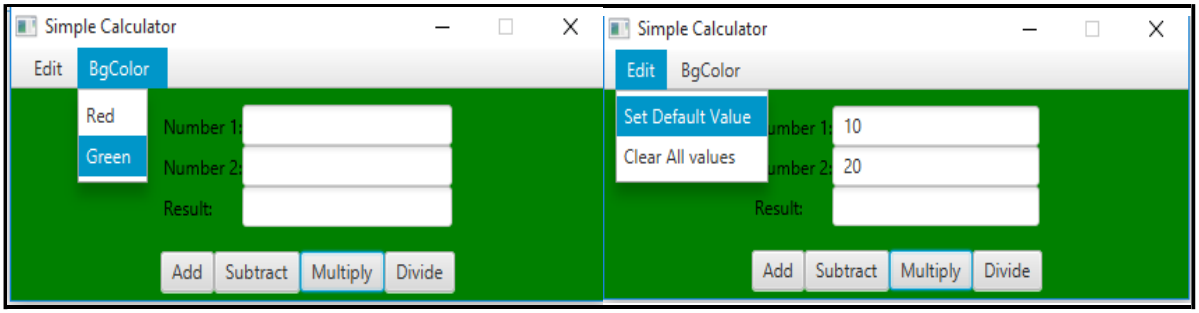
```
    mb.getMenus().add(me);
    mb.getMenus().add(mc);

    VBox vb=new VBox(mb);

    m1.setOnAction(e -> {
       tfNumber1.setText("10");
       tfNumber2.setText("20");
    });

    m2.setOnAction(e ->{
       tfNumber1.setText("");
       tfNumber2.setText("");
       tfResult.setText("");
    });

   // Create the bottom section of the UI
    Button btAdd = new Button("Add");
    Button btSubtract = new Button("Subtract");
    Button btMultiply = new Button("Multiply");
    Button btDivide = new Button("Divide");

    // Add top and bottom UI to HBox containers

    GridPane calcTop = new GridPane();
    calcTop.setAlignment(Pos.CENTER);
    calcTop.setPadding(new Insets(5));
    calcTop.add(tNumber1, 0, 0);
    calcTop.add(tfNumber1, 1, 0);
    calcTop.add(tNumber2, 0, 1);
    calcTop.add(tfNumber2, 1, 1);
    calcTop.add(tResult, 0, 2);
    calcTop.add(tfResult, 1, 2);


    FlowPane calcBottom = new FlowPane();
    calcBottom.setAlignment(Pos.CENTER);
    calcBottom.setPadding(new Insets(5));
```

```
    calcBottom.getChildren().addAll(btAdd, btSubtract, btMultiply, btDivide);

    // Add HBox containers to a BorderPane
    BorderPane pane = new BorderPane();
    pane.setTop(vb);
    pane.setCenter(calcTop);
    pane.setBottom(calcBottom);

    c1.setOnAction(e -> {
       pane.setBackground(new Background(new BackgroundFill(Color.RED,null,null)));
    });
    c2.setOnAction(e -> {
       pane.setBackground(new Background(new
BackgroundFill(Color.GREEN,null,null)));
    });

    // Register event handlers for buttons
    btAdd.setOnAction(e -> {
      double a = getDoubleFromTextField(tfNumber1);
      double b = getDoubleFromTextField(tfNumber2);
      tfResult.setText(String.valueOf(a + b));
    });

    btSubtract.setOnAction(e -> {
      double a = getDoubleFromTextField(tfNumber1);
      double b = getDoubleFromTextField(tfNumber2);
      tfResult.setText(String.valueOf(a - b));
    });

    btMultiply.setOnAction(e -> {
      double a = getDoubleFromTextField(tfNumber1);
      double b = getDoubleFromTextField(tfNumber2);
      tfResult.setText(String.valueOf(a * b));
    });

    btDivide.setOnAction(e -> {
      double a = getDoubleFromTextField(tfNumber1);
      double b = getDoubleFromTextField(tfNumber2);
      tfResult.setText(b == 0 ? "NaN" : String.valueOf(a / b));
```

```
    });

    Scene scene = new Scene(pane);
    primaryStage.setTitle("Simple Calculator");
    primaryStage.setScene(scene);
    primaryStage.setResizable(false);
    primaryStage.show();
  }

  private static double getDoubleFromTextField(TextField t) {
    return Double.parseDouble(t.getText());
  }

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







5.6: Menus – Basics – Menu – Menu bars – MenuItem.

5.6.1. JavaFX Menus, MenuItem and MenuBar:

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.

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:

1. **MenuBar():** creates a new empty menubar.
2. **MenuBar(Menu... m):** creates a new menubar with the given set of menu.

Constructor of the Menu class are:

1. **Menu():** creates an empty menu
2. **Menu(String s):** creates a menu with a string as its label
3. **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.
4. **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.
setOnShown(EventHandler v	Sets the value of the property onShown.

JavaFX Menu

- ✓ In the JavaFX application, in order to create a menu, menu items, and menu bar, Menu, MenuItem, and MenuBar class is used. The menu allows us to choose options among available choices in the application.
- ✓ All methods needed for this purpose are present in the javafx.scene.control.Menu class.

Example: **Java program to create a menu bar and add menu to it and also add menuitems to the menu**

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuBar;
import javafx.scene.control.MenuItem;
import javafx.scene.layout.VBox;
public class MenuUI extends Application {
@Override
    public void start(Stage primaryStage) throws Exception
    {
            Menu newmenu = new Menu("File");
            Menu newmenu2 = new Menu("Edit");

            MenuItem m1 = new MenuItem("Open");
            MenuItem m2 = new MenuItem("Save");
            MenuItem m3 = new MenuItem("Exit");
            MenuItem m4 = new MenuItem("Cut");
            MenuItem m5 = new MenuItem("Copy");
```

```
            MenuItem m6 = new MenuItem("Paste");
            newmenu.getItems().add(m1);
            newmenu.getItems().add(m2);
            newmenu.getItems().add(m3);
            newmenu2.getItems().add(m4);
            newmenu2.getItems().add(m5);
            newmenu2.getItems().add(m6);
            MenuBar newmb = new MenuBar();
            newmb.getMenus().add(newmenu);
            newmb.getMenus().add(newmenu2);
            VBox box = new VBox(newmb);
            Scene scene = new Scene(box,400,200);
            primaryStage.setScene(scene);
            primaryStage.setTitle("JavaFX Menu Example");
            primaryStage.show();
    }
    public static void main(String[] args)
    {
            Application.launch(args);
    }
}
```

Output: