B.Sc.(ECS)-II

# Core JAVA

## Unit-1

## Introduction:

We know that, there are two paradigms (Models) of Programming languages which are as follows:

1) Procedure oriented Paradigm (POP).
2) Object oriented Paradigm (OOP).

## Procedure Oriented Paradigm (POP):

- The POP is also called as functional programming because the strength of POP languages Isfunction. Also, all the statements in the functions are executed one after another from top to bottom in sequence manner therefore it is commonly known as "Procedure oriented"
- High level languages such as COBOL, FORTRON, and C etc. are commonly known as POP languages.

## Advantages or Characteristics of POP languages:

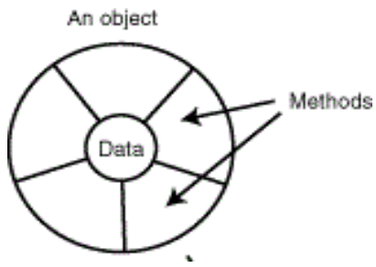
1) In case of POP the focus is on Function rather than data.
2) It fallows top-down approach.
3) Program consists of different functions.
4) Most functions share global data.
5) Communication is done between functions i.e. one function can send data to another function.
6) Data is not hidden i.e. data is easily accessed& modified by other functions therefore it is not secured.

## Object Oriented Paradigm (OOP):

The OOP considers "data" as important thing & does not allow it to flow freely around the program i.e. it provides security for data.

In OOP "data" is closely tied with functions or methods that protects it from unauthorized modification by another function.

- OOP allows us to decompose a problem into a number of entities called "object" and then build data and functions around these objects. In short, an object is combination or composition of "data" & "functions" or "methods" which is shown in following fig.




In above figure, the data of an abject is only accessed by methods associated with an object.

- High level languages such as JAVA, C#, C++ etc. are commonly known as OOP languages.

## Advantages or Characteristics of OOP languages:

1) In case of OOP the focus is on "data" rather than function.
2) It fallows bottom-up approach.
3) Program consists of different objects.
4) Main problem is divided into number of entities called "Objects"
5) Communication is done between objects through methods.
6) Data is hidden i.e. data& functions are tied together with an object that"s why it is not easily accessed or modified by other functions therefore it is secured.
7) New "data" and "function" can easily be added whenever required.

## Basic Concepts of Object Oriented Programming:

There are several OOP concepts which are as follows:

### 1) Class:

- Class is user defined data type which is collection of data and functions or methods.
- Here, "data" is nothing but properties or characteristics of class whereas "methods" are the functions that operates on data of class
- In JAVA language, all functions are defined inside class definition therefore they are called as "Methods". Even main( ) also defined inside class definition.
- Single JAVA program contains many classes but out of these classes, any single class have main( ) method from which execution of program starts.
- Class can be declared or defined by using keyword "class" followed by "class_name" which

is an identifier.

Syntax to declare or define class:

```
class       class_name
{
    data_type        data1;          <-- Data
    data_type        data2;
          |                |
          |                |
data_type dataN;

    return_typemethod1( );
    return_typemethod2( );           <-- Methods
          |                |
          |                |

    return_typemethodN( );
  }
```

- Multiple objects can be created for the single class whenever required and each object hold individual record i.e. individual data & methods.
- A class provides the blueprints for objects.
- When you define a class, you define a blueprint for a data type.

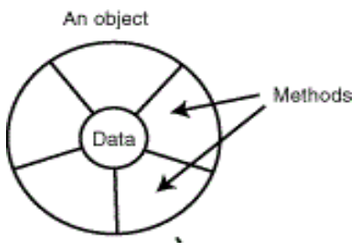
e.g. Following example shows the definition of employee class:

```
class            emp
{
        int      emp_no;
        char     name[20], add[20];      <-- Data (Attributes)
        floatsalary;

        void     get( );
        void     show( );                <-- Methods (Operations)
}
```

## 2) Object:

- An Object is an instance or variable of class.
- An object can be thought of as an entity which can represent person, place, animal etc.
- Object is such run time entity which holds entire data and methods of class.
- When program is executed, the objects interact with each other by sending messages to one another.
- There are three properties related with objects:
    - Identity: This property of an object that distinguishes it from other objects
    - State: This property describes the data stored in the object
    - Behavior: This property describes the methods in the object's interface by which the object can be used
- Multiple objects can be created for the single class whenever required and each object hold individual record.
- Data and methods are tied with an object therefore data cannot access or modified by another function.

Following Fig. shows an object:

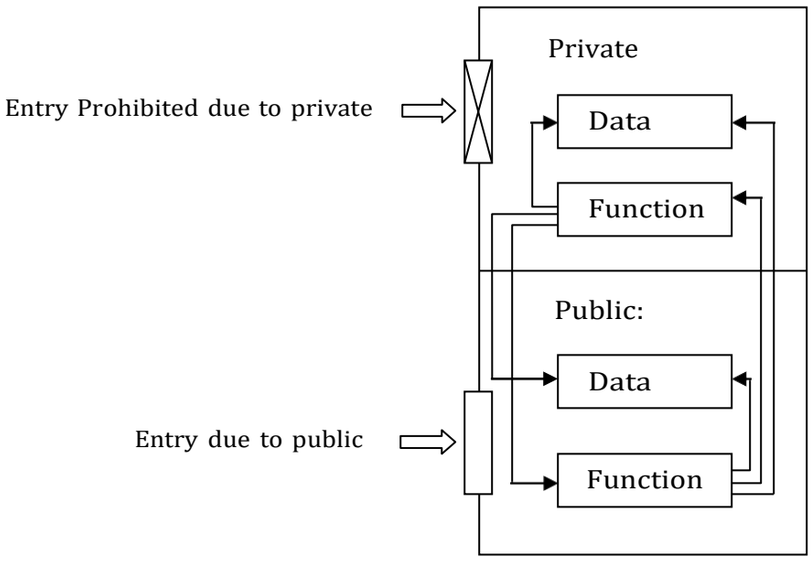



## 3) Data Encapsulation:

- The binding of data and methods into single unit is called “Data encapsulation”. And that keeps both data and methods safe from outside the world and misuse.
- Data encapsulation lead to the important OOP concept of data hiding and is accomplished by class and object. Because, Object binds "data" & "methods" together therefore this data is only accessed by methods of corresponding object and not accessed by outside the world (other methods).

- A benefit of encapsulation is that it can reduce system complexity, and thus increases robustness, by allowing the developer to limit the interdependencies between software components.
- This feature supports the data security in OOP.

Following figure shows Data encapsulation concept:




## 4) Data Abstraction:

- Data abstraction refers to, providing only essential information to the outside word and hiding their background details.
- This is closely related to encapsulation because abstraction is implemented by encapsulation.

InJAVA, only public "data" or "methods" are accessible to the outside world to manipulate object"s data, without actually knowing how class has been implemented internally.

- If we have to give accessibility to some data of the class then it has to be madeaspublicand those we have to make hide from outside world then it has to made private . Thus, Classes makes data abstraction with the help of access specifier or visibility mode.
- E.g.

Let's take one real life example of a TV. Which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players.

But you do not know its internal detailssuch as,

- you do not know how it receives signals over the air or through a cable
- How it translates them, and finally displays them on the screen.

Thus we can say, a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals

## 5) Polymorphism:

- The word polymorphism is made up of two Greek words “poly” and “morphism”.
- “Poly” means many and “morphism” means forms, so polymorphism means many-forms.
- In OOP, we implement many functions with same name but these functions are differ from each other according to their signature. (Signature refers to type of argument and number of argument accepts by the functions)
- In OOP, polymorphism means “one name” named for multiple functions that have different behaviors.
- E.g. Consider following prototype of functions:

```
int    add( );
void   add(int);
int    add(int, int);
```

In above example, many functions having same name but all these differ from each other by signatures therefore here polymorphism occurs.

Basically polymorphism having two types:

1) Compile Time polymorphism (Static linkage/Static Binding/Early Binding):

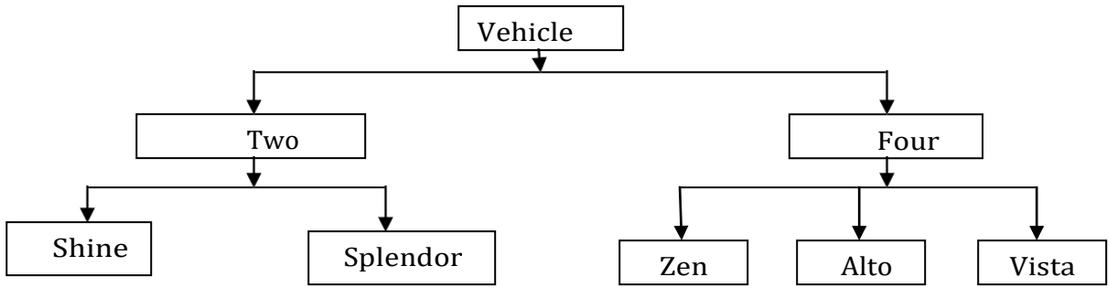
Definition: Choosing the functions at compile time of program is called as “Compile time polymorphism or Static linkage/Static Binding/Early Binding”

2) Run Time polymorphism (Dynamic linkage/Dynamic Binding/Late Binding):

Definition: Choosing the functions at run time of program is called as “run time polymorphism or dynamic linkage/dynamic Binding/late binding”

## 6) Inheritance:

- Inheritance is a one of the most important feature of OOP, where a class can inherits data members and member function (methods) of existing class.
- Inheritance is nothing but the process of deriving (Creating) new classes from existing class. The new class is called as "Derived class or sub class or child class" where as existing class is called as "Base class or S uper class or parent class".
- In inheritance, any number of classes can be linked with one another.
- Inheritance supports for “Is-A” relationship.
- When creating a class, instead of writing completely new data members and member functions, the programmer can select that the new class should inherit the members of an existing class.
- While creating new class from existing one, the derived class accesses the data from base class and also add its own features into existing class without modifying base class. Thus, existing class is reused in derived class.
- The main concept behind inheritance is “Reusability” of code is possible. i.e. we use existing program again and again with addition of extra features.
- Inheritance reduces software developing time
- Inheritance reduces software developing cost also.
- Following figure shows concept of inheritance:



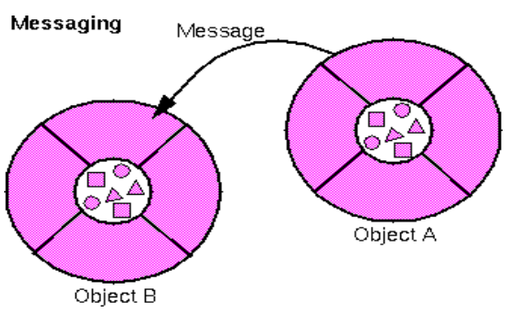

In above figure, "Vehicle" is super base class from which two classes are derived "Two wheeler" and "Four wheeler". Also, "Shine" and "Splendor" are derived classes from "Two wheeler". And "Zen", "Alto" and "Vista" are derived classes from "Four wheeler" base class.

## 7) Message Passing:

- The act of communicating with an object to get something done is called as “Message Passing”
- In message passing sending and receiving of information is done by the objects. So this helps in building real life systems. Following are the basic steps in message passing.
  - Creating classes that define objects and its behavior.
  - Creating objects from class definitions
  - Establishing communication among objects

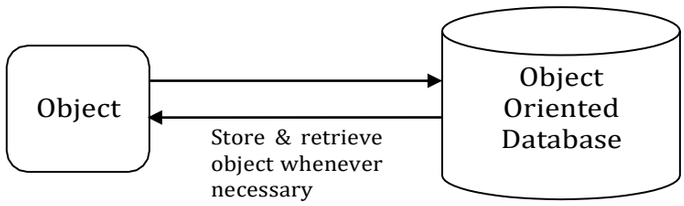
- In OOP, each object is capable of **receiving messages**, processing data, and **sending messages** to other objects and can be viewed as an independent "unit" with a distinct role or responsibility.
- Objects react when they receive messages by applying methods on themselves.
- A message is a request to an object to invoke one of its methods; in other words, a message for an object is a simply a call to one of its methods through the object.
- When objects communicate with one another, we say that they send and receive messages.
- E.g. X.getdata(a,b);

Here, we are passing a message getdata()to the objects "X " with parameters "a" and "b".Following figure shows messaging between object A and Object B:



## 8) Persistence:

- In OOP, Persistence is simply related with the "objects" that "Stick around" between the program.
- This is just serialization(It is the process of translating an "object" into a format that can be stored and reconstructed or retrieve later whenever required) an object from Object Oriented database (OO-database).
- In short, due to persistence concept, OOP language is capable to deal with the object oriented database.




# Introduction to JAVA

## JAVA is general purpose OOP language developed by "Sun Microsystems" of USA in1991

"James G osling" was the inventor (Creator) of JAVA language.

- Originally or firstly JAVA was named as "Oak" (Oak is name of tree which was found in front of Goslings Office)

Basically, JAVA was designed for the development of software"s for electronic devices like TV"s, VCR"s, set-top box, Toastersetc.

- Java runs on a variety of platforms, such as Windows, Mac OS, and the various versionsof UNIX.

The "C" and "C++" languages had limitations in terms of reliability and portability therefore they modeled their new language JAVA to overcome the drawbacks of "C" and "C++". Thus, JAVA made really simple, reliable, portable and powerful language.

### History OR Evolution of JAVA:

Following table shows some milestones happen in developing of JAVA language:

| Year | Development |
|---|---|
| 1990 | Sun Microsystem decided to develop special software that could be used to manipulate with electronic devices. A team of Sun Microsystems programmer headed by James Gosling was formed to undertake this task. |
| 1991 | After exploring the possibility of such idea, the team announced a new programming language called "Oak" (Oak was the first name for JAVA) |
| 1992 | In this year, team of Sun Microsystems actual implements there language in home appliances like Microwave Oven etc. with tiny touch-sensitive screen. |
| 1993 | In this year, team of Sun Microsystems came up with new idea to develop web based application that could run on all types of computers connected to Internet. For that, they creates 'applet' (tiny program run on Internet by the browser) |
| 1994 | In this year, team of Sun Microsystems developed web browser called "HotJava" to locate and run applet on Internet. |
| 1995 | "Oak" was renamed as "JAVA" due to some legal snags (problems). Also, many popular companies like Netscape and Microsoft announced to support for JAVA |
| 1996 | Sun Microsystem releases Java Development Kit 1.0 (JDK 1.0) to develop different kinds of software. |
| 1997 | Sun Microsystem releases Java Development Kit 1.1 (JDK 1.1) |
| 1998 | Sun Microsystem releases JAVA 2 with JDK 1.2 of Software Development Kit (SDK 1.2). |
| 1999 | Sun Microsystem releases standard Edition of Java which was called J2SE( Java 2 Standard Edition) and J2EE (Java 2 Enterprise Edition) |
| 2000 | J2SE with SDK 1.3 was released |
| 2002 | J2SE with SDK 1.4 was released |
| 2004 | J2SE with JDK 5 (JDK 1.5) was released |

- Currently Java releases their JDK 7 (JDK 1.7) version. And Java is now under administration of Oracle organization.

- Features 0r Characteristics or Advantages of Java:
- **Compiled & Interpreted:**
  - Usually, programming language is either compiled or interpreted. But Java combines both approaches that make Java "two-stage system".
  - In case of Java, First Java compiler translates or converts Java source program into "byte code" ( Byte code is not machine instruction code & byte codefile having extension ".class")
  - After compilation, Java Interpreter executes this byte code & thus we got ourdesired output.

    Thus, we can say that Java is Compiled & Interpreted language.
- **Object Oriented:**

  Java is pure object oriented language that supports for all OOP"s concepts.
  - Almost, In Java, everything is an Object. All data and methods are resided (exist in) within an object and classes.
  - The object model in Java is easy to extend because it supports for Inheritance concept.
- **Platform independent and Portable:**
  - **Portable:** We know that, after compilation of Java source program it produce ".class" file i.e. byte code which is not machine dependent that"s why such file is easily moved or transferred from one computer to another computer and hence Java is Portable.
  - **Platform independent**: After generation of byte code (.class file), this byte code is easily interpreted or executed on different kinds of computers having different platforms(Computers having different Operating system like windows, Linux, Mac OS etc and different processors etc).
- **Simple:**
  - Java is designed to be easy to learn. If you understand the basic concepts of OOP then it is easy to implement in Java language.
- **Secure:**

  We know that, most of viruses are attacked on files having extension ".exe", ".doc", ".gif", ".mpg" etc. but after compilation of Java source program it produce ".class" file i.e. byte code and which is virus free. And hence, Java enables us to develop virus-free, tamper -free systems.
- **Architectural-neutral:**
  - Java compiler generates an architecture-neutral class file format which makes the compiled code to be executable on many processors, with the presence of Java runtime system.
- **Robust:**
  - Java is strict type checking language which checks an error at both time i.e at compile time and also at run time of program.
  - Due to this ability of checking errors at run time (exception Handling), we can eliminates any risk of crashing the system using Java.
- **Multithreaded:**
  - Multithreaded means handling multiple tasks (jobs) simultaneously (at one time).
  - Java supports for multithreaded programs that means we need not wait for the application to finish its task before beginning another.
  - That is using Java, we can run multiple java applications without waiting to finish another.
- **Distributed:**
  - Java enables us to make such applications that can open and access remotely over the internet or network.
  - That is, multiple programmers at multiple remote locations are capable to work together on single project. That"s why Java is distributed.
- **Dynamic and Extensible:**
  - **Dynamic:**Java is dynamic language which is capable to link new class libraries, methods and objects dynamically.
  - **Extensible:** Java supports to write functions in C or C++ language such functions are called "native methods" and then we can add or link these methods with Java such that they can be used in many applications.
- **Ability to Deal with Database:**
  - Java supports for JDBC (Java Database Connectivity) to send & retrieve data in tabular format with the database thus with the help of Java we are able to dealwith database.
- **Automatic Memory Management:**

  We know that "memory" is very important issue while dealing with computer and we have to manage it very efficient manner.

  Java language supports for "G arbage C ollector" that automatically manages all the memory in efficient manner.

## ❖ Limitations or Disadvantages of Java:

- **Slow language:**
  - As compared to C and C++ languages, Java language compiler took much more time to compile the program & also Java interpreter took much more time to interpret the program that"s why Java is slow language.
- **Strict type checking language:**
  Due to strict type checking, Java language checks much run time errors & that"s why Java application took much time to execute.
- **Case sensitive language:**
  - Due to case sensitive language, we must have to write correct spelling of inbuilt methods, classes, interfaces etc. while doing programming.
- **Java does not support for Multiple Inheritance but we can implement it by using 'interface'.**

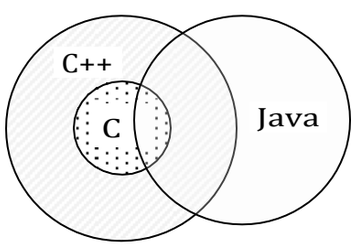
## ❖ Difference Between C and Java:

| 'C' Language | Java Language |
|---|---|
| 1) It has "sizeof" and "typedef" keywords | 1) It has not "sizeof" and "typedef" keywords |
| 2) It supports for data type "struct" and "union" | 2) It does not supports for data type "struct" and "union" |
| 3) It does not support for data type "class" | 3) It does supports for data type "class" |
| 4) It has type modifier keywords like auto, extern, register, signed and unsigned. | 4) It does not have type modifier keywords like auto, extern, register, signed and unsigned. |
| 5) It supports for "pointer" | 5) It does not supports for "pointer" |
| 6) It has preprocessor directive statements like #define, #include etc. | 6) It has no preprocessor directive statements like #define, #include etc. |
| 7) It is not OOP language. | 7) It is pure OOP language. |

## ❖ Difference Between C++ and Java:

| 'C++' Language | Java Language |
|---|---|
| 1) It supports for operator overloading. | 1) It does not supports for operator overloading |
| 2) It supports for template classes. | 2) It does not support for template classes. |
| 3) It supports for "Multiple inheritance" | 3) It does not supports for Multiple Inheritance but we implement it using "interface" |
| 4) It supports for global variable. | 4) It does not have global variable |
| 5) It supports for "pointer" | 5) It does not supports for "pointer" |
| 6) It supports for "destructor" | 6) It does not supports for "destructor" but it is replaced by finalize( ) method. |
| 7) It has "goto" statement. | 7) It has not "goto" statement. |
| 8) It has preprocessor directive statements like #define, #include etc. | 8) It has no preprocessor directive statements like #define, #include etc. |
| 9) It has three access specifiersviz: public, private and protected | 9) It has four access specifiersviz: public, Private, protected and default. |
| 10) It is not pure OOP language. | 10) It is pure OOP language. |

## Following fig. shows overlapping of C, C++ and Java:




From above fig.

- We know that, "C++" is superset of "C" language therefore every "C" program is easily executed by "C++" compiler.
- But, Java language is partly combination of "C" and "C++" language and it having its own extra features therefore Java can be considered as first cousin of "C++" and second cousin of "C"

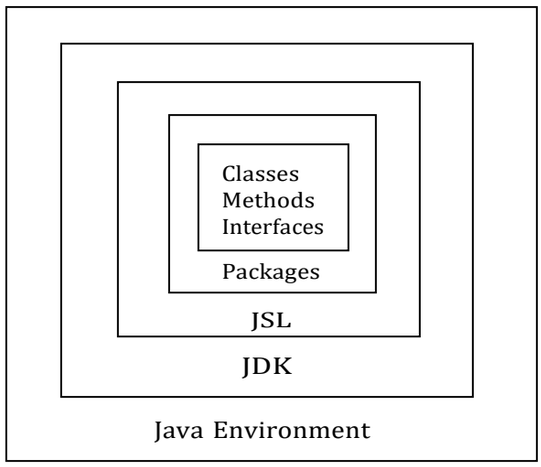
## Java Environment:

- ❖ The main part of Java Environment is JDK (Java Development Kit).
- ❖ JSL (Java Standard Library) also called as Java API (Application Programming Interface)is

the main part of JDK that contains thousands of Packages.

- Further, Packages contains thousands of classes, methods, interfaces etc.

Following Fig. shows Java Environment:




## JDK tools or Components:

Following is the list of tools or components of JDK which are used to develop and run the java programs:

| Sr.NO | Tool or Component | Description or Use |
|---|---|---|
| 1 | javac (Java Compiler) | It translates or converts java source program into byte code file & that file understood by java interpreter |
| 2 | java (java Interpreter) | It runs java applications by reading corresponding byte code file & gives result. |
| 3 | appletviewer | It runs or views java applets onto the web browser. |
| 4 | javap (Java disassembler) | It converts byte code file into program description |
| 5 | javadoc | It creates or produces HTML format documentation of java source file. |
| 6 | javah | It creates or produces header files for use of native methods. |
| 7 | jdb (Java Debugger) | It helps us to checks errors in java program. |

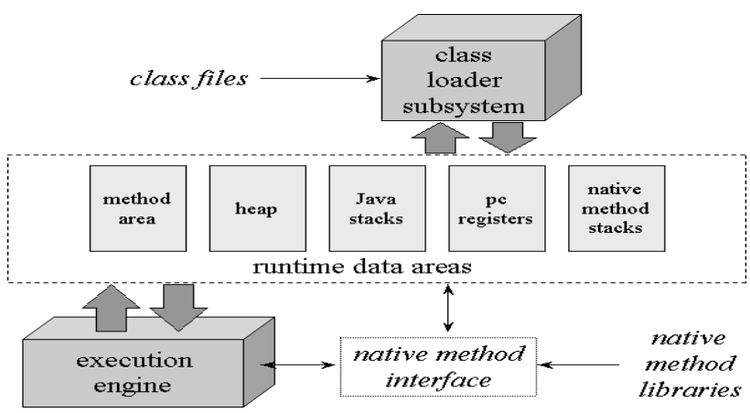
## Java Virtual Machine (JVM):

- Java Virtual Machine plays vital or important role in execution of java program thereforeit is heart of java program execution.
- JVM takes byte code file as input & convert each byte code instruction into machine language instruction such that it is easily executed by microprocessor.

**Note:**

- *Java Compiler generates or creates byte code which is machine independent or platform independent therefore it is easily interpreted by any JVM that's why it is called as "write once run anywhere".*
- *But, JVM is platform dependent i.e. windows, linux, Mac OS, Unix etc. operating system having their different-different JVM's.*

Following Figure shows Architecture of JVM:




First ofall, java sourcefile (.javafile) is converted into byte code (.class file) by the java

compiler and this byte code file isgiven to the JVM.

In JVM, there is one module or program called "Class loader sub system" which performs following functions:

- First, "Class loader sub system" loads the ".class" file into memory.
- Then it verifies whether all byte code instructions are proper or not.
- If it finds some problem in byte code then it immediately terminates the execution.
- If byte code is proper then it allocates necessary memory to execute the program.

Also, this memory is divided into 5 parts called "Runtime data area" & these parts as follows:

1) Method area:
   In this memory area; all class code, variables codes, methods codes etc. are stored.
2) Heap:
   In this memory area, all objects are created or stored.
3) Java Stacks:
   Actually, java methods are stored in "Method area" but actual execution of such java methods are happen under "Java stacks" area.
4) PC registers:
   This area contains the memory addresses of instructions of the methods.
5) Native method stacks:
   All native methods (C, C++ functions) are executed under native methods stacks.And all native methods are connected with JVM by "native method interfaces'

After, allocation of memory into corresponding parts then it comes towards "Execution Engine".

- Execution Engine can consists from two things VIZ:
  1) Interpreter 2) JIT (Just In Time) compiler.

  This interpreter and JIT compiler are responsible for converting byte code into machine instruction such that it easily executed by microprocessor.

  After, loading the ".class" file into memory, JVM first identifies which code is to be left to interpreter and which one to JIT compiler so that the performance is better. The blocks of code allocated for JIT compiler are also called "hotspots". Thus, the interpreter and JIT compiler will work simultaneously to translate the byte code into machine code.

*Note that: JIT compiler is a part of JVM which increases execution speed of program.*

## Java does not support for pointer:

1) We know that "pointers" are used to hold memory address. And most of viruses are trying to attack on memory that"s why Java does not support for pointer and hence Java is secured.
2) Also, pointers are helpful for dynamic memory allocation i.e. it is used for run time memory management, but in Java all memory management is automatically done by " "G arbage collector" that"s why not need of pointer.

### Structure of JAVA Program

- We know that single Java program may contains multiple classes but out of these classes, one class should be public and that class contains main( ) method from which JVM interprets the bytecode.
- *Note that: Java is pure OOP language i.e. all programs must have classes and objects.*

A typical Java Program is divided into several Sections which are shown in following figure:

| Documentation Section |
|---|
| Package statement Section |
| Import statement Section |
| Interface statement Section |
| Class definition section |
| main( ) method Section<br>{<br>// main() definition<br>} |

### 1) Documentation Section:

This section contains set of comments lines showing details of java source program such as program name, programmer name, date of program, version etc. this help program readability.

In Java, we can give comments by three ways VIZ:-

1) Single line comment:
   If we have to specify general information of program within single line then single line comment is used. Single line comment is given by // notation.
   Also, we can specify this comment anywhere in program.

e.g.

```
// Program Name= Addition of two numbers.
```

2) Multiline comment:

If we have to specify general information of program within multiple lines then multi line comment is used. Multiline comment is given by

```
/* ----------------
   ---------
---------- */     notation.
```

e.g.

```
/*      Program Name:   Multiplication
        Programmer: James Gosling */
```

3) Third Style comment: (Java documentation Comment)

This type of comment is specially used for documentation purpose.
If we specify description using "Third style comment" then it is shown in HTML files created by using "Javadoc"

This comment is used to provide description for every feature in Java source program. Third Style comment is given by

```
/**  ----------------------------------
     ..................................  */   notation
```

e.g.

```
/** This class is used for addition  */
public class       add
{
      /** This method is used for addition  */
        public    void addition( )
        {
          // statements
            }
    }
```

In above example, two times documentation comment is used that will show description of class "add" and description of method "addition( )" in HTML file. Note *that: For generation of HTML documentation of java source program using "javadoc" component, class and method should be public or protected.*

## 2) Package statement Section:

This section is used to declare our own package. When we declare own package then it informs to the java compiler to link all classes with our java source program.

- Syntax to specify package statement:

```
packagepackage_name;
```

e.g.

```
package       student;
```

More about package will be discussed in next chapter.

## 3) import statement Section:

In this section we can import existing package in our java source program.
We know that, in case of "C" language if we have to use printf( ) method then we include "stdio.h" header file using preprocessor directive "#include".

Similarly, if we have to use existing classes or exiting methods for JSL (Java Standard Library) then we have to import that package in our source program using "import" statement.

- Syntax to import package in program:

```
importpackage_name;
```

e.g.

```
import      java.lang.* ;
```

*Difference between #include & import:*

→

- *When we include header file in program then C/C++ compiler goes to the standard library ( it is available at c:\tc\lib) and searches for included header file there. When it finds the header file, it copies entire header file into the program where the #include statement is written. Thus, if our C/C++ program has only 10 lines still C/C++ compiler shows hundreds of line compiled this is due to copy of included header file at #include statement. Therefore our program size increases & hence it causes memory wastage.*
- *When we import package in Java program then JVM checks whether imported package is present in JSL or not. If JVM finds imported package then it executes corresponding method code there and only returns its result to source program therefore size of source program in not increased as happened in C/C++. And hence, memory wastage is solved.*

## 4) interface statement Section:

In this section we can define interfaces.
Interfaces are similar to the classes which are collections of methods etc.
This is optional section, used while implementing multiple inheritance in java.

## 5) classdefinitionSection:

We know that single Java program may contain multiple classes and every class has its own attributes (data members) and methods. Such type of classes can be defined under class definition section.

*Note that:*

- *We know that single Java program may contains multiple classes but out of these classes, one class should be public and that class contains main( ) method from which JVM interprets the byte code.*

## 6) main( ) method Section:

We know that in case of C/C++, main( ) function is compulsory from which execution of program starts. Like that java program also have main( ) method from which JVM starts program interpretation. This is compulsory section. Also, main( ) method in Java must be public. If we made main( ) as private or protected then it is not assessable for JVM also. main( ) method should be defined under any class of program.

---

## Simple Java Program:

Let"s consider following simple java program;

```
importjava.lang.*;
class           first
{
        public   static void   main(String    args[ ])
        {
                System.out.println("Welcome in JAVA programming");
        }
}
```

In above program,

"java.lang" is a package which is imported using "import" keyword. This package contains lots of inbuilt classes such as System, String, Integer, Float etc.*This is default package i.e. there is no necessary to import it.*

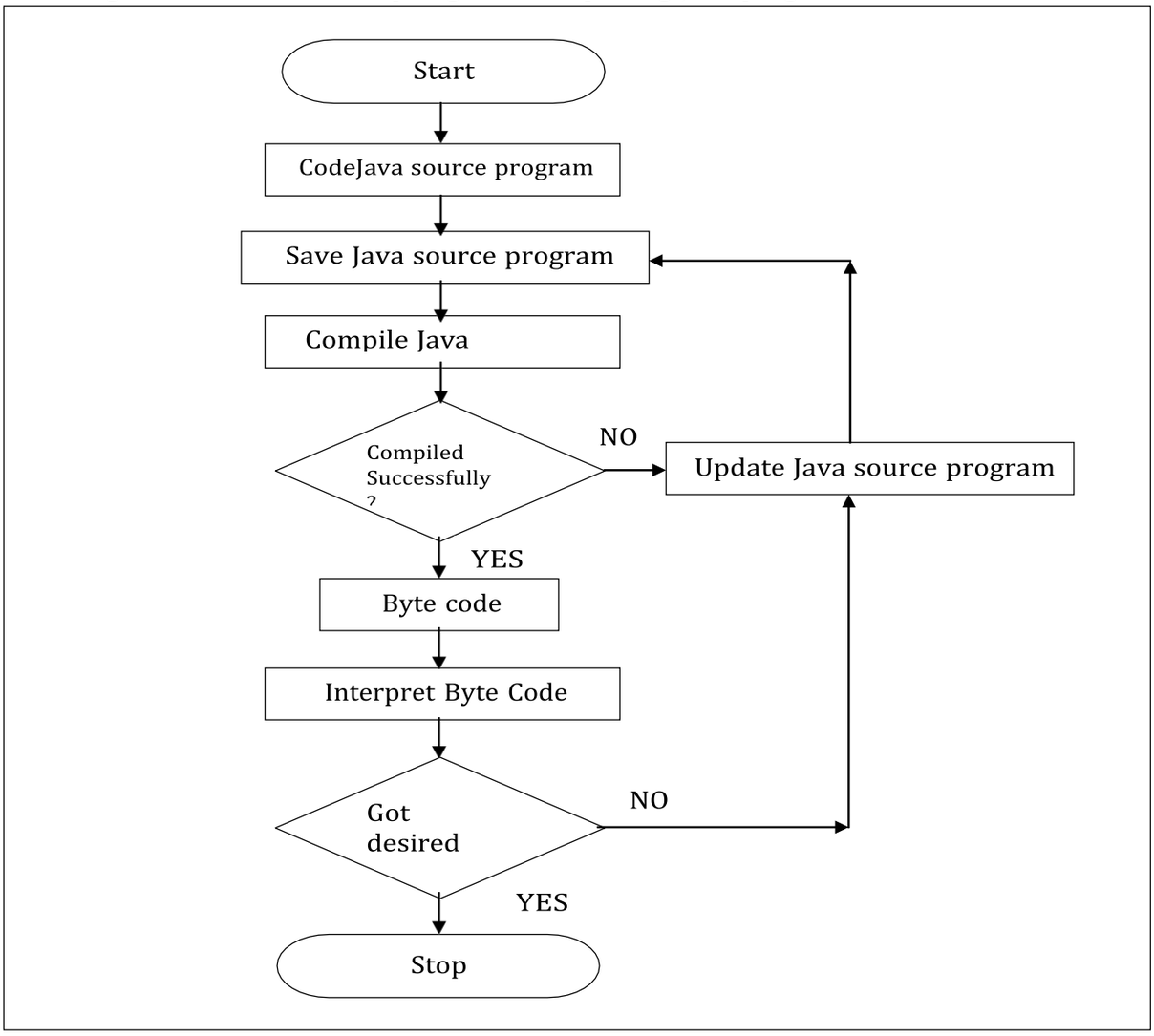
- Here, main( ) method is compulsory which is declared as *public, static and void*
  - It is ***public*** because it made available for JVM for interpretation of java program.
  - It is ***static*** because it should be called without any object; it is invoked by JVM with classname.
  - It is ***void*** because it does not return any value.
- Also, main( ) method accepts array of string as argument which is called as command line argument. The passed values are stored in args[ ] array at individual indices.
- *System.out.println( )* statement:

  "System" is inbuilt wrapper class which was found under "java.lang" package.

  "out" is object of "System" class which is "static" & hence it is accessed by "System" class name.
  - "println( )" is a method was found in "System" class used to display output and called by using "out" object.

## Steps to execute Java Program:

Following flowchart shows compiling and interpreting Java program;




## Syntax to Compile Java Source program:

Java program is compiled with "java source program" name along with "javac" component which is given as fallow:

```
javac        JavaSourcePgmName.java
```

E.g. Consider, we have "good.java" source program then we can compile it as follow:

```
javac        good.java
```

If "good.java" program have one class named "good" then "good. Class" byte code is generated.

## Syntax to Interpret or Run or Execute the Byte code:

Java program is interpreted or run or execute using bytecode (.class file) along with "java" interpreter which is given as fallow:

```
java ByteodeFile
```

E.g. Consider, we have "good.class"byte code then it isinterpreted as follow:

```
javac        good
```

*Note that: After compilation of java source program, byte code (.class file) is generated. And then JVM interpret that byte code and we got our result.*

## Syntax to pass arguments to main( ) method while interpretation ofbytecode :

We can also pass some string type arguments to main ( ) method called "command line arguments" using following syntax.

```
javac ByteCodeFile        arg1        arg2---------- argN
```

In above syntax;

arg1, arg2, - - - - ,argN are the command line arguments passed to main() method while interpreting.

*Note that: All passed arguments are stored in formal parameter (String type array) of main( ) method at individual indices.*

E.g. Consider following Program:

```
importjava.lang.*;
class           first
{
        public  static void  main(String  args[ ])
        {

                System.out.println("FirstName= "+ args[0]);
                System.out.println("MiddleName= "+ args[1]);
        System.out.println("LastName= "+ args[2]);

        }
}

OUTPUT:
                javac    first.java ↵

                java     first        SACHIN   RAMESH   SHINDE ↵
```

In above example; three command line arguments are passed to main( ) method. They are SACHIN RAMESH SHINDE.

All these arguments are stored in "args" String type array in main( ) method at individual indices as fallow;

| | |
|---|---|
| args[0]=> | SACHIN |
| args[1]=> | RAMESH |
| args[2]=> | SHINDE |

Also, we use "+" operator to concatenates two strings with each other.

# Java Source filecreation or declaration rules:

As the last part of this section let's now look into the source file declaration rules. These rules are essential when declaring classes, *import* statements and *package* statements in a source file.

- A source file can have<u>only one public class</u>.
- A source file can have multiple non-public classes.
- The <u>public class name</u> should be <u>the name of the source file</u> and which should be stored by **.java** extension.

For example : Consider there is *public class Employee{}* Then the source file should be as Employee.java

- If the class is defined inside a package, then the package statement should be the first statement in the source file.
- If import statements are present then they must be written between the package statement and the class declaration. If there are no package statements then the import statement should be the first line in the source file.
- Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file.

## <u>Java Tokens:</u>

"Token is nothing but smallest individual unit of java source program."

We know that Java is pure OOP language i.e. every program has classes and every classes has some methods and methods contains executable statement and every executable statement contains the tokens i.e. statements are made up of several tokens.

Following are the several tokens in Java program:

1) Keywords 2) Data type 3) Identifier 4) Variable
5) Constant or Literals 6) Operators 7) Special symbols.

Let us see all tokens in details:

### 1) Keywords (Reserve words):

- The words whose meaning is already known by java compiler are called as "Keywords".
- These words having fix meaning and we are not able to change that meaning therefore they are also called as "Reserve Word".
- Java language contains more than 50 keywords and they are listed as fallow:

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

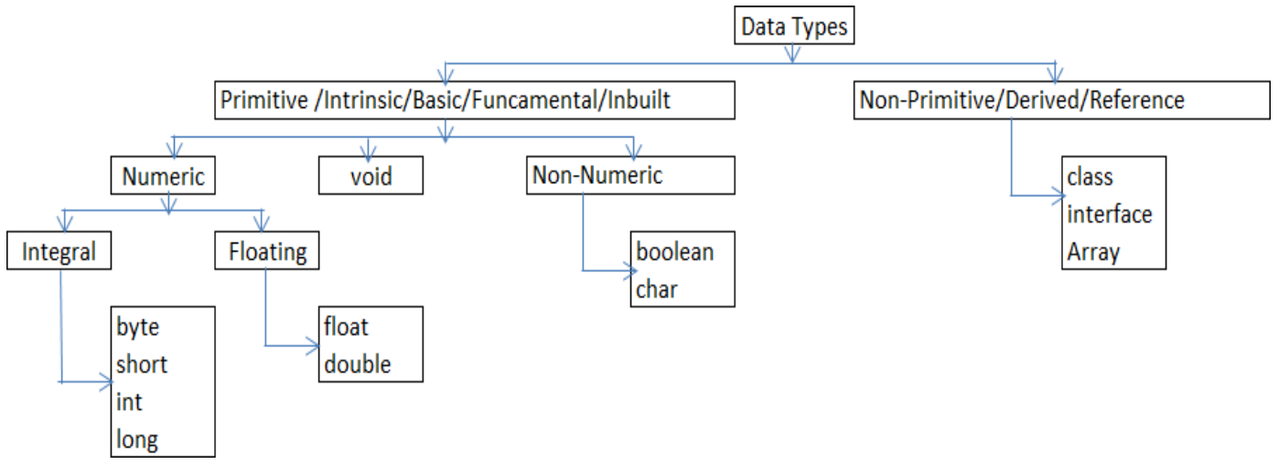
## 2) Data Type:

- Data: "Data is nothing but collection of raw information or unprocessed information that we provide for the computer for processing"
  e.g. numbers, string, alphanumeric etc.
- Data Type:
- Concept: When we give data to the computer for processing at that time compiler does not know which type of input data is.

Generally, Data types are used to tell the compiler which type of input data is.

- *Definition*: "Type of Data is called as Data Type"

Following tree diagram shows data types in Java language:




Let us see all these data types in details:

## Primitive Data Types:

There are nine primitive data types supported by Java. Primitive data types are predefined by the language and named by a keyword.

### 1) byte:

- byte data type is an 8-bit(1 byte) integral data type.
- Its Minimum range value is -128 (i.e. -2^7)
- Its Maximum range value is 127 (inclusive)( i.e. 2^7 -1)
- Its Default value is 0
- byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.
- Example: byte a = 100; byte b = -50;

### 2) short:

- short data type is a 16-bit (2 bytes) integral.
- Its Minimum range value is -32,768 (i.e. -2^15)
- Its Maximum value is 32,767 (inclusive) (i.e. 2^15 -1)
- Short data type can also be used to save memory as int data type. A short is 2 times smaller than an int
- Its Default value is 0.
- Example: short s = 10000, r = -20000;

### 3) int:

- int data type is a 32-bit (4 bytes) signed integral data type.
- Its Minimum range value is - 2,147,483,648.(i.e. -2^31)
- Its Maximum range value is 2,147,483,647(inclusive).(i.e. 2^31 -1)
- int is generally used as the default data type for integral values unless there is a concern about memory.
- Its default value is 0.
- Example: int a = 100000, b = -200000;

### 4) long:

- long data type is a 64-bit (8 bytes)signed integral data type.
- Its Minimum range value is -9,223,372,036,854,775,808.(i.e. -2^63)
- Its Maximum range value is 9,223,372,036,854,775,807 (inclusive). (i.e. 2^63 -1)

- This type is used when a wider range than *int* is needed.
- Its Default value is 0L.
- Example: long a = 100000L, int b = -200000L;

**5) float:**

- float data type is a single-precision 32-bit (4 bytes) floating data type.
- Its Minimum range value is - 3.4e38 to -1.4e-45 for negative value.
- Its Maximum range value is 3.4e38 to 1.4e-45 for positive value.
- Float is mainly used to save memory in large arrays of floating point numbers.
- Its default value is 0.0f.
- Float data type is never used for precise values such as currency.
- Example: float f1 = 234.5f;

**6) double:**

- double data type is a double-precision 64-bit(8 bytes)floating data type.
- Its Minimum range value is -1.8e308 to -4.9e324 for negative value.
- Its Maximum range value is 1.8e308 to 4.9e324 for positive value.
- This data type is generally used as the default data type for decimal values, generally the default choice.
- Double data type should never be used for precise values such as currency.
- Its Default value is 0.0d.
- Example: double d1 = 123.4;

**7) 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.
- Its default value is *false*.
- Example: booleanone = true;

**8) char:**

- char data type is a single 16-bit (2 bytes)non-numeric character data type.
- Its Minimum value is '\u0000' (or 0).
- Its Maximum value is '\uffff' (or 65,535 inclusive).
- Char data type is used to store any character.
- Example: char letter ='A';

**9) void:**

- voidmeans no value
- This data type is generally used to specify return type of method.
- If return type of method is void then that method does not return any value.

## Reference Data Types:

- There are three kinds of reference data types in Java VIZ. class, interface and array.
- If we declare variables of data type Classor interface or array then we can say that declared variables follows reference data type.*(Note that: we cannot create object of interface since they contains abstract methods but we create its reference)*
- Default value of any reference variable is null.
- A reference variable can be *used to just refer or just store any object of the declared type or any compatible type.*
- Example: 1) Consider there is one Class named "Animal" then we can create its reference as :

```
Animal monkey;
```

Here, "monkey" is reference object of "Animal" class which is used to refer another object of Animal class.

```
Animal tiger=new Animal( );
```

Here, "tiger" is an object of "Animal" class.

Then, Reference object "monkey" can be referred to "tiger" as fallow:

```
monkey = tiger ;
```

*Note that:*

** Reference object is not similar to object.*

** When we create or declare reference object of class or interface then constructor is not invoked or it is not suitable to invoke class methods. It is just used to refer or store existing object.*

** But when we create object of class then constructor automatically invoked and objects are used to invoke class methods also.*

## 3) Identifier:

- "Identifier is the name given by the programmer for any variable, package, class, interface, array, object etc."
- There are several rules to declare or define the identifier:
  1) Identifier should not be keyword.

2) Identifier should not start with digit.
3) Identifier can be combination of alphabets, digits or underscore.
4) Identifier should not contain special symbol except underscore.
5) Identifier should not contain any white space character like horizontal tab, vertical tab, new line etc.
6) Identifier should be meaningful.
7) Identifier can be of any length.

❖ Naming Conventions in Java:

*Naming Conventions* specify the rules to be followed by java programmer while writing or coding java source program.
We know that java program contains the package, classes, interfaces, methods, variables etc. and all these have separate naming conventions they are as follow:

Naming Conventions for Package:

- We know that, Package is one kind of directory that"s contains the classes and interfaces.
- *Package name in java should write in small letters only.*

Example:

java.lang
java.awt
javax.swing

Naming Conventions for class or interface:

- We know that, class is model for creating object.
- Class specifies the properties and action for objects.
- An interface is similar to class but it has abstract methods only.
- *Class and interface name in java should start with capital letter.*

Example:

System
String
Integer
Float etc.

Naming Conventions for methods:

- We know that, methods contain the executable statements or instructions after execution it produce desired result.
- *The first word of a method name is in small letters, then from second word onwards, each new word starts with capital letter as:*

Example:

println();
readLine();
getNumberInstance();

Naming Conventions for variables:

- Naming conventions for variable is same as that of methods i.e. *The first word of a variable name is in small letters, then from second word onwards, each new word starts with capital letter as:*

Example:

age;
empName;
empNetSal;

## 4) Variable:

"Variable is the name given to the memory location where the data is stored such quantity is called as Variable"

**OR**

"The quantity that changes during program execution is called as Variable"

*Concept:*

The main concept behind variable is that every variable has an ability to store the data.

Syntax to declare variable:

```
DataType    variableName ;
```

Here;
DataType is any valid data type in "Java" language.
variableName is an identifier.

Example:
```
int  rollno;
Char  x;
```

- There are several rules to declare the variable:
  1) Variable should not be keyword.
  2) Variable should not start with digit.
  3) Variable can be combination of alphabets, digits or underscore.
  4) Variable should not contain special symbol except underscore.
  5) Variable should not contain any white space character like horizontal tab, vertical tab, new line etc.

6) Variable should be meaningful.
7) Variable can be of any length.
8) Declared variable must be initialized anywhere in block.

## Types of Variables in java:

- **Local variables:**
  - The variables which are declared inside methods, constructors or blocks are called local variables.
  - These variables are declared and initialized within the method and they will be destroyed automatically when the method has completed its execution.
- **Instance variables:**
  - Instance variables are variables which are declared within a class but outside any method.
  - These variables are instantiated when the class is loaded.
  - Instance variables can be accessed from inside any method, constructor or blocks of that particular class but not accessed within static method *directly*.
- **Class variables:**
  - Class variables are variables which are declared within a class but outside any method and declared with the static keyword.
  - These types of variables are common to all objects of class i.e. all static data are shared among all objects of class commonly.

## 5) Constant (Literals):

A *literal* represent a fixed value that is stored into variable directly in the program. They are represented directly in the code without any computation.
Literals can be assigned to any primitive type variable.
For example:

- bytep = 68;
- char a = 'A';

Java has different types of literals VIZ:

1) Integer Literals
2) String Literals
3) Character Literals
4) Float Literals
5) Boolean Literals

Let us see all literals in details:

### 1) Integer Literals:

Integer literals represent the fixed integer values like 23, 78, 658, -745 etc.
The data type *byte, int, long, short*belongs to *decimal number system* that uses 10 digits ( from 0 to 9 ) or *octal number system*that uses 8 digits (from 0 to 7) or *hexadecimal number system* that uses 16 digits (from 0 to F) to represent any number.
*Note that:*
*Prefix 0 is used to indicate octal and prefix 0x indicates hexadecimal when using these number systems for literals.*
For example:

- intdecimal = 100;
- intoctal = 0144;
- inthexa = 0x64;

### 2) String Literals:

String literals are collection of characters which are representing in between a pair of *double quotes*.
Example:

- String x="Hello World";

### 3) Character Literals:

Character literals are characters which are representing in between a pair of *single quotes*.
Character literals are like "A" to "Z", "a" to "z", "0" to "9"or Unicode character like "\u0042" or escape sequence like "\n", "\b" etc.
Example:

- Char x= " Z";

### 4) Float Literals:

Float literals representsfractional values like 2.3, 86.58, 0.0, -74.5 etc.
These types of literals are used with *float* and *double* data types.
While writing such literals, we can use E or e for scientific notation, F or f for float literal and D or d for double literals (this is default and generally omitted)
For Example:

```
float   p = 9.26;
double  q = 1.56e3;
float   m =986.8f;
```

### 5) Boolean Literals:

Float literals representsonly two values – true or false. It means we can store either "true" or

"false" into a Boolean type variable
For Example:
boolean p =true;

## 6) Operators:

An"operator"is a symbol that tells computer to perform specific task.

OR

An "Operator" is a symbol that operates onto the operand to perform specific task.

Following are the several operators present in Java:

1) Arithmetic operators
2) Relational operators
3) Logical operators
4) Increment and decrement operator
5) Assignment operator
6) conditional operator
7) Bitwise operators
8) new operator
9) instanceof" operator
10) cast operator

Let"s see some operators of Java language as follows:

## 7) Bitwise operators:

- Bitwise operators are used to manipulate data at bit (0 or 1) level.
- These operators act on individual bits of the operands.
- Bitwise operators only act on integral data types such as byte, int, short, long. That is they are not worked on float and double data type.
- When these operators work on data then internally (automatically) data is converted into binary format & then they start their working.
- There are 7 different bitwise operators present in Java as follows:

| Operator | Meaning |
|---|---|
| & | Bitwise AND |
| \| | Bitwise OR |
| ^ | Bitwise exclusive OR (i.e. XOR) |
| ~ | Bitwise Complement |
| << | Bitwise left Shift |
| >> | Bitwise Right Shift |
| >>> | Bitwise Zero fill Right shift |

The truth table or working of bitwise operator & , | and ^ is shown in following table:

| Op1 | Op2 | Op1 & Op2 | Op1 \| Op2 | Op1 ^ Op2 |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 |

### Bitwise AND operator (&):

This operator performs "AND" operation on individual bits of the numbers. To understand the working of "&" operator see following example.

E.g.:

1) 22&5

→

| | | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| 22 | → | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 5 | → | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 22&5 | → | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** |

In above table, "1" bit is found in "4" column only therefore "22&5" gives result **'4'**

### Bitwise OR operator (|):

This operator performs "OR" operation on individual bits of the numbers. To understand the working of "|" operator see following example.

E.g.:

1) 35|7
→

| | | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| 35 | → | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 7 | → | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 35\|7 | → | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** |

In above table, "1" bit is found in "1", "2", "4" and "32" columns therefore "35|7" gives result
1+2+4+32= **39**

## Bitwise XOR operator (^):

This operator performs "exclusive OR" operation on individual bits of the numbers. Its symbol is denoted by "^" which is called cap, carat or circumflex symbol. To understand the working of "^" operator see following example.

E.g.:1) 47^4
→

| | | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| 47 | → | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 4 | → | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 47^4 | → | **0** | **0** | **1** | **0** | **1** | **0** | **1** | **1** |

In above table, "1" bit is found in "1", "2", "8" and "32" columns therefore "47^4" gives result
1+2+8+32=**43**

## Bitwise Complementoperator (~):

This operator gives complement form of the given number. Its symbol is denoted by "~" which is called "tiled" symbol. To understand the working of "~" operator see following example.

E.g.:

1) ~47

→ It gives result= -48

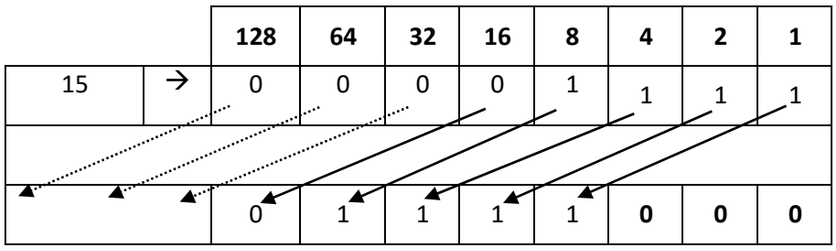
2) ~(-26)

→ It gives result= 25

## Bitwise leftshiftoperator (<<):

This operator shifts the bits towards *left side* by a specified number of positions. Its symbol is denoted by "<<" which is called double less than symbol. To understand the working of "<<" operator see following example.

E.g.:1) 15<<3
→

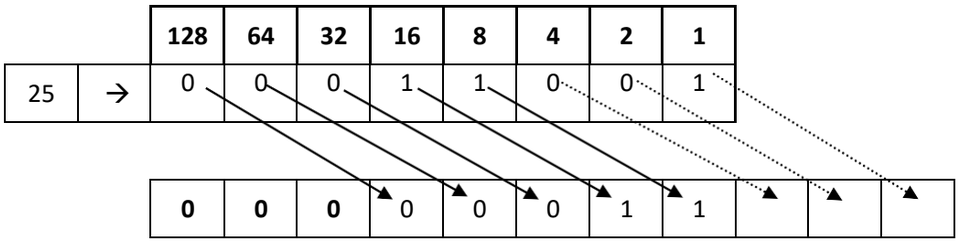



In above table, "1" bit is found in "8", "16", "32" and "64" columns therefore "15<<3" gives result
8+16+32+64=**120**

## Bitwise Rightshiftoperator (>>):

This operator shifts the bits towards *right side* by a specified number of positions. Its symbol is denoted by ">>" which is called double greater than symbol. To understand the working of ">>" operator see following example.

E.g.:1) 25>>3
→




In above table, "1" bit is found in "1", and "2" columns therefore "25>>3" gives result
1+2=**3**

## Bitwise Zero Fill Rightshiftoperator (>>>):

This operator also shifts the bits towards *right side* by a specified number of positions. But, it stores "0" in the sign bit. Its symbol is denoted by ">>>" which is called triple greater *than symbol. Since, it always fills "0" in the sign bit therefore it is called zero fill right shift operator*.

In case of negative numbers, its output will be positive because sign bit is filled with "0"

## 8) 'new' operator:

- "new" operator is used to create object of class.
- We know that, objects are created on "heap" memory by JVM dynamically.

### Syntax to create object:

```
className obj=new     className( );
```

Here,

"className" is name of the class.
"obj" is name of created object which is an identifier.

Example: Consider, there is class named "Employee" then we create its object as follow,

```
Employee emp = new Employee( );
```

Here, "emp" is an object of class "Employee"

## 9) 'instanceof' operator:

- "instanceof" operator is used to check an object belongs to class or not.
- This operator also used *to check an object belongs to interface or not*.

Syntax:

```
booleanvar=objinstanceofclassName;
                    OR
booleanvar=objinstanceofinterfaceName;
```

Here,

"var" isvariable of boolean data type.
"obj" is object of class or interface.
"className" is name of class.
"interfaceName" is name of interface.

*Example:*

boolean x=empinstanceof Employee;

Here, "instanceof" operator checks an object "emp" is an object of class "Employee" or not.

If "emp" is an object is class "Employee" then "instanceof" operator return "true" otherwise it returns "fals e"

```
// Program that demonstrate use of „instanceof" operator
class Worker
{


}
class Employee extends Worker //Inherited from Worker Class
{
        public static void main(String []args)
        {
                Employee emp=new Employee( );
                Worker wk=new Worker( );
                boolean x=wkinstanceof Employee; //checks 'wk' is an object of „Employee" class or not.
                if(x == true )
                {
                System.out.println("It is object of Employee class");
                 }
                else
        {
                System.out.println("It is object of Worker class");
                 }

        }
}
```

**OUTPUT:** It is object of Worker class (Since, „wk" is object of „Worker" class)

## 10) 'cast' operator:

castoperator is used to convert one data type into another data type.

To convert data type of any variable or an expression, just we have to specify conversion data type before variable or expression within simple bracket (braces).

Example:

**1)**

```
double x=15.26;
int y=x;          //Error- because data type of "x" and "y" are different.
```

To store value of "x" into "y", we have to convert data type of "x" into data type of "y" as follow,

int y= (int) x; //here, the data type of "x" is converted into data type of "y" using ***(int) cast operator***

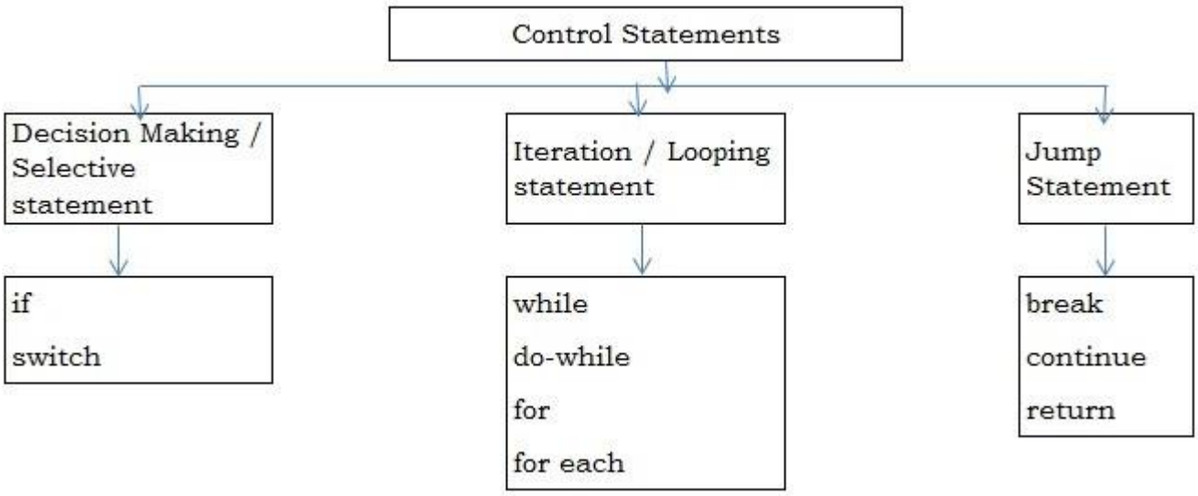
**2)**

```
int x=65;
char y = (char) x;//here, the data type of "x" is converted into data type of "y" using (char) cast operator
```

## Control Statement in Java:

The statement that controls the flow of execution of program is called as "Control statement" or "Control Structure".

The following tree diagram shows control statements in Java language:




*(Note that: All the **Control Statement in Java is same as that of C/C++** language therefore refer notes of C/C++ language)*

**for-each loop:**

- This loop is specially designed to handle elements of "collection".
- *Collection* represents a group or set of elements or objects.

For example: We can take an "array" as collection because "array is set or group of elements

- Also, any class in "java.util" package can be considered as "collection" because any class in "java.util" package handles group of objects such as "stack", "vector", "LinkedList" etc.
- The *for-each* loop repeatedly executes a group of statements for each element of the collection.
- The execution of for-each loop depends upon total number of elements or objects present in the collection.

## Syntax:

```
for (datatypevar : collection )
{
        Statements;
}
```

Here,
"var" is an identifier which represents each element of collection one by one. Suppose, the collection has 5 elements then this loop will be executed 5 times and "var" will store each element of collection one by one.
"datatype" is any valid datatype in Java which is same as collection.
"collection" is any collection such as array, stack, linked list, vector etc.

```
// Program that demonstrate use of for-each loop
classmyloop
{
        public  static void main(String   []args)
        {
                intarr[]={5,6,7,8,9};

                for (int  i : arr )          // „i' represents each element of „arr"
        {
                System.out.println(i);
              }
        }
}
```

## 'continue'statement:

"continue"statement is specially used in looping statement.
When "continue" statement is executed then control transferred back to check the condition *in loop* and rest of statements are ignored.

```
// Syntax or execution of „continue"  statement

While  (  condition1  )                 <.....
{
                if  (  condition2  )
                {
                continue;          .....
                }


}
```

---
---

```
// Program that demonstrate use of "continue" statement
classmyloop
{
        public  static void main(String   []st)
        {
                int      i=10;
                while(i>=1)
                {
                        if(i>5)
                            {
                                System.out.print("\t"+i);
                                i--;
                                continue;
                            }
                        else
                            {
                                i--;
                            }
                }


        }
```

**Reading Inputes by Scanner class**

We can read varieties of inputs from keyboard or from text file using methods of "Scanner" class.

Scanner class belongs to "java.util" package.

When Scanner class receives input, it breaks the input into several pieces, called "tokens" and these tokens can be retrieved using object of *Scanner* class.

- Note that: Following methods of *Scanner* class are *non-static* therefore they are called or accessed with the help of *object* of *Scanner* class.

We can create object of Scanner class as follows:

```
Scanner      obj=new    Scanner(System.in);
```

Here, "obj" is object of Scanner class.

"System.in" represents InputS tream object, which is by default represents standard input device i.e. Keyboard.

There are several methods of *Scanner* class used to take different inputs as follows:

| Method | Working |
|---|---|
| next( ) | It is used to read single string |
| nextByte( ) | It is used to read single byte type value |
| nextInt( ) | It is used to read single integer type value |
| nextFloat( ) | It is used to read single Float type value |
| nextLong( ) | It is used to read single Long type value |
| nextDouble( ) | It is used to read single Double type value |
| nextShort( ) | It is used to read single short type value |

Following program demonstrate the use of different methods of *Scanner* class.

```
importjava.util.Scanner;
class    cricket
{
        public static void main(String []args)
        {
                byte    no;
                String name;
                long    contact;
                int      t_sc;
                shortt_wk;
                floatball_avg;
                double  bat_avg;
                Scanner sc=new Scanner(System.in);
                System.out.print("Enter Cricketer No= ");
                no=sc.nextByte();
                System.out.print("Enter Cricketer Name= ");
                name=sc.next();
                System.out.print("Enter Cricketer Contact No= ");
                contact=sc.nextLong();
                System.out.print("Enter Cricketer Total Score= ");
                t_sc=sc.nextInt();
                System.out.print("Enter Cricketer Wickets= ");
                t_wk=sc.nextShort();
                System.out.print("Enter Ball AVG= ");
                ball_avg=sc.nextFloat();
                System.out.print("Enter Batting AVG= ");
                bat_avg=sc.nextDouble();
                System.out.println("..........");
                System.out.println("CricketerNO="+no);
                System.out.println("CricketerName="+name);
                System.out.println("ContactNo="+contact);
                System.out.println("Total SCore="+t_sc);
                System.out.println("Total Wickets="+t_wk);
                System.out.println("Balling AVG="+ball_avg);
                System.out.println("Batting AVG="+bat_avg);
          }
}
```

# Array

"An Array is collection of homogeneous (having same type) elements or items referred by common name"

In an array the individual element is accessed with the help of integer value and is called **subscript or index.** Also all elements of array are stored in **continuous memory** allocation.

*Note that:*

- *We know that, In C/C++ language, memory for array is get allocated at compile time (i.e. static memory allocation)*
- *But, in JAVA everything is dynamic i.e. for variable, array, objects etc. memory is get allocated at run time (Dynamic memory allocation) by JVM*

## Types of Array:

Depending on number of subscripts used in array, array having three types:

### 1) One Dimensional array: (1 D array)

"The array having **only one subscript** is called as *One dimensional array*"

Declaration Syntax:

```
datatype    array_name[ ]=new datatype [Size];
                        OR
datatype    [ ]array_name=new  datatype  [Size];
```

here;
*datatype* is any valid *datatype* in JAVA language
*array_name*is name of array which is an identifier.
"s ize" is integer value that denotes total number of elements stored in array
"new" is an operator.
e.g.

*1)* int x[ ]=new int[5];//declares array "x" & allocates memory for 5 integers

**OR**

int [ ]x =new int[5];

here "x" is array which can holds(stores) **five**integers at a time.

*2)* int marks[ ];//declares array "marks"
marks=new int[5]; //*allocates memory for 5 integers*

### Initialization of One dimensional array:

One dimensional array can be initialized as fallow;

e.g. int p[] = { 10 , 20 , 30 , 40 , 50 } ;

here; "p" is integer type array which stores five integers at a time. The storage of all elements in array "p" is shown in following figure:

| Index → | *0* | *1* | *2* | *3* | *4* |
|---|---|---|---|---|---|
| Elements of 'P' | **10** | **20** | **30** | **40** | **50** |
| Address → | 110 | 114 | 118 | 122 | 126 |

In above figure the elements 10, 20, 30, 40, 50 are stored in array "p" at individual index.
That is the element 10 is stored at index 0, 20 is stored at index 1 like that, 50 is stored at index 4. Also all elements are stored in continuous memory location.
*Note: Index is nothing but position of the element in an array.*

Example of one diamensional array

```
import java.util.*;
class A
{
   Scanner sc=new Scanner(System.in);
   int x[]=new int[3];
   int i;
   void show()
   {
   System.out.println("enter array elements");
      for(i=0;i<3;i++)
      {
         x[i]=sc.nextInt();
      }
      System.out.println(" array elements are-:");
      for(i=0;i<3;i++)
      {
         System.out.print("\t"+x[i]);
      }
   }

}
class demo
{
   public static void main(String[] args)
   {
     A p=new A();
     p.show();
   }
}
```

## 2) Two Dimensional array: (2D array)

" The array having **two subscripts**is called as Two dimensional array or matrix"
The two dimensional array is used to perform matrix operations.

*Declaration Syntax*:

```
datatype      array_name[ ][ ]=new  datatype  [Size1][Size2];
                                  OR
datatype      [ ][ ]array_name=new  datatype  [Size1][Size2];
```

here;
*datatype* is any valid *datatype* in Java language
*array_name*is name of array which is an identifier.
"S ize1" is integer value that denotes total number of rows in array
"S ize2" is integer value that denotes total number of columns in array
e.g.

1) int x[ ][]=newint[3][2]; **OR** int [ ][ ] x=newint[3][2];

here ;
"x" is array which can holds total 6 integers at a time.
3 is rowsize i.e. there are 3 rows in matrix "x"
2 is columnsize i.e. there are 2 columns in matrix "x"

*Initialization of Two dimensional array:*

Two dimensional array can be initialized as follow;

e.g.

1) int p[ ][ ]={{3,4},{2,9},{7,6}};

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

"z" is integer type two dimensional array which stores nine integers at a time. The storage
here;
of all elements in array "z" is shown in following figure:

| row index | 0 | 1 | 2 | column index |
|---|---|---|---|---|
| **0** | 5 | 3 | 6 | |
| **1** | 1 | 7 | 8 | |
| **2** | 9 | 4 | 2 | |

In above figure the individual element of array "z" is also accessed by following way:

The element 8 can be accessed as z[1][2]
The element 7 can be accessed as z[1][1]
The element 9 can be accessed as z[2][0] etc.

like that we can access all individual elements of matrix "z"

```
import java.util.*;
class A
{
   Scanner sc=new Scanner(System.in);
   int x[][]=new int[2][2];
   int i,j;
   void show()
   {
   System.out.println("enter array elements");
      for(i=0;i<3;i++)
      {
      for(j=0;j<3;j++)
      {
       x[i][j]=sc.nextInt();
      }
      }
      System.out.println(" array elements are-:");
      for(i=0;i<3;i++)
      {
      for(j=0;j<3;j++)
      {
         System.out.print("\t"+x[i][j]);
      }
      System.out.println("");
      }
   }
```

```
class demo
{
   public static void main(String[] args)
   {
     A p=new A();
     p.show();
   }
}
```

## Multi-Dimensional array:

"The array having **more than two subscripts**is called as *multi-dimensional* array"

Declaration Syntax:

```
datatype    array_name[ ][ ]……[ ]=new datatype [Size1][Size2]……[SizeN];
                                  OR
datatype    [ ][ ]……[ ]array_name=new datatype [Size1][Size2]……[SizeN];
```

here;

*datatype* is any valid *datatype* in Java language.
*array_name* is an identifier (name given by programmer)
*size1, size2,……sizeN* are **integer constants** and that denotes total number of elements stored in an array.

e.g. 1) int z[][][]=newint[2]**[4][3]**;

The above array is the multidimensional array having three subscripts and which stores 12 integer values at a time.

*That is above multi-dimensional array stores **2 sets of 4*3 matrices**.*

2) float x[ ][][][]=new float[2][3][2][2];

The above array "x" is also multidimensional array having four subscripts and which stores 24 float values at a time.

*Initialization of Multi-dimensional array:*

Multi-dimensional array can be initialized as follows:

Example:

```
1) int  x[][][]={{{1,2},{4,5},{7,8},{3,6}},
                 {{11,12},{14,15},{17,18},{13,16}} };
```

In above multi-dimensional array, elements are stored in following manner;

x[0] ⇨

| | *0* | *1* |
|---|---|---|
| *0* | **1** | **2** |
| *1* | **4** | **5** |

| | | |
|---|---|---|
| 2 | 7 | 8 |
| 3 | 3 | 6 |

x[1] ⇨

| | 0 | 1 |
|---|---|---|
| 0 | 11 | 12 |
| 1 | 14 | 15 |
| 2 | 17 | 18 |
| 3 | 13 | 16 |

From above, multi-dimensional array we can access individual element as follow:

X[0][1][0] access 4
X[0][2][1] access 8
X[0][3][0] access 3
X[1][1][1] access 15
X[1][2][0] access 17
X[1][3][1] access 16

## 'length' property of array:

"length" is property of an array that returns one integer value which is nothing but ***size** of array*.

We use this property as follows:

intvar=arrayname.length;

Here, "var" is an int type variable which stores size of array return by arrayname.lengthpoperty.

Exampl:

*1)*
```
intarr[ ]=new int[10];    //declares array with 10 size
arr[0]=45;
arr[1]=90;
intsz=arr.length;      //returns array size
System.out.print("Array Size="+sz);
```

OUTPUT:

Array Size=10

Note that: In above example "arr" is array declared with 10 size. And arr[0] and arr[1] are initialized with values 45 and 90 respectively. But "arr.length" statement returns value 10 which is size of array (Since, "length" property returns ***size of array**.It not returns array elements*)

*Also, in case of 2D or Multi-dimensional array, "length" property returns number of rows of the array.*

*Following program shows use of "length" property of array.*

```
classarrsize
{
    public static void main(String s[])
    {
        int a[ ]={7,3,9,7,2,4};
        int b[ ][ ]=new int[5][7];
        int c[ ][ ][ ]=new int[9][3][2];
        intx,y,z;
        x=a.length;
        y=b.length;
        z=c.length;
        System.out.println("1D array size="+x);
        System.out.println("2D array size="+y);
        System.out.println("Multi array size="+z);
    }
}
```

OUT PUT: 1D array size=6
2D array size=5
Multi array size=9

## Jagged Array:

- Jagged array is special array found in Java, which can stores group of different types of arrays in it.That is, Jagged array can stores 1D, 2D or multi-dimensional arrays of different sizes.
- When we create jagged array then other arrays can become an elements of created jagged array.
- Jagged array are also called as "Irregular multi-dimensional" array.
- Jagged array is helpful when dealing with group of arrays with different sizes.
- Following example shows jagged array:

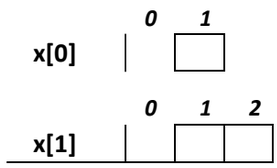
1) int x[ ][ ]=new [2]**[ ]**;
Here, "x" is jagged array having size 2i.e it stores two 1D arrays. So its elements will

stores from x[0] and x[1].
In given expression: intx[ ][ ]=new int[2]**[ ]**;

Here, last pair of brace or bracket is empty. It means that it is jagged array *for 1D array*. And therefore*x[0] and x[1]* can individually stores 1D array separately.
After declaration of jagged array, we have to allocate memory for x[0] and x[1] which is given as follow:

```
x[0]=new int[2];
x[1]=new int[3];
```

Here,
x[0] can have 2 elements which can stores in "x" at x[0][0] and x[0][1]
x[1] can have 3 elements which can stores in "x" at x[1][0] , x[1][1] and x[1][2]
And complete jagged array "x" is shown as follow:




2) double z[ ][ ][ ]=newdouble [3][ ][ ];

Here, "z" is jagged array having size 3i.e it stores three 2D arrays. So its elements will stores from z[0] , z[1] and z[2].
In given expression: double z[ ][ ][ ]=new double[3]**[ ][ ]**;

Here,Two last pair of brace or bracket is empty. It means that it is jagged array*for 2D array*. And therefore*z[0], z[1] and z[2]* can individually stores 2D array separately.
After declaration of jagged array, we have to allocate memory for z[0], z[1] and z[2] which is given as follow:

```
z[0]=new double[2][2];
z[1]=new double[3][2];
z[2]=new double[3][3];
```

Here,
z[0] can stores a matrix of order 2*2 as:

```
z[0][0][0]  z[0][0][1]
z[0][1][0]  z[0][1][1]
```

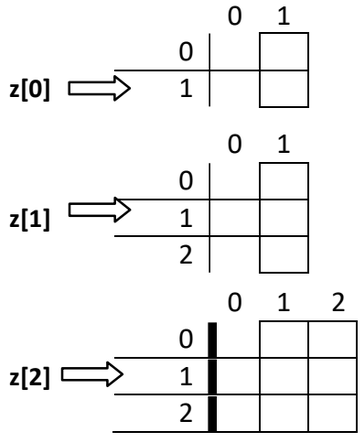
z[1] can stores a matrix of order 3*2 as:

```
z[1][0][0]  z[1][0][1]
z[1][1][0]  z[1][1][1]
z[1][2][0]  z[1][2][1]
```

z[2] can stores a matrix of order 3*3 as:

```
z[2][0][0]  z[2][0][1]  z[2][0][2]
z[2][1][0]  z[2][1][1]  z[2][1][2]
z[2][2][0]  z[2][2][1]  z[2][2][2]
```

And complete jagged array "z" is shown as follow:



```
//Program of jagged array for 1-D array


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

int     z[ ][ ]=new     int [2][ ];

        z[0]=new     int[4];
        z[1]=new     int[2];

z[0][0]=10;
z[0][1]=11;
z[0][2]=12;
z[0][3]=13;

        for(int i=0;i<=3;i++)
        {

System.out.print("\t"+z[0][i]);
        }

        System.out.println();
        System.out.println();

z[1][0]=5;
z[1][1]=6;

        for(int i=0;i<=1;i++)
        {

System.out.print("\t"+z[1][i]);

        }

        }
}
```

```
//Program of jagged array for 2-D array

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

        int     z[ ][ ][ ]=new     int [3][ ][ ];

        z[0]=new     int[2][2];
        z[1]=new     int[3][2];
        z[2]=new     int[3][3];

z[0][0][0]=10;
z[0][0][1]=11;
z[0][1][0]=12;
z[0][1][1]=13;
        for(int i=0;i<=1;i++)
        {
                for(int j=0;j<=1;j++)
                {
                        System.out.print("\t"+z[0][i][j]);
                }
                System.out.println();
        }
        System.out.println();
        System.out.println();

z[1][0][0]=5;
z[1][0][1]=6;
z[1][1][0]=7;
z[1][1][1]=8;
z[1][2][0]=9;
z[1][2][1]=4;
for(int i=0;i<=2;i++)
        {
                for(int j=0;j<=1;j++)
                {
                        System.out.print("\t"+z[1][i][j]);
                }
                System.out.println();
        }
        System.out.println();
        System.out.println();
z[2][0][0]=11;
z[2][0][1]=22;
z[2][0][2]=33;
z[2][1][0]=44;
z[2][1][1]=55;
z[2][1][2]=66;
z[2][2][0]=77;
z[2][2][1]=88;
z[2][2][2]=99;
for(int i=0;i<=2;i++)
        {
                for(int j=0;j<=2;j++)
                {
                        System.out.print("\t"+z[2][i][j]);
                }
                System.out.println();
        }
}
}
```

# STRING

## *Introduction:*

Most of data that transmit from one place to another place is in the form of group of characters. Such *group or set of characters are called as "String"*.

We know that, in case of C/C++ language string is nothing but "array of characters" & that terminates with "\0" character i.e. NULL character. But this is not true in Java language.

In JAVA, *"String is nothing but Object of String class*" and which is not array of character. Also, in JAVA we can create array of character but it is not treated as string.

While dealing with string, JAVA language supports special class called "String" class which was found under "java.lang" package.

## Creating Strings in Java:

We can create Sting in Java by three ways:

- We can create a String by assigning group of characters to a String type object:

```
String    str;    //declare string type object
str= "Hello";     //assign a group of characters to it
```

Above two statements can be combined as follow:

```
String    str= "Hello";
```

In this case, JVM creates an object & stores the string "Hello"

We can create an object of String class by using "new" operator:

```
String    str = new    String("Hello");
```

In this case, First we create object "s tr" using new operator and then we store "Hello" string in it.

- Third way of creating String is by converting array of character into String:

```
char      arr ={"H", "e", "l", "l", "o"};    //create character array.
String    str=new   String(arr);    //creating string "str" by passing "arr" to it.
```

## Difference between object & Reference object:

| Object | Reference Object |
|---|---|
| 1) This type of object is created using "new" operator.<br>e.g. String str=new String( ); | 1) This type of object is not created using "new" operator.<br>e.g. String str; |
| 2) S uch type of object is stored in "Heap" section by "class loader sub system" of JVM. | 2) Such type of object is stored in "Method area" section by "class loader sub system" of JVM. |
| 3) When such object is created then, Constructor is implicitly invoked. | 3) When such object is created then, Constructor is NOT implicitly invoked. |
| 4) When such object is created then JVM allocated separate memory location to each object. | 4) When such object is created then JVM insert it into "*String constant pool*" (Special memory block where String references are stored) |

## String Class Methods:

While dealing with Strings, there are several methods belong to String class:

*Note that: Following methods of String class are non-static therefore they are called with object of String class.*

### 1) length ( ) :

- This method is used to find length of string.
- This method returns one integer value which is length (total characters) of string.

Syntax:

```
int    len = s.length( );
```

Here, "s" is an object of String class and that contains the string.

*" len" is integer variable used to store length of string.*

E.g.

```
class   stringLen
{
        public static void main(String args[ ])
        {
                String  str=new  String("Hello");
                int     len=str.length( );
                System.out.println("Length="+len);
        }
}
OUTPUT:
                Length=5
```

## 2) concat ( ) :

- This method is used to concatenate two strings together.
- That is, it concatenates second string at the end of first string and returns concatenated string as a result.

Syntax:

```
String     str = s1.concat(s2);
```

Here, "s1" is an object of String class and that contains the first string.
"s2" is also object of String class and that contains the second string.
"str" is also string that stores concatenated string.

E.g.

```
class    stringCat
{
        public  static  void  main(String    args[ ])
        {
                String    s1= “Delhi”;
                String    s2= “Mumbai”;
        String    str=s1.concat(s2);
                System.out.println(str);
        }
}
OUTPUT:                DelhiMumbai
```

## 3) charAt ( ) :

- This method accepts one integer value and returns one character from string corresponding to passed integer value
- That is, it accepts index value (position of element) and returns corresponding character from string.

Syntax:

```
char     ch = s.charAt(pos);
```

Here, "s" is an object of String class and that contains the string.
"pos" is an integer value.
"ch" is char variable which stores returned character from string at position "pos"

E.g.

```
class    stringChar
{
        public  static  void  main(String    args[ ])
        {
                String    s= “Delhi”;
                char    ch=s.charAt(3);
                System.out.println(ch);
        }
}
OUTPUT:
        h
```

## 4) equals( ) :

- This method is also used to compare two strings with each other for equality.
- This method returns boolean value depending upon Strings content.

It returns boolean value "true" if both strings are equal otherwise it returns "false".

Syntax:

```
boolean     m = s1.equals(s2);
```

Here, "s1" is an object of String class and that contains first string.
"s2" is also an object of String class and that contains second string.
"m" is boolean variable to store returned value.

E.g.

```
class    stringEqual
{
        public static void main(String    args[ ])
        {
                String    s1= “box”;
                String    s2= “BOX”;
                boolean  m= s1.equals(s2);
                if (m = = true)
                        System.out.print(“Both strings are equal”);
                else
                        System.out.print(“Strings are not equal”);
        }
}
OUTPUT:          Strings are not equal
```

**5)** equalsIgnoreCase( ) :

- This method is also used to compare two strings with each other for equality. But it ignores the case of characters present in strings. i.e. it treated "box" string same as "BOX"
- This method returns boolean value depending upon Strings content.
  It returns boolean value "true" if both strings are equal otherwise it returns "false".

Syntax:

```
boolean     m = s1.equalsIgnoreCae(s2);
```

Here, "s1" is an object of String class and that contains first string.
"s2" is also an object of String class and that contains second string.
"m" is boolean variable to store returned value.

E.g.

```
class    stringEqual
{
        public static void main(String    args[ ])
        {
                String    s1= "box";
                String    s2= "BOX";
                boolean m= s1.equalsIgnoreCase(s2);
                if (m = = true)
                        System.out.print("Both strings are equal");
                else
                        System.out.print("Strings are not equal");
        }
}
OUTPUT:        Both Strings are equal
```

**6)** compareTo ( ) :

- This method is used to compare two strings with each other for equality.
- This method returns one integer value depending upon Strings content.
  - ❖ It returned value is Zero then both strings are equal.
  - ❖ If returned value is positive then first string is greater than second string.
  - ❖ If returned value is negative then Second string is greater than first string.

Syntax:

```
int     m = s1.compareTo(s2);
```

Here, "s1" is an object of String class and that contains first string.

"s2" is also an object of String class and that contains second string.

"m" is integer variable to store returned value.

E.g.

```
class    stringCmp
{
        public static void main(String    args[ ])
        {
                String    s1= "abc";
                String    s2= "abd";
                int m= s1.compareTo(s2);
                if (m = = 0)
                        System.out.print("Both strings are equal");
                else if( m > 0)
                        System.out.print("First string is greater");
                else
                        System.out.print("Second string is greater");
        }
}
OUTPUT:        Second string is greater
```

**7)** compareToIgnoreCase( ) :

- This method is also used to compare two strings with each other for equality. But it ignores the case of characters present in strings. i.e. it treated "box" string same as "BOX"
- This method returns one integer value depending upon Strings content.
  - ❖ If returned value is Zero then both strings are equal.
  - ❖ If returned value is positive then first string is greater than second string.
  - ❖ If returned value is negative then Second string is greater than first string.

Syntax:

```
int     m = s1.compareToIgnoreCase(s2);
```

Here, "s1" is an object of String class and that contains first string.
"s2" is also an object of String class and that contains second string.
"m" is integer variable to store returned value.

E.g.

```
class    stringCmp
{
        public  static  void  main(String    args[ ])
        {
                String    s1= "box";
                String    s2= "BOX";
                int m= s1.compareToIgnoreCase(s2);
                if (m = = 0)
                        System.out.print("Both strings are equal");
                else if( m > 0)
                        System.out.print("First string is greater");
                else
                        System.out.print("Second string is greater");
        }
}
OUTPUT:
          Both strings are equal
```

**8)** startsWith( ) :

- This method returns boolean value "true" if first string starts with second string otherwise it returns "false".

Syntax:

```
boolean     m = s1.startsWith(s2);
```

Here, "s1" is an object of String class and that contains first string.
"s2" is also an object of String class and that contains second string.
"m" is boolean variable to store returned value.

E.g.

```
class    stringStart
{
        public  static  void  main(String    args[ ])
        {
                String    s1= "Box is heavy";
                String    s2= "Box";
                boolean m= s1.startsWith(s2);
                if (m = = true)
                        System.out.print("First string start with second string");
                else
                        System.out.print("First string NOT start with second string");
        }
}
OUTPUT:       First string start with second string
```

**9)** endsWith( ) :

- This method returns boolean value "true" if first string ends with second string otherwise it returns "false".

Syntax:

```
boolean     m = s1.endsWith(s2);
```

Here, "s1" is an object of String class and that contains first string.
"s2" is also an object of String class and that contains second string.
"m" is boolean variable to store returned value.

E.g.

```
class    stringEnd
{
        public  static  void  main(String    args[ ])
        {
                String    s1= “Box is heavy”;
                String    s2= “heavy”;
                boolean m= s1.startsWith(s2);
                if (m = = true)
                        System.out.print(“First string ends with second string”);
                else
                        System.out.print(“First string NOT ends with second string”);
        }
}
OUTPUT:        First string ends with second string
```

## 10) indexOf( ) :

- This method returns integer value which is nothing but first position of substring into main string.
- If substring is not found in main string then it returns negative value.

Syntax:

```
int    m = s1.indexOf(s2);
```

Here, "s1" is an object of String class and that contains first string.
"s2" is also an object of String class and that contains second string.
"m" is integer variable to store returned value.

E.g.

```
class    stringFirstIndex
{
        public  static  void  main(String    args[ ])
        {
                String    s1= “Box is heavy and it is dirty”;
                String    s2= “is”;
                int     m= s1.indexOf(s2);
                if (m < 0 )
                        System.out.print(“Substring Not found”);
                else
                        System.out.print(“Substring found at= ”+m);
        }
}
OUTPUT:        Substring found at= 4
```

## 11) lastIndexOf( ) :

- This method returns integer value which is nothing but last position of substring into main string.
- If substring is not found in main string then it returns negative value.

Syntax:

```
int    m = s1.lastIndexOf(s2);
```

Here, "s1" is an object of String class and that contains first string.
"s2" is also an object of String class and that contains second string.
"m" is integer variable to store returned value.

E.g.

```
class    stringLastIndex
{
        public  static  void  main(String    args[ ])
        {
                String    s1= “Box is heavy and it is dirty”;
                String    s2= “is”;
                int     m=  s1.lastIndexOf(s2);
                if (m < 0 )
                        System.out.print(“Substring Not found”);
                else
                        System.out.print(“Substring found at= ”+m);
        }
}
OUTPUT:        Substring found at= 20
```

## 12) replace( ) :

- This method replaces characters of existing string with new given character.

Syntax:

```
String    str = s.replace(old_char,new_char);
```

Here, "old_char" is character to be replaced by "new_char" of string.
"str" is String variable to store returned value.

E.g.

```
class    stringReplace
{
        public  static  void  main(String    args[ ])
        {
                String    s= “Box is heavy”;
                String    str= s.replace("B","D");
                System.out.print(str);
        }
}
OUTPUT:        Dox is heavy
```

## 13) substring( ) :

- This method has two forms:-

### I) substring(int):

- This method returns a new string consisting of all characters from *given position* to the *end of string*.

Syntax:

```
String    str  =  s.substring(pos);
```

Here, "s" is an object of String class and that contains string.
"pos" is an integer value from which new string is to be started.
"str" is string object to store resultant string.

E.g.

```
class    stringSubstr
{
        public  static  void  main(String    args[ ])
        {
                String    s= “Box is heavy”;
                String    str= s.substring(4);
                System.out.print(str);
        }
}
OUTPUT:        is heavy
```

### II) substring(int,int):

- This method returns a new string consisting of all characters from *given first position* to ***the last position -1***.

Syntax:

```
String    str = s.substring(pos1,pos2);
```

Here, "s" is an object of String class and that contains string.
"pos1" is an integer value i.e. firs t position from which new string is to be started.
"pos2" is an integer value i.e. last position and new string ends at last postion-1.
"str" is string object to store resultant string.

E.g.

```
class    stringSubstr1
{
        public  static  void  main(String    args[ ])
        {
                String    s= “Box is heavy”;
                String    str= s.substring(7,11);
                System.out.print(str);
        }
}
OUTPUT:        heav
```

## 14) toLowerCase( ) :

- This method converts all characters of string into *lower case* and returns that lower-cased string as result.

Syntax:

```
String    str = s.toLowerCase( );
```

Here, "s" is an object of String class and that contains string.
"str" is also an object of String that stores returned lower-cased string.

E.g.

```
class    stringLower
{
        public  static  void  main(String    args[ ])
        {
                String    s= “BOX IS HEAVY”;
                String    str= s.toLowerCase( );
                System.out.print(str);
        }
}
OUTPUT:        box is heavy
```

## 15) toUpperCase( ) :

- This method converts all characters of string into *upper case* and returns that upper-cased string as result.

Syntax:

```
String    str = s.toUpperCase( );
```

Here, "s" is an object of String class and that contains string.
"str" is also an object of String that stores returned upper-cased string.

E.g.

```
class    stringUpper
{
        public  static  void  main(String    args[ ])
        {
                String    s= “box is heavy”;
                String    str= s.toUpperCase( );
                System.out.print(str);
        }
}
OUTPUT:        BOX IS HEAVY
```

## 16) getChars( ) :

- This method copies characters from *string into character array*.
- Characters copied into character array from starting position "pos1" up to last position "pos2-1" to location starting from "pos3" in a character array.

Syntax:

```
s.getChars(pos1,pos2,arr,pos3);
```

Here, "s" is an object of String class and that contains string.
"pos1" is an integer value represents starting position from which string copied.
"pos2" is an integer value represents ending position & string copied ends at pos2-1 index.
"arr" is character array that stores copied characters from string "s".
"pos3" is an integer value from which character array "arr" stores the copied character.

E.g.

```
class    stringGetchar
{
        public  static  void  main(String    args[ ])
        {
                String    s= "Box is heavy";
                char [ ]arr=new char[20];
                s.getChars(7,11,arr,0);
                for(char i:arr)
                {
                    System.out.print(i);
                }
        }
}
OUTPUT:        heav
```

**17)** trim( ) :

- This method removes unwanted i.e. extra spaces which were found *before and after* the string.
- *Note that:* This method does *not remove extra spaces between two words* of string.

Syntax:

```
String      str = s.trim( );
```

Here, "s" is an object of String class and that contains string.
"str" is also object of String class used to store resultant string.

E.g.

```
class    stringTrim
{
        public  static  void  main(String    args[ ])
        {
                String    s= "          Box          is          heavy          ";
                String    str = s.trim( );
                System.out.print(str);
        }
}

OUTPUT:
Box          is          heavy
```

**18)** split( ) :

- This method splits or cuts the given string into number of pieces (sub strings) corresponding to delimiter (specified character) and store it into array of string.

Syntax:

```
String      [ ] str = s.split(delimiter);
```

Here, "s" is an object of String class and that contains string.
"delimiter" is string at which we specify splitting character
"str" is array of String that stores splitted strings at individual indices.

E.g.

```
class    stringSplit
{
        public  static  void  main(String    args[ ])
        {
                String    s= "Box is heavy,dirty,red and useless";
                String [ ]str=s.split(",");
                for(int i=0;i<=str.length-1;i++)
                {
                        System.out.println(str[i]);
                }
        }
}
OUTPUT:
                Box is heavy
                dirty
                red and useless
```

## String Comparison:

* We know that for comparison of two or more quantities with each other, we use relational operators like <, >, <=, >=, !=, ==.
* But for string comparison, relational operators are *not* suitable because these operators *compare reference (address) of object with each other*. They ***not*** *compare strings contents*.
* For that purpose, compareTo() or equals() methods are used to compare two strings. And *these functions compare strings contents for equality*.

Consider following example;

```
class    stringCmp
{
        public  static  void  main(String    args[ ])
        {
                String    s1=new    String("Hello");
                String    s2=new    String("Hello");
                if(s1= = s2)
                        System.out.println("Strings  are  same");
                else
                        System.out.println("Strings  are  NOT  same");
        }
}

OUTPUT:        Strings are NOT same
```

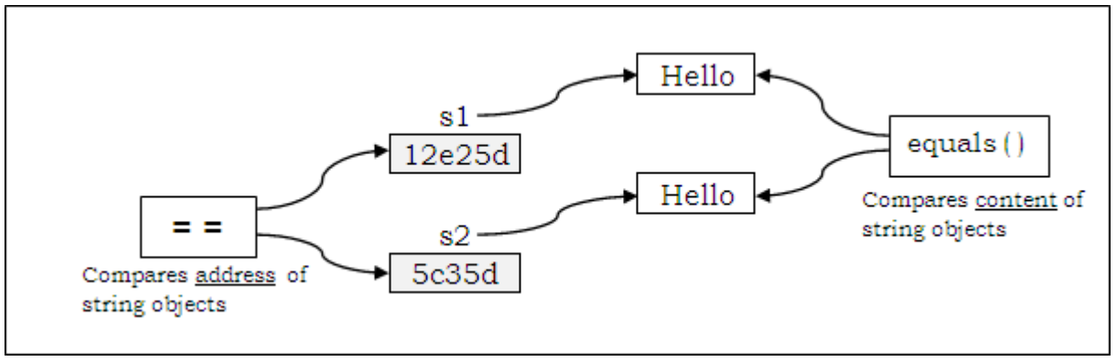
- In above example, we got output “Strings are NOT same” still content of strings are same. This is due to relational operator (= =) because it compares address (*address is in Hexadecimal number*) of both strings, it not compares strings content.
- When an object is created by JVM, it returns the memory address of the object as a hexadecimal number which is called “*object reference*” and this *object reference is separate for every object*. i.e. whenever an abject is created, a new address number is allotted to it by JVM.

Now, consider another example:

```
class    stringCmp
{
        public static void main(String    args[ ])
        {
                String    s1=new    String(“Hello”);
                String    s2=new    String(“Hello”);
                if(s1.equals(s2)==true)
                        System.out.println(“Strings are same”);
                else
                        System.out.println(“Strings are NOT same”);
        }
}

OUTPUT:       Strings are same
```

- In above example, we got output “Strings are same” which is right because *equals( )* method compares strings content, *not their addresses*.

Above mentioned concepts shown in following figures:




Now, consider One another example: There is slightly change in object creation of String. Here, object of String is created without using "new" object:

```
class    stringCmp
{
        public static void main(String    args[ ])
        {
                String    s1=“Hello”;
                String    s2=“Hello”;
                if(s1==s2)
                        System.out.println(“Strings are same”);
                else
                        System.out.println(“Strings are NOT same”);
        }
}

OUTPUT:       Strings are same
```

- In above example, we got output “Strings are same”. Still we are using relational operator (= =) for string comparison. How it is possible?

→

Here, String objects are created without using "new" operator at that time JVM uses separate memory block which is called "string constant pool" and such objects are store there.

When first statement i.e. String s1= “Hello” is executed then JVM insert object "s1" into "string constant pool" with separate address.

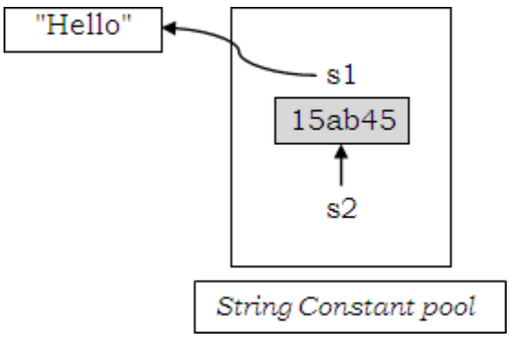
And when second statement i.e. String *s2 “Hello”* is executed then JVM, searches in "string constant pool" to know whether the object with same content is already exist there or not?

If any object with same content is found in "string constant pool" then JVM just *attach second object at reference (address) of first object*.

If any object with same content is NOT found in "s tring constant pool" then JVM *allocate separate reference (address) to second object.*

Here, content of String "s1" and "s2" are same therefore JVM just attach "s2" at address of "s1". And that"s made both objects addresses are same. And hence, we got output "Strings are same"

Above mentioned concept is shown in following fig.




## Immutability of String:

We know that, an "object" is basic runtime entity that holds data or some content. And every Objects can broadly classified into two categories viz: *1) Mutable object 2) Immutable object.* Let"s see them in details:

### 1) Mutable Object:

The objects whose content can be changeable or modifiable are called "Mutable" objects.

### 2) Immutable Object:

The objects whose content can NOT be changeable or modifiable are called "Immutable" O bjects.

And Object of String class is *Immutable* i.e. we cannot modify the content of String object. And hence, we can say that "S tring" C lass is Immutable class.

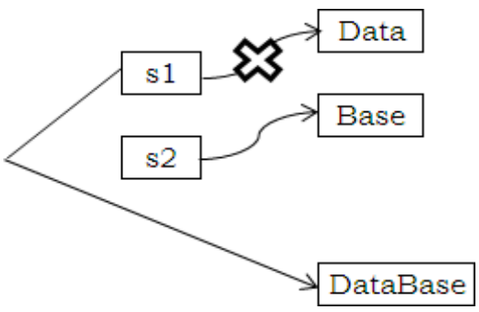
Consider following example to test immutability of String.

```
class    myString
{
        public  static  void  main(String    args[ ])
        {
                String    s1="Data";
                String    s2="Base";
                s1=s1+s2;
                System.out.println(s1);
        }
}

OUTPUT:      DataBase
```

In above example we got output "DataBase". It seems that, the content of "s1" is modified. Because, earlier "s1" has content "Data" and "s2" has content "Base". And after "s1+s2" they are joined together and total string becomes "DataBase". This result is assigned to "s1". If "s1" is mutable then it gets new string "DataBase" and we also got "DataBase" as result.

But "s1" is object of S tring class & we learned that "S tring objects are immutable", then how we got result "DataBase"?

Definitely String object "s1" is immutable. Consider following figure that will explain this concept briefly:




In the program, JVM creates two objects "s1" and "s2" separately, as shown in above figure. When "s1+s2" is done then JVM creates new object and stores "DataBase" in that created object. After creating new object, the reference "s1" is just attached or assigned to newly created object. Remember, it does not modify exis ting content of "s1". And that"s why we are called "String objects *are immutable."*

The old object that contains "Data" has lost its reference. So it is called "unreferenced object" and garbage collector remove it from memory.

# CLASSES AND OBJECTS

## *Class:*

"Class" is user defined data type which is collection of variables and methods.
That is, these variables are called "properties or attributes" and methods are called "actions"
Also, "Class" is blue print for object because everything in class can also hold by object. i.e. class decides how the object was ?

- Since, all variables and methods are also available in objects, because they are created from class therefore they are called "instance variable" and "instance method".

Consider, following defined class for "person".

```
class          person
{
        int      age;
        String   name;       //Properties- i.e. instance variables of class

       void    talk( )    // action- i.e. instance method
      {
            System.out.println("Hi, My name is "+name);
            System.out.println("My age is "+age);
       }
}
```

In above, example *person* is class that consists of instance variables (properties)*age, name*and having instance method (action)*talk( )*.

## *Object:*

"Object" is basic run time entity that holds entire data (variables and methods) of class.
O bject is also called as "instance" of class.
When object is created then that created object is stored in "Heap area" by JVM.

- Also, whenever an object is created then JVM allocates separate memory reference (address) for created object called "hash code"

This "hash code" is unique hexadecimal number created by JVM for object.

- We can also see *hash code* of every created object using *hashCode()* method of *Object* class of *java.lang* package.

Syntax to create object:

```
class_name      obj= new     class_name( );
```

Here, *class_name* is name of the class which is an identifier
*obj* is created object of class.

## *Purpose to create object:*

There are following purpose to create the object:

1) To store entire data (variables and methods) of class.
2) To access variable or to make call for methods of class from outside class.

Following program demonstrate the use of *hashCode()* method.

```
Class       Demo
{
       public    static    void    main(String [ ] args)
       {
              Demo   a=new   Demo();
              Demo   b=new   Demo();
              System.out.println("First Object's Hash Code="+a.hashCode( ));
              System.out.println("Second Object's Hash Code="+b.hashCode( ));
       }
}

OUT PUT:
       First Object's Hash Code=53ab83406
       Second Object's Hash Code=1b6164678
```

## *Types of Variable in Java:*

There are three kinds of variable found in Java language:

1) Local Variable
2) Instance Variable (Non-static Variable)
3) Class Variable (Static Variable)

Let"s see all variables in details:

## 1) Local Variable:

The variable is declared within body any method is called "Local Variable".

- The scope of Local variable is limited to that method in which it is declared i.e. it is not accessible within another method.

## 2) Instance Variable (Non-static Variable):

The variable is declared within class body without using static keyword is called "Instance Variable".

- These variables are called *instance variable* because their content is in *object* of class.
- Such, instance variable can easily be accessed by class methods of same class in which it is declared.

Instance variable"s S eparate copy is given all objects of class.

- Instance variables are stored at Heap area by JVM.

Following table shows default values allocated by Java compiler to Instance variable.

| ***Data Type*** | ***Default Value*** |
|---|---|
| byte | 0 |
| short | 0 |
| int | 0 |
| long | 0 |
| float | 0.0 |
| double | 0.0 |
| char | space |
| String | null |
| any class type (i.e. Object) | null |
| boolean | false |

## 3) Class Variable (Static Variable):

The variable is declared within class body with using s tatic keyword is called "Class Variable" or "Static variable".

- Static variables are common to all objects of class i.e. Common copy of static variable is shared among all objects of class.
- These variables are called *class variable* because they are in class scope.
- Such, class variable can easily be accessed by class methods of same class in which it is declared.
- Class variables are stored at method area by JVM.

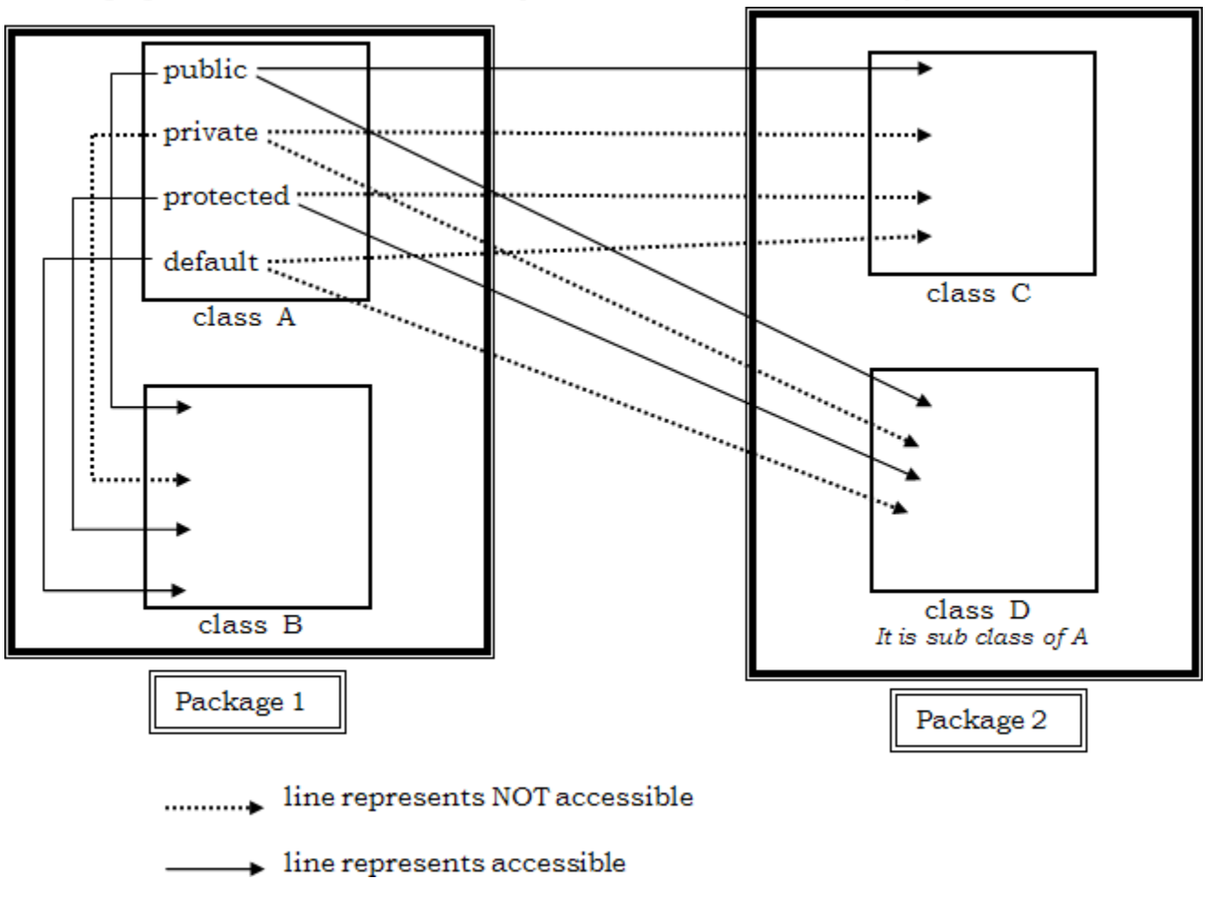
## Access Specifiers:(Member Access Controls)

- Access Specifier is a keyword that specifies, How to access the members of class or a class itself.
- We can use access Specifiers before a class and its members.

There are 4 access specifiers in Java language VIZ.

1) public 2) private 3) protected 4) default

Following figure shows different access Specifier with their accessibility:



Let us see all these access Specifier in details:

## public:

"public" members of class are accessible everywhere therefore they have **global scope**.

*They are accessible:*

1) Within methods of same class.
2) Within methods of sub classes.
3) Within methods of other class of same package.
4) Within methods of class of other package.

## private:

"private" members of class are only accessible by methods of same class therefore they have **class scope**. That is private members are not accessible outside the class.

*They are accessible:*

1) Within methods of same class.

*They are **NOT** accessible:*

1) Within methods of sub classes.
2) Within methods of other class of same package.
3) Within methods of class of other package.

## protected:

The accessibility of "protected" members of class is given as follow;

*They are accessible:*

1) Within methods of same class.
2) Within methods of sub classes.
3) Within methods of other class of same package.
4) Within methods of classes of other package (Since, other package class should be sub class)

*They are **NOT** accessible:*

1) Within methods of classes of other package (If other package class should **NOT** be sub Class)

## default:

If no access specifier is written by the programmer then, Java compiler uses "default" access specifier.

The "default" members of class should be accessible by the methods of class of same package i.e. they are not accessible out the package. Therefore *'default'* access specifier has *'package'* scope.

*They are accessible:*

1) Within methods of same class.
2) Within methods of sub classes.
3) Within methods of other class of same package.

*They are **NOT** accessible:*

1) Within methods of classes of other package.

## Methods in Java:

In Java, there are two kinds of methods found;

1) Instance Methods (Non-Static Methods) 2) Class Methods (Static Methods)

Let us see all these methods in details:

### 1) Instance Methods (Non-Static Methods):

The method which is defined within class body without using keyword "static" are called as "Instance methods or Non-static methods".

These types of methods are act on "instance variable" of class.

They are called "Instance" method because their content is in instance (object) of class.

- Instance methods are called with the help of object of class using syntax:

  object.method(arg1,arg2, --- );
- One specialty of instance method is that they are capable to access instance variable (non-static variable) and class variable (static variable) directly.

### 2) Class Methods (Static Methods):

The method which is defined within class body using keyword 'static' is called as "class methods or static methods".

These types of methods are act on "class variable (static variable)" of class.

They are called "Class" methods because they are defined using "static" keyword & it is related with class.

- Class methods are called with the help of class name using syntax:

  Class_Name.method(arg1,arg2, ---- );
- Class methods are capable to access only class variable (static variable) directly.
- Still, if we want to access instance variable inside class method then it can be accessed only by using object of class.

(**HOME WORK**: *Write Difference between Instance Methods and Class Methods)*

## Types of Instance Methods:

Further, Instance methods are of two types:

1) Accessor Method 2) Mutator Method

Let"s see these methods in details:

### 1) Accessor Method:

This type of instance methods are *only* access or read instance variables i.e. they do not modify value of instance variable are called as "Accessor Method"

### 2) Mutator Method:

This type of instance methods are access or read instance variables and *also modify* value of instance variable are called as "Mutator Method".

Following program shows the use of Accessor and Mutator methods.

```
class     Access
{
        int n;
        void  get( )   //mutator  method
        {
                n=50;
        }
        void  show( )     //Accessor  method
        {
                System.out.println("Value="+n);
        }
}
```

```
class       Call_me
{
        public  static   void   main(String   [ ] args)
        {
                Access   t=new   Access();
                t.get();
                t.show();
        }
}
```

## ➢ Static data:

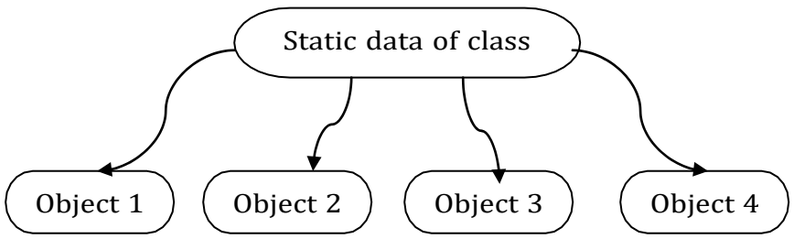
- We know that, for single class we can create multiple objects. And each object holds their individual record. But there are some data which are common to all objects in such situation that data can be made as static.
- Once the data is made as static then this data is common to all objects of class.
- In short, static data is common data and which shared by all objects of class.
- Only one copy of static data members is maintained for entire class therefore it is shared among all objects.
- Static data by default initialized to zero.
- Static data are initialized at once therefore it is used like a counter.
- Static data are stored separately rather than as a part of an object.
- Syntax to declare static variable:

```
static      datatype     variable;
```

Here,

"static" is keyword which is used to declare member as static
"datatype" is any valid data type in Java.
"variable" is name of static variable.

Following fig. shows sharing of static data by different objects of class:




In above fig. static data of class is common to all objects Object1, Object2, Object3, Object4. Or the same copy of static data is shared among all objects.

*Following program demonstrate the use of static variable that counts total object created for class:*

```
class     count
{
        static   int      cnt;
        count( )
        {
            cnt++;
        }
        static void show( )
        {
          System.out.println("Total  Objects=  "+cnt);
        }
}
```

```
class      call_count
{
    public     static    void    main(String  [ ]args)
        {
                count a=new  count( );
                count b=new  count( );
                count c=new  count( );
                count d=new  count( );
                count.show( );
        }
}
OUTPUT:        Total Objects= 4
```

## Constructor:

- The constructor is special method whose name is same as that of class name.
- The main task of constructor is to initialize values for object of its class.
- It is called constructor because it construct the values for object of the class.
- The constructors are implicitly invoked by the compiler whenever an object of class is created.
- A constructor is called concurrently when the object creation is going on. By the time, object creation is completed, the constructor execution is also completed.

Example:

Consider, there is one class having name "person" then its constructor is given as:

```
class        person
{
        person( )       // constructor
        {
                -------
                -------
        }
}
```

Properties/Characteristics/Features of Constructor:

1) Constructor name is same as that of class name.
2) Constructor does not have any return type even void.
3) Constructors are implicitly invoked by the compiler whenever object of class is created.
4) Constructor can be overloaded.
5) Constructors are not static.
6) Call to constructor is depends on number of objects created for class.

## Types of constructor:

Depending upon arguments passed to constructor, it has two types.

1) Default Constructor
2) Parameterized constructor

*NOTE:*

1) *Java language does not support for copy constructor.*
2) *If we specify return type for constructor then Java compiler treats that method as normal method.*

## 1) Default constructor:

The constructor that does not have any argument or parameter as input, such constructor is called as "Default Constructor".

Default constructor is implicitly invoked by compiler whenever object of class is crated

*Syntax:*

```
class      class_name
{
        class_name( )          // default constructor.
        {
                -------
                -------
        }
}
```

Implementation of default constructor:

```
class      person
{
   int        age;
   String     name,add;
   double     sal;
     person( )    // default constructor
   {
      age=70;
      name= "Sachin";
      add= "Pune";
      sal= 4568.50
    }
```

```
        void  show( )
        {
            System.out.println("Age="+age);
            System.out.println("Name="+name);
            System.out.println("Address="+add);
            System.out.println("Salary="+sal);
        }
}
public    static   void      main(String [ ] args )
{
        person    x=new person( );   //call to def. constructor
        x.show( );
 }
```

## 2) Parameterized constructor:

- The constructor that accepts any argument or parameter as input, such constructor is called as "parameterized Constructor".
- Parameterized constructor is implicitly invoked by compiler when we pass some values direct to object at the time of its creation.

*Syntax:*

```
class      class_name
{
        class_name( int   x, String  y)          // parameterized constructor.
        {
                -------
                -------
        }
}
```

Implementation of parameterized constructor:

```
class      person
{
   int        age;
   String    name,add;
   double    sal;
   person(int a, String  n ) //para.constructor
   {
     age=a;
     name=n;
     add= "Pune";
     sal= 4568.50
    }
```

```
void  show( )
{
   System.out.println("Age="+age);
   System.out.println("Name="+name);
   System.out.println("Address="+add);
   System.out.println("Salary="+sal);
 }
}
public    static   void      main(String [ ] args )
{
    person    x=new person(45, "Raj" ); //call to def. constructor
    x.show( );
 }
```

## Overloading of Constructor (Multiple Constructor in Class):

- We know that in case of Method overloading, we can define or implement many methods with same name within same class, but all these methods are differ from each other according to their signature (Type of argument and number of argument accept by methods).
- Like method overloading, we are also able to overload the constructor.
- Constructor overloading means implementing or defining multiple constructors for single class. But all these constructors are differing from each other according to their signatures.

Implementation of Constructor overloading:

```
class  person
{
   int        age;
   String   name,add;
   double sal;

    person( )    // default constructor
   {
     age=70;
     name= "Sachin";
     add= "Pune";
     sal= 4568.50
    }

 person(int a, String  n ) //para.constructor
 {
     age=a;
     name=n;
     add= "MUMBAI";
     sal= 54508.75;
 }
```

```
 void  show( )
 {
   System.out.println("Age="+age);
   System.out.println("Name="+name);
   System.out.println("Address="+add);
   System.out.println("Salary="+sal);
  }
}

public    static   void      main(String [ ] args )
{
    person    p=new  person( );
    person    q=new person(45, "Raj" );
    p.show( );
    q.show( );
 }
```

## Parameter passing technique in Java:

- When we pass any primitive data type variable or any object or even object-reference to method then they are passed as "Pass by value" or "Call by value" concept in Java.

That is, Default parameter passing technique in Java is "Pass by value" or "Call by value".

- Pass by address (Pass by pointer) or call by address (Call by pointer) is invalid in Java, because Java does not support for pointer.

Everything is passed to method in Java by "Pass by value" or "Call by value" concept.

## Passing Object as 'pass by value':

We know that, everything is passed to method in Java by "Pass by value" or "Call by value" concept whether it is normal variable or any objet or any object-reference.
Consider, following program that illustrate passing object as pass by value.

```
class    emp
{
        int   id;
        emp(int id)
        {
                this.id=id;
        }
}
class    swap
{
        emp   c;
        void    doswap(emp a,emp  b)
        {
                c=a;
                a=b;
                b=c;
        }
}
```

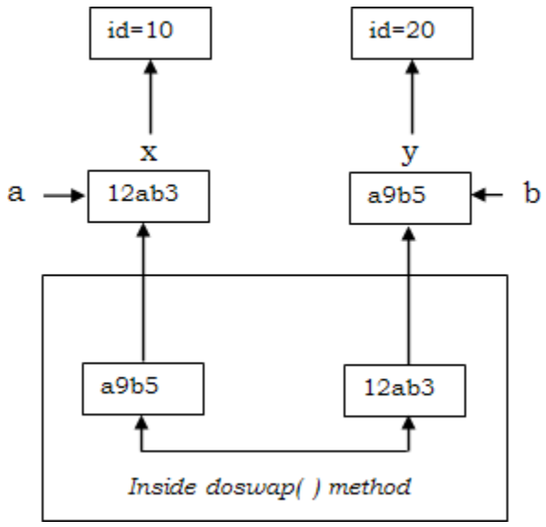
```
class      callswap
{
  public    static    void    main(String     [ ]sd)
   {
         emp   x=new   emp(10);
         emp   y=new   emp(20);
         swap p=new swap( );
    System.out.println("Before swapping ID's= "+x.id+"   "+y.id);
         p.doswap(x,y);
     System.out.println("After swapping ID's=  "+x.id+"   "+y.id);
   }
}
```

OUT PUT:
Before swapping ID's=10  20
After swapping ID's= **10  20**

In above program, we pass two objects "x', 'y' of "emp' class to dos wap( ) method. Here, objects are pass by value. These passed objects are stored in corresponding formal objects "a' and *'b'*.

- We do some swapping mechanism over objects *'a'* and *'b'* inside *doswap( )* method that does not affects on values of *'id'* hold by objects *'x'* and *'y'* because they are passed as pass by value.
- Here, we interchange the objects that ***interchange the references*** of objects. That"s why we got same output corresponding to input i.e. no interchanging of *'id'* is done.

This is shown in following figure.



## Passing Array to method:

Like normal variable, we can also pass array to the method.
In java array can be pass to method by using *only array name*.
Following program demonstrate passing array to method.

```
class    passarr
{
        void   show(int    z[ ])
        {
                System.out.print("Array Ele= ");
                for( int i : z)
                {
                    System.out.print("  "+i);
                }
        }
}
```

```
class      pass
{
        public   static  void   main(String [ ] args)
        {
                int x[ ]={10,20,30,40,50};
                passarr    p=new    passarr( );
                p.show(x);
        }
}
OUTPUT:
Array Ele= 10  20   30   40   50
```

## Returning Array from method:

Like any value, we can also return array from the method.
Following program demonstrate returning array from method.

```
class    Ret_arr
{
        int[ ]    show( )
        {
                int x[ ]={10,20,30,40,50};
                return(x);
        }
}
```

```
class     callarray
{
        public   static  void   main(String [ ] args)
        {
                int   z[ ];
                Ret_arr  p=new   Ret_arr ( );
                z=p.show( );
                System.out.print("Array Ele= ");
                for(int i : z)
                {
                        System.out.print("    "+i);
                }
        }
}
```

## Nesting of classes:

- If we define or declare one class inside another class then such a form is called as “class within class or Nested class”
- Using this facility complex data type can be created.
- Nested class is used to access elements of class which is part of another class.

## Syntax for nested class:

```
class          outerClass
{
        -----------
        -----------
        -----------
        class          innerClass
        {
                ------------
                ------------
                ------------
                class          superinnerClass
                {
                        ------------
                        ------------
                        ------------
                }
        }

}
```

## Syntax to create objects:

1) Create object of outer class as:
   *outerClass  obj1=new  outerClass( );*
2) Create object of inner class as:
   *outerClass.innerClass  obj2=obj1.new  innerClass( );*
3) Create object of supper inner class as:
   *outerClass.innerClass.superinnerClas  obj3=obj2.new  superinnerClass( );*

Like that we can create object of different inner classes also.

Following program demonstrates the concept of nested classes:

```
class    university
{
    String    uname="Solapur";
     void ushow( )
     {
        System.out.println("University=  "+uname);
     }
   class    college
   {
        String    cname="XYZ";
        void cshow( )
        {
          System.out.println("College=  "+cname);
        }
        class  dept
        {
           String dname="Computer";
           void dshow( )
           {
            System.out.println("Department=  "+dname);
           }
        }
   }
}
```

```
class    nesting
{
   public   static   void   main(String [ ] args)
   {
       university    a=new university( );
      university.college b=a.new college( );
      university.college.dept  c=b.new dept( );
         a.ushow();
         b.cshow();
         c.dshow();
   }
}
```

## Static Block:

A static block is block of statements declared with keyword "static", something like this:

```
static
{
        ------------
        ------------
        ------------
}
```

- *Static block has highest priority* therefore JVM interprets this block first even before main( ) method also.
- In, Java interpretation, JVM first searches for static block if it is found then JVM executes it.
- If JVM not found any static block then it searches for main( ) method for interpretation. If main( ) method founds then JVM execute it. If main( ) method not found then it gives error.
- Also, Single Java program may contains multiple static blocks; in such situation JVM executes them one after another in sequence (FIFO) manner.

Following program demonstrate use of static block.

```
class        Test
{
     static
     {
        System.out.println("First");
     }
    public  static   void   main(String   [ ] as)
    {
       System.out.println("Main  Method");
    }
    static
     {
        System.out.println("Third");
     }
     static
     {
        System.out.println("Second");
     }
 }
OUTPUT:
          First
          Third
          Second
          Main Method
```

*Note:*

*Java program may be compile and run without main( ) method with presence of static block.*

# The keyword 'this':

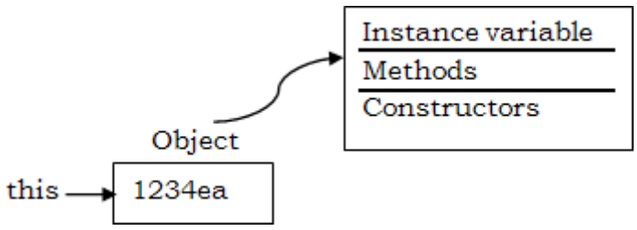
- When an object is created to a class, then *JVM implicitly create default reference to created object and that default reference is called "this".*

"this" keyword always refers to current object of class.

"this" is non-static therefore it cannot use within static method or static block.

- Generally, we write instance variable, methods and constructors in a class. All these members are stored in object. As well as *all these members are also refers by 'this' reference*.

"this" reference is shown in following figure:




## Use of "this":

We know that "this" is reference for current object of class therefore it is used like object i.e.

1) "this" is used to invoke default constructor of present class using syntax=> this( );
   Also, we made call to parameterized constructor of present class using syntax=>
   this (arg1, arg2,.....);
2) "this" is also used to call instance methods of class using syntax=> this.method(arg1,arg2, ... );
3) "this" is also used to access instance variable of class using syntax=> this.variable;
4) When instance variables of class and formal parameters of methods are same then that creates confusion. To solve this confusion "this' keyword is used to access instance variables of class.

Following program shows use of "this" keyword:

```
class      sample
{
        int         age;
        String    name;
        sample( )
        {
                this(45,"Sachin");          //call to parameterized constructor
        }
        sample( int    age, String     name)
        {
                                        /* Instance variable & formal parameters are same.
                                        Therefore 'this' is used to access instance variable*/
                this.age=age;
                this.name=name;
        }
        void    show( )
        {
                System.out.println("Age="+age);
                System.out.println("Name="+name);
        }
}
class       call_sample
{
        public    static    void  main(String     [ ]args)
        {
                sample     p=new    sample( );
                p.show( );
        }
}
OUTPUT:
                Age= 45
                Name=Sachin
```

# Inheritance

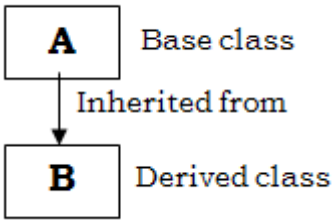
## *Introduction:*

- We know that, inheritance is one of the most important object oriented concept and Java language supports it.
- The main concept behind inheritance is reusability of code (Software) is possible by adding some extra feature into existing software without modifying it.
- In java language, '*Object'* is super class of all classes even your own defined class also.

## Definition:

- "The mechanism of creating new class from existing class is called as Inheritance".
- The existing or old class is called as 'Base class' or 'Parent class' or 'Super class' and new class is called as 'Derived class' or 'Child class' or 'Sub class'.
- While deriving the new class from exiting one then,*the derived class will have some or all the features of existing class and also it can add its own features*i.e sub class will acquires features of base class without modifying it.
- When a new class is derived from exiting class then all *public,protected and default* data of base class can easily accessed into its derived class where as private data of baseclass cannot inherited i.e. this private data cannot accessed inderived class.

Consider following example:




In above figure class B is inherited from class A. Therefore class A is called as *Super class* or or*parent class or base class* of class Bwhere asclass B is called as *Sub class or child class or derived class*.

## Need or Features of Inheritance:

1) It is always nice to use existing software instead of creating new one by adding some extra features without modifying it. This can be done by only inheritance.
2) Due to inheritance software cost is reduces.
3) Due to inheritance software developing time is also reduces.
4) Inheritance allows reusability of code.
5) Due to inheritance we can add some enhancement to the base class without modifying it this is due to creating new class from exiting one.

## Syntax to derive new class from existing class:

```
class    derived_class_name extends base_class_name
{
        //Declaration of Variables and definition of methods.
}
```

Here;

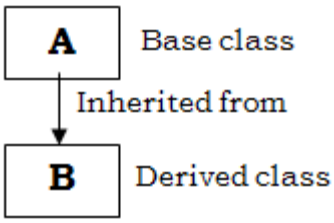
*class* is keyword to define class.
*extends* is keyword used to create or derive or extends new class.
derived_class_name is name of newly created class which is an identifier.
base_class_name is name of existing class which is also an identifier.

## Types (forms) of Inheritance:

Depending upon number of base classes and number of derived classes and their arrangement, inheritance has following types.

### 1) Single Inheritance:

In case of Single inheritance there is only *one base class from which we derive only one new class*.
e.g.



From above figure we can write single inheritance as:

```
class       A
{
  //Declaration of Variables and definition of methods.
}

class       B   extends       A
{
  //Declaration of Variables and definition of methods.
}
```

Implementation of Single Inheritance is as fallow:

```
importjava.util.Scanner;
class     A
{
          Int x,y;
           Scanner   sc=new   Scanner(System.in);
          void get( )
          {
                    System.out.println("Enter two numbers ");
                     x=sc.nextInt( );
                     y=sc.nextInt( );
          }
}
class      add    extends      A
{
          int z;
          void      doadd( )
          {
                    z=x+y;
                    System.out.println("Addition="+z);
          }
}

classsingle
{
          publicstatic void main(String [ ] args)
          {
                    add   p=new  add( );
                    p.get();
                    p.doadd( );
          }
}
```

# ‘super’ keyword:

- If we create an object of super class then we can access only super class members, i.e. we are not able to access the sub class members using object of super class.
- But, *if we create an object of sub class, then all the members of super class as well as sub class are easily accessible by object of sub class*. And that’s why we always create an object of sub class instead of super class.
- Some time, both super class and sub class may have members (variable or methods) with same name in such situation, *only members of sub class is accessible with its object*. Here, super class members are not considered. And hence, in such condition if we want to access super class members into its sub class then ‘super’ keyword is used along with member of base class.

## Use of 'super' keyword:

- 'super' keyword is used to access members of base class into its derived class.
- We can access, instance variables of base class into its derived class using syntax:

```
super.variable;
```

- We can access, methods of base class into its derived class using syntax:

```
super.method(arg1,arg2, ...);
```

- Note that, *we need not call the default constructor of super class because it is defaultly available to its sub class*. And hence, We can made call to parameterized constructor of base class into its derived class using syntax:

```
super( arg1,arg2,.....);
```

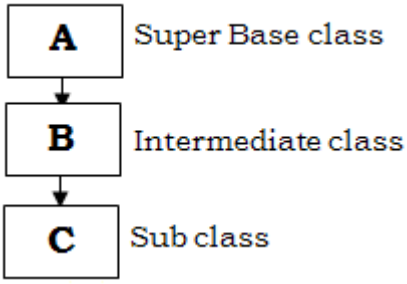
Consider following example that demonstrate use of 'super' keyword:

```
classA
{
        int   x=55;

}
class   B   extends   A
{
        int    x=70;
        void   get( )
        {
        System.out.println("Super Class variable="+super.x);   //access 'x' of base class using 'super' keyword
        System.out.println("Sub Class variable="+x);
        }
}

class  use
{
        public  static  void   main(String [ ]  args)
        {
                B  p=new  B( );
                p.get( );
        }
}

OUTPUT:
                Super Class variable= 55
              Sub Class variable= 70
```

## 2) Multilevel Inheritance:

- In case of multilevel inheritance we can derive *new class from another derived classes, further from derived classes we again derive new class and so on*.
- Also, there are some levels of inheritances therefore it is called as "Multilevel inheritance"

E.g.

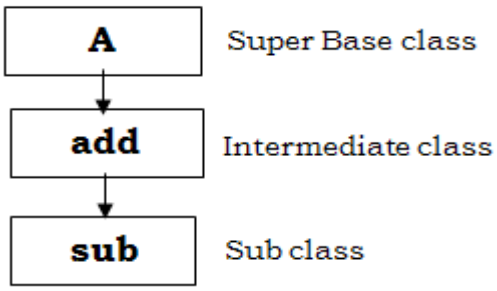


From above figure we can write multilevel inheritance as fallow:

```
class      A
{
  //Declaration of Variables and definition of methods.
}
class      B   extends   A
{
  //Declaration of Variables and definition of methods.
}
class      C    extends   B
{
  //Declaration of Variables and definition of methods.
}
```

Implementation of Multilevel Inheritance is as fallow:

Figure:



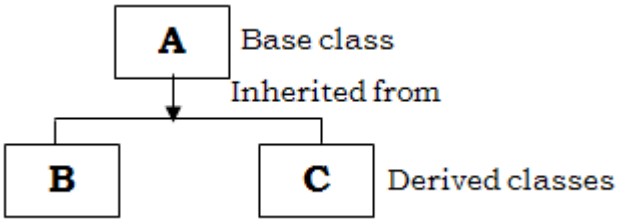

**(Program Implementation HOME WORK)**

## 3) Hierarchical Inheritance:

- In case of hierarchical inheritance there is *only one base class from that class we can derive multiple new classes*.
- The base class provides all its features to all derived classes which are common to all sub classes.
- The hierarchical inheritance comes into picture when certain feature of one level is shared by many other derived classes.

e.g.

From

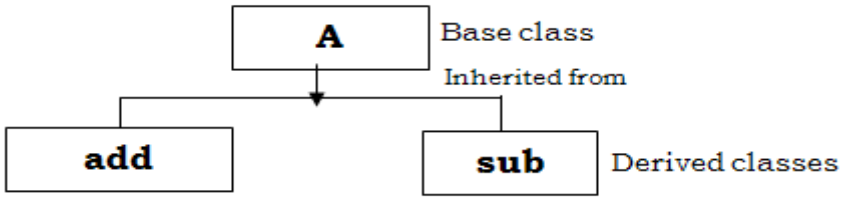



above figure we can write hierarchical inheritance as fallow:

```
class      A
{
  //Declaration of Variables and definition of methods.
}
class      B   extends   A
{
  //Declaration of Variables and definition of methods.
}
class      C    extends   A
{
  //Declaration of Variables and definition of methods.
}
```

Implementation of Hierarchical Inheritance is as fallow:

Figure:

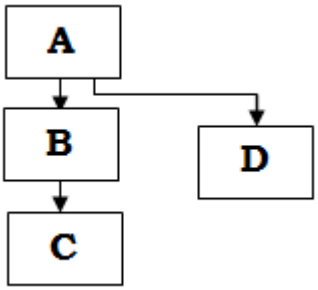



**(Program Implementation HOME WORK)**

## 4) Hybrid Inheritance:

- Hybrid inheritance is the *combination of two or more forms of inheritances*.
- To implement hybrid inheritance, we have to combine two or more forms of inheritances and such a combination of different forms of inheritances is called "Hybrid inheritance".

E.g.

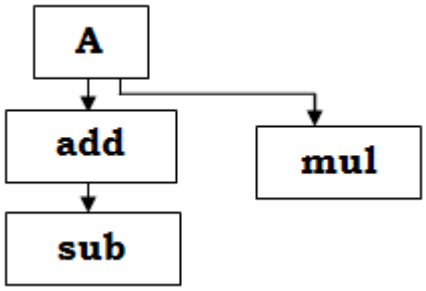



*The above inheritance is a combination of multilevel and hierarchical inheritances therefore it is "Hybrid Inheritance".*

From above figure we can write hybrid inheritance as fallow:

```
class      A
{
  //Declaration of Variables and definition of methods.
}
class      B   extends   A
{
  //Declaration of Variables and definition of methods.
}
class      C    extends   B
{
  //Declaration of Variables and definition of methods.
}
class      D   extends   A
{
  //Declaration of Variables and definition of methods.
}
```

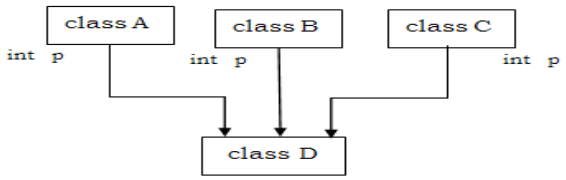
Implementation of Hybrid Inheritance is as fallow:




**(Program Implementation HOME WORK)**

## 5) MultipleInheritance:

- Multiple inheritance means deriving or creating *only one sub class from multiple super classes*.
- But, Java does not supportfor multiple inheritance. And that multiple inheritance leads confusion for programmer, which is shown in following figure:




- In above fig, Super classes i.e.*Class A, Class B* and *Class C* has same member 'int p' and from these classes *Class D* is derived or extended. In this case, *Class D* has confusion regarding with copy of'int p'. That is *Class D*does not know whose copy of 'int p' is accessed.
- Also, *class D extends A, B, C*such syntax is invalid in Java.
- If Java does not support for 'Multiple inheritance' then it is not considered as pure OOP language. In fact Java is pure OOP language.
- But, In Java, *multiple inheritance can be implemented using 'interface'* concept. We can implement multiple inheritance latter in chapter 'Interface'.

## Constructor in Inheritance:

- We know that, constructor is used to initialize the object.
- Without initializing base class members, the derived class members cannot initializes therefore compiler automatically (implicitly) *invoke base class constructor first and then the derived class constructor is invoked*.
- That is order of execution of constructor in inheritance is **base to derive**.
- Every constructor in derived class first implicitly make call to default constructor of base class.
- When a class is derived from base class then *derived class has implicitly copy of default constructor of base class*.

Following program shows invocation of constructor in Inheritance:

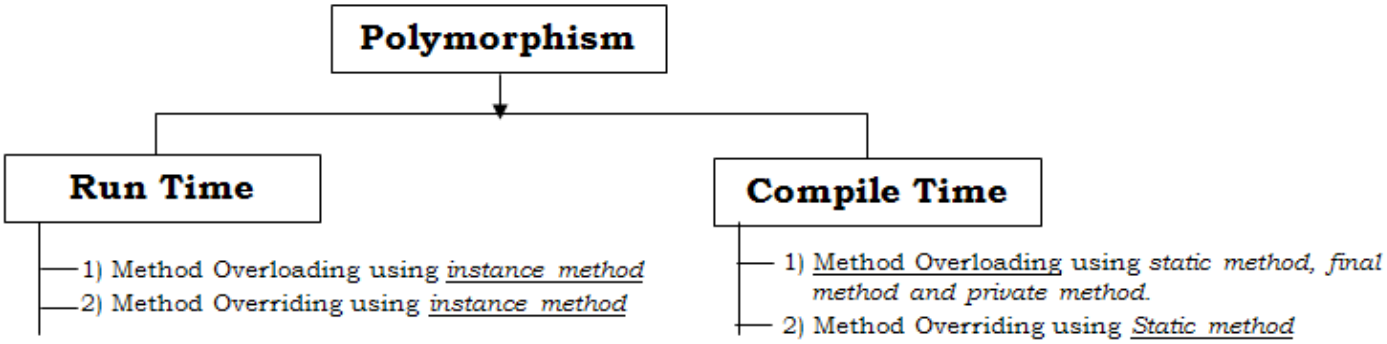
```
class    base
{
        base( )
        {
                System.out.println("Base class default Constructor ");
        }
        base(int   x)
        {
                System.out.println("Base class Parameterized constructor="+x);
        }
}
class   derive    extends       base
{
        derive( )
        {
                System.out.println("Derived class default Constructor ");
        }
        derive(int   p)
        {
                super(p);       //call to base class parameterized constructor
                System.out.println("Derived class Parameterized constructor="+p);
        }
}
class   myConstructor
{
        public  static  void    main(String   args[ ])
        {
                derive    p=new         derive( );
                derive    q=new         derive(75);
        }
}
```

# POLYMORPHISM

## *Introduction:*

- Polymorphism is Greek word that consists of two words viz. 'poly' & 'morphism'
- 'poly' means 'many' and 'morphism' means 'forms'.
- *The ability to define or write multiple methods with same name that performs different task such concept is called 'Polymorphism' .*
- Advantage of polymorphism is that, the programmer can write flexible code depending upon signature (type of argument & number of argument accept by method) of method.

Polymorphism can be classified into two categories that is shown in following figure:




Let's see all these types in briefly:

## 1) Dynamic Polymorphism (Runtime Polymorphism):

The polymorphism is happen at runtime of program is called 'Runtime or Dynamic polymorphism' or 'Dynamic Binding'

**OR**

Choosing or selecting the methods at runtime of program is called 'Runtime or Dynamic Polymorphism' or 'Dynamic Binding'

- In this case, *Java compiler* does **not** know which method is called or selected at compile time but only *JVM* knows which method is called or selected at runtime. Hence, such type of polymorphism is called 'Dynamic or Runtime polymorphism' or 'Dynamic Binding'

E.g.

1) Method overloading using instance methods.
2) Method overriding using instance methods.

## 2) Static Polymorphism (Compile time Polymorphism):

The polymorphism is happen at compile time of program is called 'Compile time or static polymorphism' or 'Static Binding'

**OR**

Choosing or selecting the methods at compile time of program is called 'Compile time or static polymorphism' or 'Static Binding'

- In this case, *Java compiler* knows which method is called or selected at compile timeandOf course, *JVM* executes that selectedmethod later.
- Here, compiler knows method code at compile time. Hence, such type of polymorphism is called 'Static or Compile time polymorphism' or 'Static Binding'

E.g.

1) Method overloading using *static methods, final methods and private methods.*
2) Method overriding using static methods.

## Method Overloading:

Writing or *defining multiple methods with same name within same class* but *all defined methods are differ from each other according to their signatures*(type of argument & number of argument accept by method) such concept is called as 'Method overloading'

Consider following example which shows Method overloading concept:

```
class     myClass
{
          void   get( )
          {
                    ----------
                    ----------
                    ----------
          }
          void  get(int    x)
          {
                    ----------
                    ----------
                    ----------
          }
          int  get(char    y, int    z)
          {
                    ----------
                    ----------
                    ----------
          }
  }
```

In above example, get( ) method is defined three times within same class 'myClass' but all these defined methods are differ from each other according to their signatures& hence, here method overloading is done.

## Method Overriding:

- When both classes i.e. super class & sub class have same methods with same signatures i.e. both super class and sub class have same method prototype then sub class method overrides (replaces) the super class method such a concept is called as 'Method overriding'
- We know that, super class methods are also executed by object of sub class. But, in method overriding *both classes has same methods with same signatures in such situation;JVM only executes methods of sub class i.e. here sub class method replaces (overrides) the super class method and hence super class method not executes*.
- That is we can say that, super class method is overridden by sub class method. And hence super class method not executed by sub class object.

Consider following example which shows Method overriding concept:

```
        class       first
        {
                    void   get(int    p)
                    {
                              ----------
                              ----------
                              ----------
                    }
      }
        class       second   extends   first
        {
                    void  get(int    x)
                    {
                              ----------
                              ----------
                              ----------
                    }
        }
```

In above example, 'first' is super class whereas 'second' is sub class of 'first' class.

Also, both classes has *get( )* method with same signatures. And if we made call to *get()* method *using object of sub class* then only sub class *get( )* method is invoked i.e. sub class method overrides the base class method &*get()* method of 'first' class never executed. Hence, here method overriding is done.

## Difference between method overloading and method overriding:

| Method Overloading | Method Overriding |
|---|---|
| 1) Defining multiple method with same name with different signature is called 'Method overloading' | 1) Defining multiple method with same name with same signature is called 'Method overriding' |
| 2) Method overloading is done within same class. | 2) Method overriding is done within different classes i.e. in super class & sub class. |
| 3) In method overloading, return type of methods can be same or different. | 3) In method overriding, return type of methods must be same. |
| 4) JVM decides which method has to be called depending upon parameters of methods. | 4) JVM decides which method has to be called depending upon object of sub class or super class. |
| 5) Method overloading is code development. Same method is developed to perform different task. | 5) Method overriding is code replacement. The sub class method overrides (replaces) the super class method. |

## Dynamic Polymorphism (Runtime Polymorphism) using *Instance method*:

Dynamic polymorphism is achieved by *method overloading and method overriding by using Instance method*.
Let's seeall these concepts in details:

### 1) Method Overloading Using *Instance methods*:

- We know that method overloading is nothing but writing or *defining multiple methods with same name within same class* but *all defined methods are differ from each other according to their signatures*(type of argument & number of argument accept by method).

Here,

- We can overload methods using instance method. And to call such instance methods,object of class is required.
- Therefore, Java compiler has to wait till object is created and object creation is takes place at runtime of programtherefore JVM only knows which methods has to execute at runtime. Here Java compiler does not know which method has to be selected at compile time.
- Hence, overloading methods using instance method is type of 'Runtime or Dynamic polymorphism'

Following program demonstrate the Method overloading concept using instance method:

```
class   myClass
{
int     p= 45,q= 55;
void    add( )
        {
        System.out.println("First="+(p+q));
        }
void   add(int    x)
        {
          System.out.println("Second="+(p+x));
        }
void   add(int    a, int    b)
 {
        System.out.println("Third="+(a+b));
}
 }
```

```
class     callmyClass
{
    public   static   void   main( String  [ ]ar)
   {
myClass     t=new    myClass( );
t.add( );
t.add(5);
t.add(2,3);
 }
}

OUTPUT:
          First= 100
          Second= 50
          Third= 5
```

### 2) Method Overriding Using *Instance methods*:

We know that, method overriding is nothing but, when both classes i.e. super class & sub class have same methods with same signatures i.e. both super class and sub class have same method prototype then sub class method overrides (replaces) the super class method such a concept is called as 'Method overriding'
Following program demonstrate the Method overriding concept using instance method:

```
class   one
{
void   add( )
        {
        System.out.println("I am in Base class");
        }
}
class two   extends   one
{
   void  add()
    {
        System.out.println("I am in Derived Class");
}
 }
```

```
class     all
{
    public   static   void   main( String  [ ]ar)
   {
two     t=new    two( );
t.add( );
 }
}

OUTPUT:
            I am in Derived Class
```

## Static Polymorphism (Compile time Polymorphism):

Compiletime polymorphism is achieved by:

1) Method overloading using *static methods, private methods and final methods.*
2) Method overriding using static methods.

Let's see all these concepts in details:

## 1) Method Overloading usingStatic *methods*:

- We know that method overloading is nothing but writing or *defining multiple methods with same name within same class* but *all defined methods are differ from each other according to their signatures*(type of argument & number of argument accept by method).

Here,

- We can overload methods using static methods. And **to call** such static methods, **object is not required**
- **We made call to static method using *class name along with method name*.**Therefore, Java compiler selects such method at compiling time of program.
- Here Java compiler knows which method has to be selected at compile time.
- Hence, overloading methods using static methods is type of 'Compile time or Static polymorphism'

Following program demonstrate the Method overloading concept using static method:

```
class   myClass
{
static  int     p= 45,q= 55;
static void   add( )
        {
        System.out.println("First="+(p+q));
        }
static  void   add(int    x)
        {
          System.out.println("Second="+(p+x));
        }
static void   add(int   a, int    b)
 {
        System.out.println("Third="+(a+b));
}
 }
```

```
class     callmyClass
{
    public   static   void   main( String  [ ]ar)
   {
//here, call made using class name.
myClass.add( );
myClass.add(5);
myClass.add(2,3);
 }
}

OUTPUT:
          First= 100
          Second= 50
          Third= 5
```

## 2) Method Overloading using private *methods*:

- Private methods are those methods that are defined using 'private' access specifier.
- This specifier makes the method not to available outside the class body.
- Private methods of base class cannot accessible into its derived class therefore method overriding using private methods are not possible. That is private methods are not overridden.
- *Remember, we may define same private methods into both base class and derived class but it is not considered as method overriding because derived class does not know the base class private methodand therefore derived class method not overrides base class private method.*
- But, we can overload private methods. Now, question is that how to call these overloaded private methods? Since, they are not accessible outside the class.
- **We made call to all overloaded private methodsfrom the definition of another public or default or protected methods.**

- Here, calls to private methods are made without using object from the definition of other non-private methods of same class. Therefore Java compiler knows which method has to be selected at compile time.
- Hence, overloading methods using private methods is type of 'Compile time or Static polymorphism'

Following program demonstrate the Method overloading concept using private methods:

```
class   myClass
{
int     p= 45,q= 55;
private void   add( )
        {
        System.out.println("First="+(p+q));
        }
private void   add(int    x)
        {
          System.out.println("Second="+(p+x));
        }
void   madecall()
 {
//here, call is made without object of class.
          add( );
         add(5);
}
 }
```

```
class     call
{
    public   static   void   main( String  [ ]ar)
   {

myClass    t=new   myClass( );
t.madecall();
 }
}

OUTPUT:
          First= 100
          Second= 50
```

## Note:

1) Private methods of base class cannot accessible into its derived class therefore method overriding using private methods are not possible. That is private methods are not overridden.
2) *Remember, we may define same private methods into both base class and derived class but it is not considered as method overriding because derived class does not know the base class private method and therefore derived class method not overrides base class private method.*

## *'final' Keyword:*

We can use 'final' keyword *before variable declaration or before method definition or before class definition.*We explain all these as follows:

## 1) *final* Variable:

- If we use 'final' keyword before variable declaration then that variable becomes 'final' variable.
- And final variables are treated in Java as constant.
- That is, to declare constant in Java, 'final' keyword is used.
- *final variable* cannot be modified because they treated as constant.

Syntax:

```
final datatype      name=value;
```

here,

'final' is keyword.
'datatype' is valid data type in java.
'name' is an identifier which is constant name .
'value' is any constant value assigned to final variable.

Ex:

```
final       double          PI= 3.14;
```

here, Incrementation or Decrementation or initialization other value to *'PI'* is invalid because it is 'final' and not modified.

## 2) *final* Method:

- If we use 'final' keyword before method definition then that method becomes 'final' method.
- And *final methods*cannot override by its derived or sub class. That is sub class method *cannot replace* code of its base class *final method*.
- *Note that: 'final' methods of base class **can be** accessible into its derived class.*

Syntax:

```
fina lreturn_typeMethod_name( )
{
          ----------
          ----------
          ----------
}
```

**Note:**
*Final methods of base class cannot override into its derived class therefore method overriding using final methods are not possible. That is final methods are also not overridden.*
*But, 'final' methods of base class can be accessible into its derived class.*

## 3) *final*Class:

- If we use 'final' keyword before class definition then that class becomes 'final' class.
- And *final class*prevents inheritance. That is, *we are not able to create or extends new class from 'final class'*
- If we do not allow sub class to access its base class features then base class has to made as 'final'

Syntax:

```
final    class    class_name
{
         ----------
         ----------
         ----------
}
```

Ex.

```
final    class    A
{
         --------
         --------
}
class    B  extends  A        //Is invalid.....
```

## 3) Method Overloading using *finalmethods*:

- *final* methods are those methods that are defined using 'final' keyword.
- This *final*keyword makes the method not to available into its sub class.
- We know that, final methods of base class cannot override into its derived class therefore method overriding using final methods are not possible.
- **We made call to all overloaded *final* methods from the definition of another public or default or protected methods.**
- Here, calls to final methods are made without using object from the definition of other non-private methods of same class. Therefore Java compiler knows which method has to be selected at compile time.
- Hence, overloading methods using final methods is type of 'Compile time or Static polymorphism'

Following program demonstrate the Method overloading concept usingfinal methods:

```
class Over_final
{
int    p= 45,q= 55;
     final   void   add( )
{
          System.out.println("First="+(p+q));
}
     final  void   add(int    x)
{
          System.out.println("Second="+(p+x));
}
void madecall()
  {
                         //call without object
     add( );
          add(25);
}
 }
```

```
class     call_Over_final
{
    public   static   void   main( String  [ ]ar)
   {

Over_final    t=new   Over_final( );
t.madecall();
    }
}

OUTPUT:
              First=100
              Second=70
```

## 4) Method Overriding Using *static methods*:

We know that, method overriding is nothing but, when both classes i.e. super class & sub class have same methods with same signatures i.e. both super class and sub class have same method prototype then sub class method overrides (replaces) the super class method such a concept is called as 'Method overriding'
Following program demonstrate the Method overriding concept using static method:

```
class      one
{
   static void    add( )
{
         System.out.println("I am in Base class");
}
}
class  two    extends    one
{
    static void   add( )
     {
         System.out.println("I am in Derived Class");
     }
}
```

```
class     all
{
     public    static    void    main( String  [ ]ar)
    {
two.add();   //call without object
     }
}

OUTPUT:
                        I am in Derived Class
```

# Abstract Method and Abstract class

## Abstract Method:

- Abstract method is such method that does not have any definition i.e. it has no body.
- Abstract method contains only method header i.e. method prototype.
- Abstract method has no body therefore they are also called 'incomplete method'
- Abstract method should be declared with keyword '*abstract*'
- When abstract method is declared into super class then *it is compulsory to define it into its sub classes*.
- If any abstract method not implemented into particular sub class , then that sub class should be declared as 'abstract'
- An abstract method is written when *same method has to perform different tasks depending upon object requirements*.

Ex:

```
abstract   class    ABC
{
       abstract  void    get( );
}
```

In above example, get( ) method is declared as abstract using keyword 'abstract'. And that method does not have any definition.

## Abstract class:

- Abstract class is such class that *contains zero or more abstract method*.
- Abstract classmay also containinstance variable & concrete methods.
- Abstract class should be declared with keyword '*abstract*'
- When abstract method is declared into super class then *it is compulsory to define it into its sub classes*.
- We *cannot create object of Abstract class* i.e. Abstract class cannot instantiated. Because abstract class contains incomplete abstract methods.
- But *we can create reference of Abstract class*. And that *reference* may refer to *object of its sub classes*.
- Abstract class may have constructors. Then, question is that who invoke constructor of abstract class?Since, object of abstract class cannot be created.

Answer is that:constructors of abstract class isinvoked by object of its sub classes.

- We cannot declare a class as *final* and *abstract* at a time. Because, *final*keyword prevents to create sub classes and *abstract* keyword allows super class to have sub classes.

Ex:

```
abstract   class    ABC
{
       abstract  void    get( );
       void    show( )
       {
            ----------
            ----------
        }
}
```

In above example, class *ABC* is declared as abstract using keyword 'abstract'. And that contains get( ) abstract methodand show( ) concrete method.
Following program demonstrate the use of abstract class and abstract methods.

```
abstract class Myabs
{
        abstract void calculate(double x);
}
class  sub1  extends Myabs
{
        void calculate(double  a)
        {
        System.out.println("Square="+(a*a));
        }
}
class  sub2  extends  Myabs
{
void calculate(double  a)
{
System.out.println("SquareRoot="+Math.sqrt(a));
}
}
```

```
class  sub3  extendsMyabs
{
        void calculate(double  a)
        {
        System.out.println("Cube="+(a*a*a));
        }
}
class call_abstract
{
        public static void main(String as[])
        {
                sub1  t1=new sub1();
                sub2  t2=new sub2();
                sub3  t3=new sub3();
                t1.calculate(5);
                t2.calculate(36);
                t3.calculate(4);
        }
}
```

# Interface

## Introduction:

We know that, Java language does not supports for multiple inheritance but we can implement multiple inheritance using 'interface'.

## Interface:

- An 'interface' is collection of only abstract methods.
- All the methods of interface by default *public & abstract*.
- Also, 'interface' has variables but by default all interface variables are *static, final and public*.
- An interface contains only abstract methods which are all incomplete therefore it is not possible to create object of interface.
- But, we can create separate classes from interface where we can implement all the methods of interface. These classes are called as 'implementation' classes.
- Since, 'implementation classes' contains method definition therefore we can create object of implementation classes.
- Every implementation class can have its own implementation of abstract methods of the interface.

Syntax to define interface:

```
interface Interface_name
{
        datatype     var1=value;
                        ⋮
                        ▼
        return_type Method1( );
                        ⋮
                        ▼
}
```

here,

'interface' is keyword used to define interface.
'Interface_name' is name of interface which is an identifier.

Syntax to define new class (implementation class) from interface:

```
class implement_class_name    implements  Interface_name
{
      datatype    var1;
            ⋮
      public  return_type    Method1( )
      {
            --------
            --------
      }
            ⋮
}
```

here,

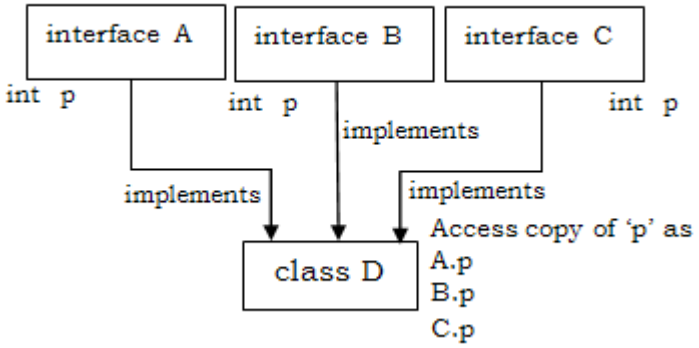
'implements' is keyword used to define or create new class from interface.
'Interface_name' is name of interface from which *implement_class_name* is created.

## Characteristic or properties of Interface:

1) An 'interface' is a *specification of method prototype* only i.e. methods does not have any body
2) An interface will have zero or more abstract methods which are all public & abstract by default.
3) An interfacecan have variables which are public, static and final by default. This means all variables of interface are constant therefore we have to assign them value.
4) None of the methods in interface can be private, protected or static.
5) We cannot create an object of interface but we can create reference to interface.
6) All methods of interface should be defined in its implementation classes. If any method is not implemented then that implementation class should be declared as 'abstract'
7) An interface can extend another interface.
8) An interface cannot implement another interface.
9) A class can implement (not extends) multiple interfaces.
   For Ex: class MyClass implements interface1,interface2,interface3 . . . .

## Multiple Inheritance using Interfaces:

- Multiple inheritance means deriving or creating *only one sub class from multiple super classes*.
- But, Java does not support for multiple inheritance. And that multiple inheritance leads confusion for programmer, which is shown in following figure:




- In above fig, *class D* is implementation class which is implemented from three interfaces viz.*interface A, interface B* and *interface C*. In this case, *Class D*can access individual copy of*'int p'*from all interfaces using*A.p, B.p and C.p*. Because, by default all variables of interface are static therefore they are accessed by class name. Thus, confusion regarding copy of *'intp'* in*Class D* is solved.
- Also, *class DimplementsA, B, C*such syntax is valid in Java.
- Thus, In Java,*multiple inheritanceis implemented using 'interface'* concept.

Following program shows multiple inheritance using interface.

```
interface A
{
        intp=10;
void calculate();
}
interface B
{
        intp=20;
void calculate();
}
interface C
{
        intp=30;
void calculate();
}
```

```
class D implementsA,B,C
{
public void calculate()
{
int z;
z=A.p+B.p+C.p;
System.out.println("Addition="+z);
}
}
class multiple
{
public static void main(String s[])
{
D   obj=new  D();
obj.calculate();
}
}
```

In above program:

*class D* is implementation class which is implemented from interfaces A,B and C.

The *calculate( )*method in implementation *'class D'* must be defined with **'public'** modifier because it overrides its abstract methods of interfaces A,B,C. And by default all interface methods are **public**.

If do not define it with public modifier then compiler gives compilation error.

## Difference between Abstract class and Interface:

| Abstract Class | Interface |
|---|---|
| 1)An abstract class contains some abstract methods and also some concrete methods | 1) An interface only contains abstract methods |
| 2) An abstract class contains instance variable | 2) An interface cannot contain instance variables. It contains only constants. |
| 3) By default abstract methods of abstract class is not public. | 3) By default abstract methods of interface is public. |
| 4) All abstract methods of abstract class should be implemented into its sub classes. | 4) All abstract methods of interface should be implemented into its implementation classes. |
| 5) Abstract class is declared using 'abstract' keyword. | 5) Interface is declared using 'interface' keyword. |
| 6) Methods in abstract class can be private, protected or static. | 6) Methods in interface cannot be private, protected or static. |
| 7) Abstract class may contains 'constructors' | 7) Interface does not have 'constructors' |

**Wrapper Classes**

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 |

Sometimes you must use wrapper classes, for example when working with Collection objects, such as ArrayList, where primitive types cannot be used (the list can only store objects):

ArrayList<Integer> myNumbers = new ArrayList<Integer>();

**Creating Wrapper Objects**
To create a wrapper object, use the wrapper class instead of the primitive type. To get the value, you can just print the object:

```
class Main
{
  public static void main(String[] args)
 {
    Integer myInt = 5;
    Double myDouble = 5.99;
    Character myChar = 'A';
    System.out.println(myInt);
    System.out.println(myDouble);
    System.out.println(myChar);
  }
}
```

Since you're now working with objects, you can use certain methods to get information about the specific object.

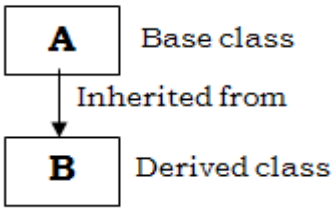
For example, the following methods are used to get the value associated with the corresponding wrapper object: intValue(), byteValue(), shortValue(), longValue(), floatValue(), doubleValue(), charValue(), booleanValue().

This example will output the same result as the example above:

```
class Main
{
  public static void main(String[] args)
 {
    Integer myInt = 5;
    Double myDouble = 5.99;
    Character myChar = 'A';
    System.out.println(myInt.intValue());
    System.out.println(myDouble.doubleValue());
    System.out.println(myChar.charValue());
  }
}
```

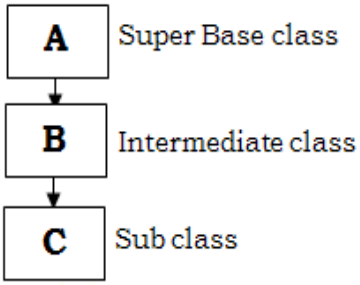
# Java Inheritance Programs

1) Single Inheritance




```
 class A
{
        int x=10;
}
class B extends A //Sigle Inheriance
{
        int y=5;
        void show()
        {
    int z=x+y;
    System.out.println("addition is="+z);
        }
}
class demo
{
        public static void main(String args[])
        {
          B m=new B();
          m.show();
        }
}
```

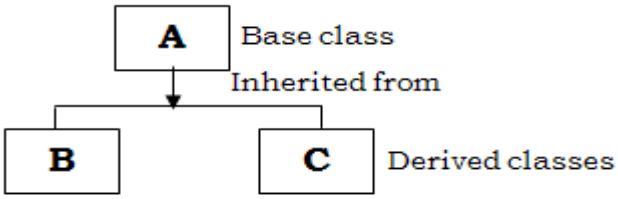
2) Multilevel Inheritance



```
class A
{
        int x=10;
}
class B extends A
{
        int y=5;
}
class C extends B
{
        int z;
        void show()
        {
    z=x+y;
   System.out.println("addition is="+z);
        }
}
```

```
class demo
{
        public static void main(String args[])
        {
          C n=new C();
          n.show();
        }
}
```

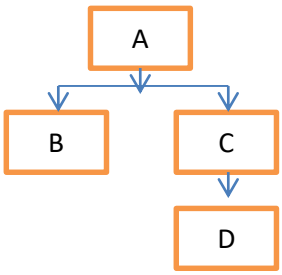
3) Hierarchical Inheritance




```
class A
{
        int x=10;
        int y=5;
}
class B extends A
{
        void add()
        {
    int p=x+y;
    System.out.println("addition is="+p);
        }
}
class C extends A
{
        int z=5;
        void sub()
        {
    int q=x-y;
    System.out.println("subtraction is="+q);
        }
}
```

```
class demo
{
        public static void main(String args[])
        {
          B m=new B();
          m.add();

          C n=new C();
          n.sub();
        }
}
```

4) Hybrid Inheritance




```
class A
{
        int x=10;
        int y=5;
}
class B extends A
{
        void sub()
        {
   int p=x-y;
   System.out.println("subtraction is="+p);
        }
}
class C extends A
{
        int z=5;
}
class D extends C
{
        void add()
        {
   int q=x+y+z;
   System.out.println("addition is="+q);
        }
}
```

```
class demo
{
        class demo
{
        public static void main(String args[])
        {
          B m=new B();
          m.sub();

          D n=new D();
          n.add();
        }
}
```