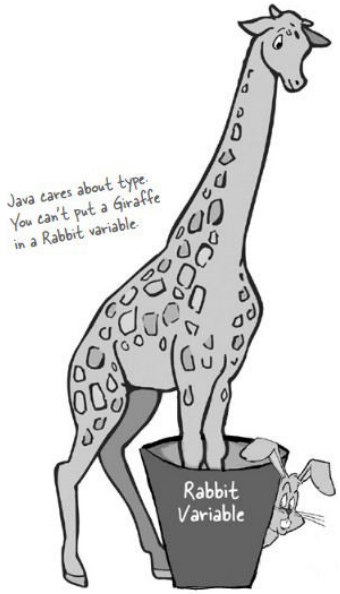
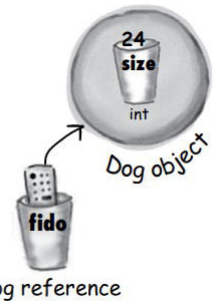
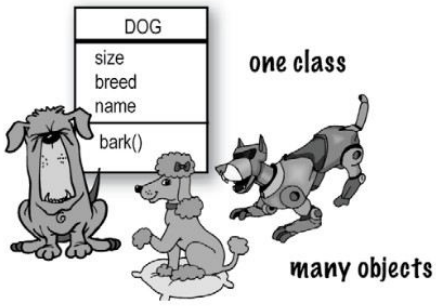
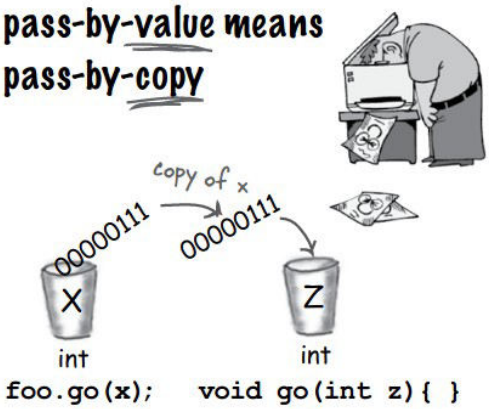
OBJECT ORIENTED PROGRAMMING through JAVA

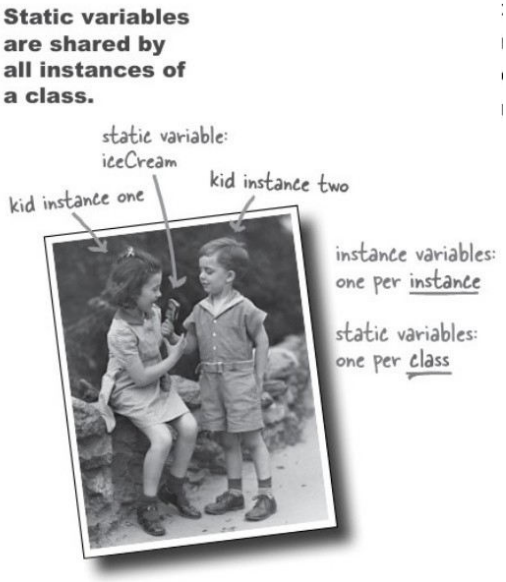
INDEX











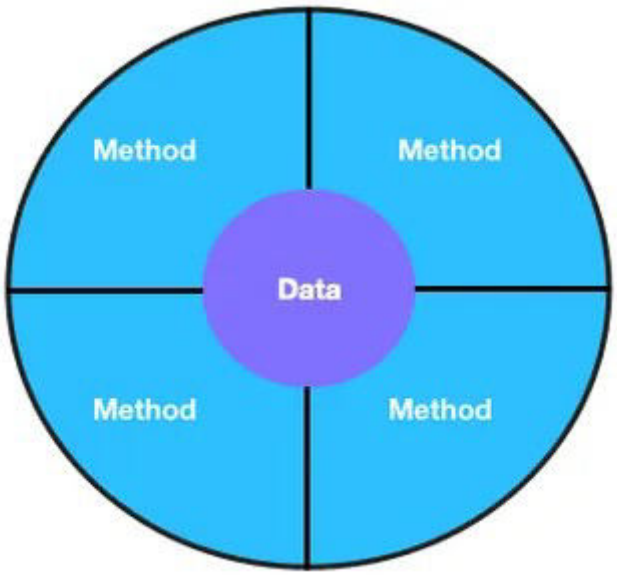
1.FUNDAMENTALS OF OBJECT ORIENTED PROGRAMMING

Object oriented programming:

Object oriented programming, as a concept, was introduced by **xerox corporation** in the early 1970s. The first object oriented language was **small talk**. Some languages are successfully implemented object oriented programming are **C++**, **Java** and **Eiffel**.

1.1 Object Oriented Paradigm :

The major objective of object oriented approach is to eliminate some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it and protects it from unintentional modification by other functions. OOP allow us to decompose a problem into a number of entities called **Objects**. Then build data and functions (known as methods in Java) around these entities. The combination of **data** and **methods** make up an **object**.



Object=Method+Data

The data of an object can be accessed only by the methods associated with that object. However, methods of one object can access the methods of other objects. Some of the features of object-oriented paradigm are -

- Emphasis is on data rather than procedure.
- Programs are divided in to what are known as Object.
- Data structures are designed such that they characterize the objects.

- Methods that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through methods.
- New data and methods can be easily added whenever necessary.
- Follows *bottom-up* approach in program design.

Definition of Object Oriented Program is:

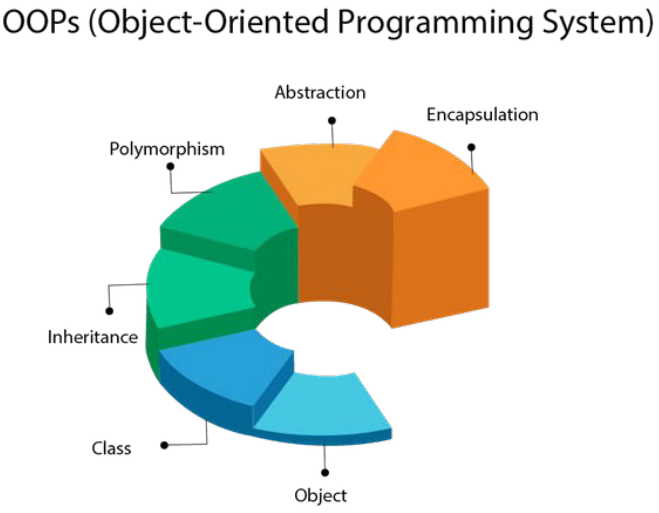
Object-oriented programming is an approach that provides a way of modularising programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand.

This means an **object** is considered to be a partitioned area of computer memory that stores data and a set of operations that can access that data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.

1.2 Basic OOPS Concepts in Java with Examples

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

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

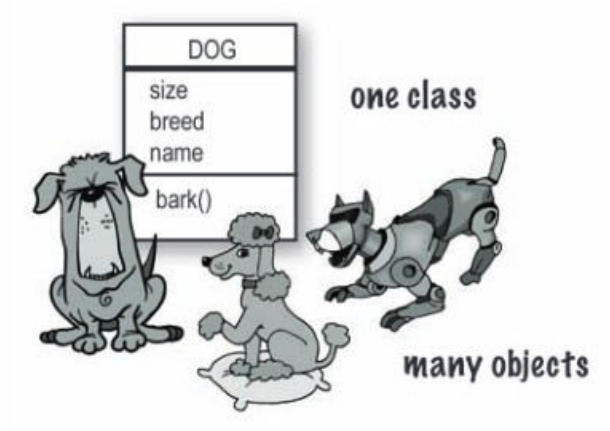


1. Objects:

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

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

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



2. Classes:

A class is a ‘data-type’ and an object as a ‘variable’ of that type. Any number of objects can be created after a class is created.

The collection of objects of similar types is termed as a class. For Example, apple, orange, and mango are the objects of the class Fruit. Classes behave like built-in data types of a programming language but are user-defined data types.

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

3. Data Encapsulation

The wrapping up of the data and methods into the single unit is known as **encapsulation**. The data is accessible only to those methods, which are wrapped in the class, and not to the outside world. This insulation of data from the direct access of the program is called data hiding. Encapsulation of the object makes it possible for the objects to be treated like ‘black boxes’ that perform a specific task without any concern for internal implementation.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Encapsulation- Objects as "black-boxes"

4. Data Abstraction:

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and defined as a list of abstract attributes such as size, weight and cost , and functions to operate on these attributes. Sometimes the **attributes are called members** and **functions called methods** or members functions.

Since the classes use the concept of data abstraction, they are known as **Abstract data types(ADT).**

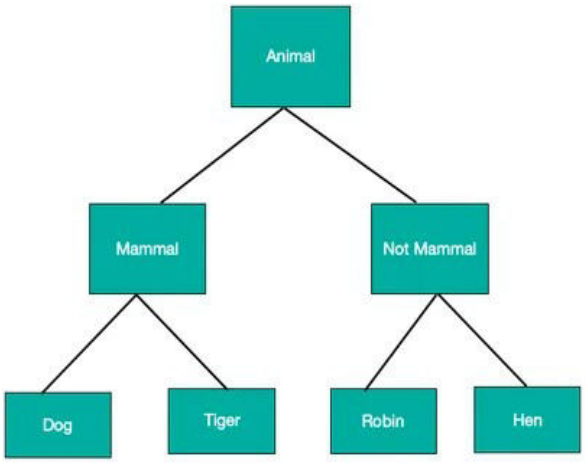
Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing.

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

5. Inheritance:

Inheritance is the process by which objects of one class acquire some properties of objects of another class.

Inheritance supports the concept of hierarchical classification. For Example, a bird Robin is part of the class, not a mammal, which is again a part of the class Animal. The principle behind this division is that each subclass shares common characteristics from the class from its parent class.



Inheritance

In OOP, the idea of inheritance provides the concept of reusability. It means that we can add additional features to parent class without modification; this is possible by deriving a new class from the parent class. The new class consists of the combined features from both the classes. In Java, the derived class is also known as the subclass.

Types of Inheritance

- Single
- Multiple
- Multilevel
- Hybrid

6. Polymorphism:

Polymorphism is an important OOP concept; Polymorphism means the ability to take many forms. For Example, an operation may give different answers in different instances. If we add two numbers, the answer would be a sum. If we add two strings the answer would be a concatenated string.

A single function name can be used to perform different tasks, but the function should have different number of arguments or different types of arguments. Polymorphism is extensively used in implementing inheritance.

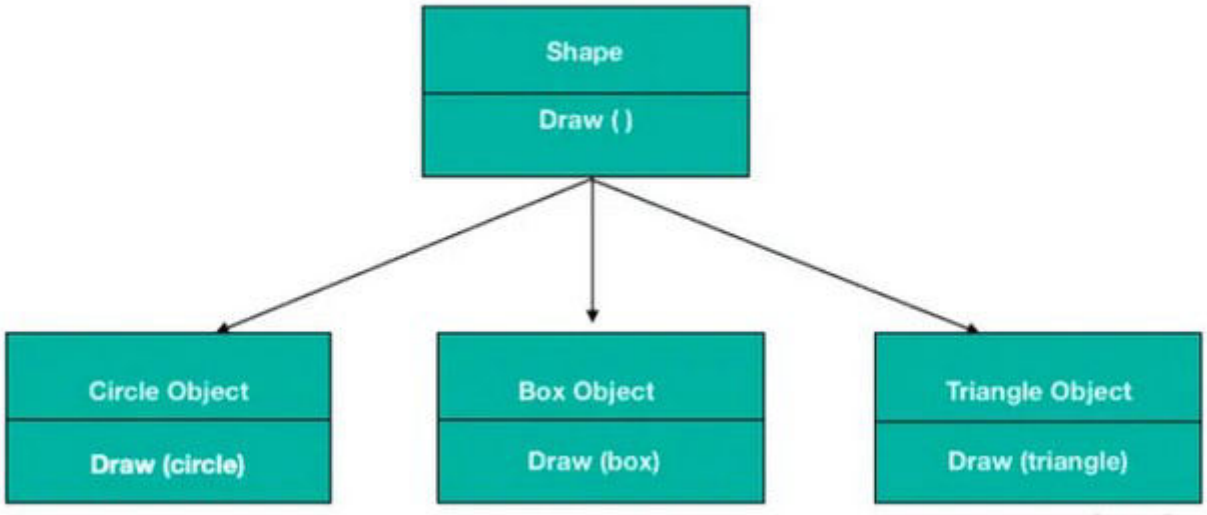


Figure: Polymorphism

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

Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

7. Dynamic Binding

Binding is the process of linking a procedure call to the code to be executed in response to the call. Dynamic Binding means that the code associated with the given procedure call is not known until the time of the call at runtime. It is associated with inheritance and polymorphism.

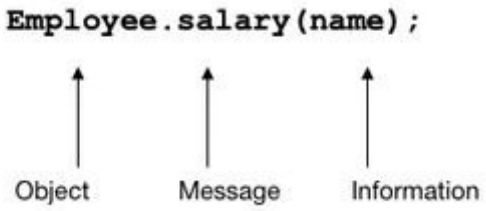
8. Message Communication

An object -oriented program consists of a set of objects that communicate with each other.

Objects communication with one another by sending and receiving information much the same way as people pass messages to one another.

Message passing involves specifying the name of the object, the name of the method(message) and information to be sent.

For Example, consider the statement.



1.3 BENEFITS AND APPLICATIONS OF OOP:

Benefits of OOP:

OOP offers several benefits to both the program designer and the user.

- Through **inheritance**, we can eliminate *redundant code* and extend the use of existing classes.
- ***Message passing technique*** between objects for communication makes interface description with external systems much more straightforward.
- The principle of ***data hiding*** helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.

- It has **multiple objects** to co-exist without any interference.
- It is possible to **map objects** in the problem domain within a program.
- It is easy to **partition** the work in a project based on objects.
- Wrapping up of **data into a single unit.**
- Object oriented systems can be easily upgraded from small to large systems.
- ***Software complexity*** can be easily managed.
- Code recycle and reuse.

Applications of OOP:

OOP is successful in this type of applications because it can simplify a complex problem. The promising areas for application of OOP includes:

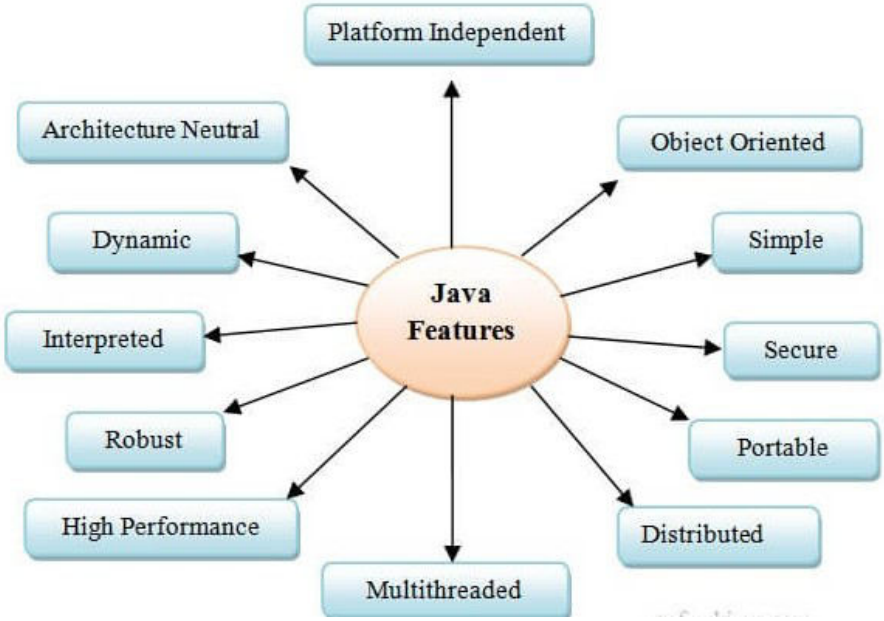
- Real-time systems design
- Simulation and modeling system
- Object oriented databases
- Hypertext, and hypermedia
- AI and expert systems
- Neural network and parallel programming
- Decision support and Office Automation Systems
- CIM/CAD/CAM systems
- Client server system

1.4 Differences between Procedure Oriented Programming(POP) and Object Oriented Programming(OOP):

Procedure Oriented Programming	Object Oriented Programming
1. In POP, program is **divided into** small parts called **functions.**	1. In OOP, program is **divided into** parts called **objects**
2. In POP, **importance** is not given to data but to functions as well as sequence of actions to be done.	2. In OOP, **Importance** is given to the data rather than procedures or functions because it works as a **real world.**
3. POP follows **top down approach.**	3. OOP follows **Bottom Up approach**
4. POP does not have any **access specifier**	4. OOP has **access specifiers** named public, private, protected etc.
5. In POP, data **can move** freely from function to function in the system	5. In OOP, objects **can move** and communicate with each other through member functions
6. **To add new data and function** in POP is not so easy.	6. OOP provides an easy way **to add new data and function.**
7. In POP, most function uses global **data** for sharing that can be **accessed** freely from function to function in the system	7. In OOP, **data** can not move easily from function to function, it can be kept public or private so we control the **access** of data.
8. POP does not have any proper way for **hiding data** so it is **less secure**	8. OOP provides **data hiding** so provides **more security**

9. In POP, **Overloading** is not possible	9. In OOP, **overloading** is possible in the the form of function overloading and operator overloading
10. **Example** of POP are C, FORTRAN, Pascal	10. **Example** of OOP are: C++ , JAVA, VB.NET, C#.NET

1.5 FEATURES OF JAVA:



1. Simple:-

Java is a simple programming language because:

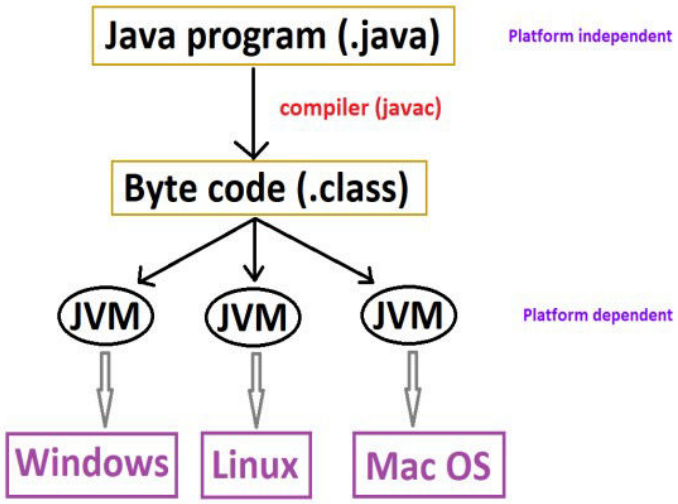
- Java technology has eliminated all the difficult and confusion oriented concepts like pointers, multiple inheritance in the java language.
- The c,cpp syntaxes easy to understand and easy to write. Java maintains C and CPP syntax mainly hence java is simple language.
- Java tech takes less time to compile and execute the program.

2. Object Oriented:-

Java is object oriented technology because to represent total data in the form of object. By using object reference we are calling all the methods, variables which is present in that class.

3. Platform Independent :-

- Compile the Java program on one OS (operating system)that compiled file can execute in any OS(operating system) is called Platform Independent Nature.
- The java is platform independent language. The java applications allow its applications compilation one operating system that compiled (.class) files can be executed in any operating system.



4. Architectural Neutral:-

Java tech applications compiled in one Architecture (hardware----RAM, Hard Disk) and that Compiled program runs on any hardware architecture (hardware) is called Architectural Neutral.

5. Portable:-

In Java tech the applications are compiled and executed in any OS (operating system) and any Architecture (hardware) hence we can say java is a portable language.

6. Robust:-

Any technology if it is good at two main areas it is said to be ROBUST
1. Exception Handling
2. Memory Allocation

JAVA is Robust because

- JAVA is having very good predefined Exception Handling mechanism whenever we are getting exception we are having meaning full information.
- JAVA is having very good memory management system that is Dynamic Memory (at runtime the memory is allocated) Allocation which allocates and deallocates memory for objects at runtime.

7. Secure:-

- To provide implicit security Java provide one component inside JVM called Security Manager.
- To provide explicit security for the Java applications we are having very good predefined library in the form of java.Security.package.

8. Dynamic:-

Java is dynamic technology it follows dynamic memory allocation(at runtime the memory is allocated) and dynamic loading to perform the operations.

9. Distributed:-

By using JAVA technology we are preparing standalone applications and Distributed applications.

Standalone applications are java applications it doesn't need client server architecture.

web applications are java applications it need client server architecture.

Distributed applications are the applications the project code is distributed in multiple number of jvm's.

10. Multithreaded: -

- Thread is a light weight process and a small task in large program.
- If any tech allows executing single thread at a time such type of technologies is called single threaded technology.
- If any technology allows creating and executing more than one thread called as multithreaded technology called JAVA.

11. Interpretive:-

JAVA tech is both Interpretive and Completive by using Interpreter we are converting source code into byte code and the interpreter is a part of JVM.

12. High Performance:-

If any technology having features like Robust, Security, Platform Independent, Dynamic and so on then that technology is high performance

1.6 DIFFERENCES BETWEEN C ,C++ AND JAVA:

Java differs from C and C++ in many ways. Few major differences are:

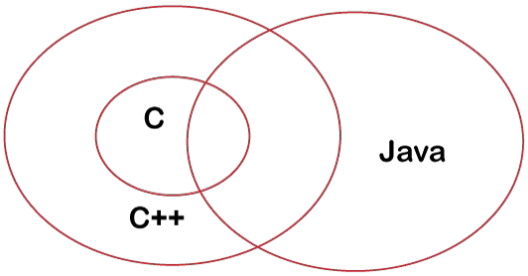
Java and C

The major difference between java and C is that

- Java is an object oriented language where as C is a structured programming language.
- Java doesnot include the C unique statement keywords **sizeof**, **typedef**.
- Java doesnot contain the data types **struct** and **union** .
- Java doesnot define the type modifiers keywords **auto**, **extern**, **register** , **signed** and **unsigned**
- Java doesnot support an explicit pointer type.
- Java doesnot have a preprocessor and therefore we cannot use **#define**, **#include** and **typedef** statements.
- Java adds new operators such as **instanceof** and **>>>**
- Java adds labelled **break** and **continue** statements.

Java and C++

- Java does not support **operator overloading**.
- Java does not have **template classes** as in C++
- Java doesnot support **multiple inheritance** of classes. This is accomplished using a new feature called "i**nterface**".



- Java does not use **pointers**.
- Java has replaced the **destructor** function with **finalize()** function.
- There are **no header files** in java.

Differences of C, C++, Java languages

C-language	*Cpp-language*	Java -language
1) Author: **Dennis Ritchie**	1)Author : **Bjarne Stroustrup**	1) Author : **James Gosling**
2) Implementation languages: BCPL, B…	2) implementation languages are c ,ada,ALGOL68…..	2) implementation languages are C,CPP,ObjectiveC…….
3) In C lang program, Execution starts from main method called by **operating system.**	3) program execution starts from main method called by **operating system.**	3) program execution starts from main method called by **JVM(Java Virtual Machine)**
4) In c-lang the predefined support is available in the form of header files **Ex:- stdio.h, conio.h**	4)cpp language the predefined is maintained in the form of header files. **Ex:- iostream.h**	4)In java predefined support available in the form of packages. **Ex: -java.lang, java.io**
5) The header files contain predefined functions. **Ex:- printf,scanf…..**	5) The header files contains predefined functions. **Ex:- cout, cin….**	5) The packages contains predefined classes and class contains predefined funtions. **Ex:- String,System**
6) To make available predefined support into our applications use #include statement. **Ex:- #include<stdio.h>**	6) To make available predefined support into our application use #include statement. **Ex:- #include<iostream>**	6) To make available predefined support into our application use import statement. **Ex:- import java.lang.*;** [*] mean all
7) To print some statements into output console use “printf” function. **printf(“hi ratan ”);**	7) To print the statements use “cout” function. **cout<<”hi raju”;**	7)To print the statements we have to use **System.out.println(“hi raju”);**
8)extensions used :-**.c ,.obj , .h**	8)extensions used :-**.cpp ,.h**	8)extensions used : -**.java, .class**
C –sample application:- #include<stdio.h> void main() { printf(“hello world”); }	**CPP –sample application:-** #include<iostream.h> void main() {cout<<“hello world”; }	

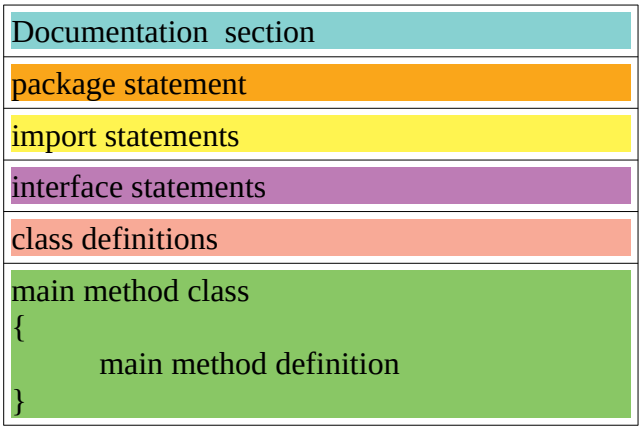
Java sample application:-

```
import java.lang.System;
import java.lang.String;
class Test
{
        public static void main (String[] args)
        {
                System.out.println (“Hello world”);
        }
}
```

2. OVERVIEW OF JAVA LANGUAGE

2.1 JAVA PROGRAM STRUCTURE:

A java program may contain many classes. Classes contain data methods that operate on the data members of the class. A java program may contain one or more sections as given below.



Documentation section:

It comprises a set of comment lines giving the name of the program, the author and other details which the programmer would like to refer to at a later stage. Java uses the comment style as **/**...*/** known as documentation comment.

package statements:

The first statement allowed in a java file is a package statement. This statement declares a package name and informs the compiler that the classes defined here belong to this package.
Ex: package student;

import statements:

This is similar to #include statement in C.
Ex: import student.test;
This statement instructs the interpreter to load the test class contained in the package student.

interface statements:

An interface is like a class but includes a group of method declarations. This is optional section.

class definition:

A java program may contain multiple class definitions. Classes are the primary and essential elements of a java program.

main method class:

The main method creates objects of various classes and establishes communication between them. On reaching the end of main, the program terminates and the control passes back to the operating system.

main method:

This is similar to the main() in C/C++. Every java application program must include the main() method. This is the starting point for the interpreter to begin the execution of the program. A java application can have any number of classes but only one of them must include the main() method to initiate the execution. The main() form is

public static void main(String args[ ])

This line contains a number of keywords public,static, and void

public: The keyword public is an access specifier that declares the main method as unprotected and therefore making it accessible to all other classes.
static: The main( ) must always be declared as static since the interpreter uses this method before any objects are created.
void: The type modifier void states that the main method does not return any value

All parameters to a method are declared inside pair of parenthesis. Here String args[ ] declares a parameter named args[ ], which contains an array of objects of the class type String.

2.2. IMPLEMENTING JAVA PROGRAM:

Implementation of a java application program involves a series of steps. They include:

1. Creating the program
2. Compiling the program
3. Running the program

1. Creating the program:

```
class Test
{
        public static void main(String[] args)
        {
        System.out.println("Hello!");
        System.out.println("Welcome to the world of java");
        }
}
```

We must save this program in a file called **Test.java** ensuring that the filename contains the classname properly. This file is called the *source file.* Note that *a*ll the java source files will have the extension **.java.** Note also that if a program contains multiple classes, the file name must be the classname of the class containing the main method.

2. Compiling the program:

To compile the program, we must run the Java compiler **javac**, with the name of the source file on the command line as shown below:

javac Test.java

If everything is OK, the **javac compiler** creates a file called **Test.class** containing the bytecodes of the program. Note that the compiler automatically names the bytecode file as **<classname>.class**

3. Running the program:

We need to use the Java interpreter to run a stand alone program. At the command prompt, type

java Test

Now the interpreter looks for the main method in the program and begins execution from there . When executed, our program displays the following:

```
student@student-HP-245-G6-Notebook-PC:~$ javac Test.java
student@student-HP-245-G6-Notebook-PC:~$ java Test
Hello!
Welcome to the world of java
```

2.3 VARIOUS TYPES OF JAVA TOKENS:

- A java program is basically a collection of classes .
- A class is defined by a set of declaration statements and methods containing executable statements.
- ***Smallest individual units in a program are known as tokens***.
- The compiler reorganizes them for building up expressions and statements.
- A java program is collection of tokens, comments and white spaces.
- **Java language includes five types of tokens.**

1. Identifiers
2. Reserved keywords
3. Literals
4. Operators
5. Seperators

Seperators: Seperators are symbols used to indicate where groups of code are divided and arranged . They basically define the shape and function of our code. Seperators are

parenthesis ( )
braces { }
brackets[ ]
semicolon ;
comma ,
period .

DATA TYPES

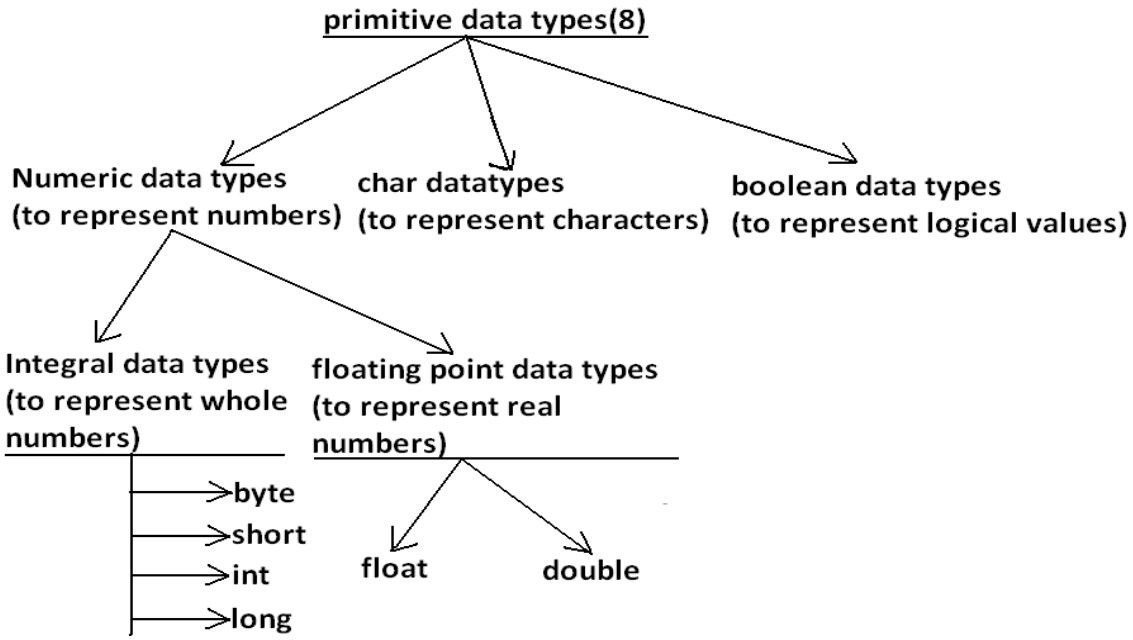
Agenda:

Data types
- Integral data types
  - byte
  - short
  - int
  - long
- floating point data types
- boolean data type
- char data type
- Java is pure object oriented programming or not ?
- Summary of java primitive data type

Data types:

Every variable has a type, every expression has a type and all types are strictly defined. More over every assignment should be checked by the compiler by the type compatibility Hence java language is considered as strongly typed programming language.

Diagram:

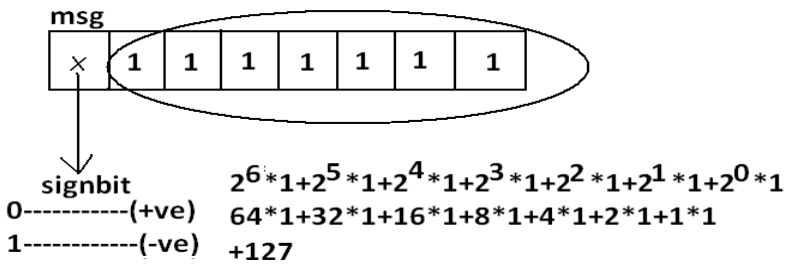


Except Boolean and char all remaining data types are considered as signed data types because we can represent both "+ve" and"-ve" numbers.

Integral data types :

byte:

```
Size: 1byte (8bits)
Maxvalue: +127
Minvalue:-128
Range:-128to 127[-2^7 to 2^7-1]
```



- The most significant bit acts as sign bit. "0" means "+ve" number and "1" means "–ve" number.
- "+ve" numbers will be represented directly in the memory whereas "–ve" numbers will be represented in 2's complement form.

```
Example:
byte b=10;
byte b2=130;//C.E:possible loss of precision
                    found : int
                    required : byte
byte b=10.5;//C.E:possible loss of precision
byte b=true;//C.E:incompatible types
byte b="ashok";//C.E:incompatible types
                         found : java.lang.String
                         required : byte
```

byte data type is best suitable if we are handling data in terms of streams either from the file or from the network.

short:

The most rarely used data type in java is short.

```
Size: 2 bytes
Range: -32768 to 32767(-2^15 to 2^15-1)
```

Example:

```
short s=130;
short s=32768;//C.E:possible loss of precision
```

```
short s=true;//C.E:incompatible types
```
short data type is best suitable for 16 bit processors like 8086 but these processors are completely outdated and hence the corresponding short data type is also out data type.

int:

This is most commonly used data type in java.

```
Size: 4 bytes
Range:-2147483648 to 2147483647 (-2^31 to 2^31-1)
```
Example:
```
int i=130;
int i=10.5;//C.E:possible loss of precision
int i=true;//C.E:incompatible types
```

long:

Whenever int is not enough to hold big values then we should go for long data type.

Example:
To hold the no. of characters present in a big file int may not enough hence the return type of length() method is long.
```
long l=f.length();//f is a file
Size: 8 bytes
Range:-2^63  to 2^63-1
```
Note: All the above data types (byte, short, int and long) can be used to represent whole numbers. If we want to represent real numbers then we should go for floating point data types.

Floating Point Data types:

float	double
If we want to 5 to 6 decimal places of accuracy then we should go for float.	If we want to 14 to 15 decimal places of accuracy then we should go for double.
Size:4 bytes.	Size:8 bytes.
Range:-3.4e38 to 3.4e38.	-1.7e308 to1.7e308.
float follows single precision.	double follows double precision.

boolean data type:

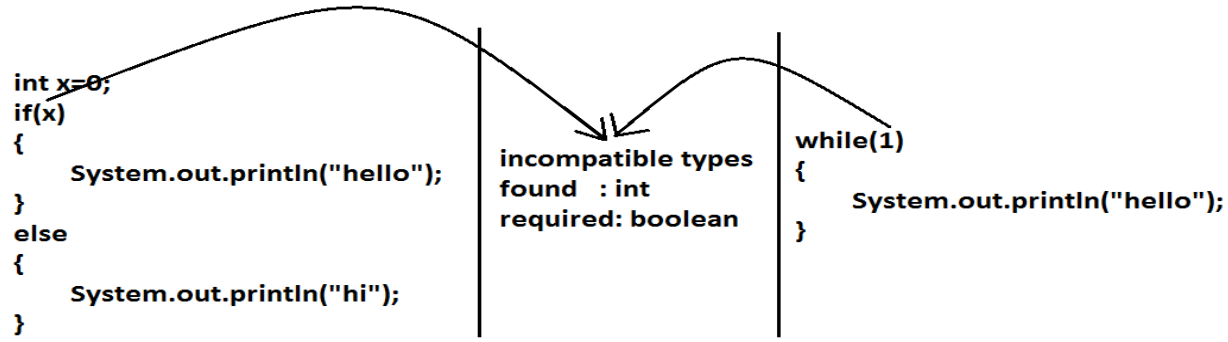
```
Size: Not applicable (virtual machine dependent)
Range: Not applicable but allowed values are true or false.
```

Which of the following boolean declarations are valid?

Example 1:

```
boolean b=true;
boolean b=True;//C.E:cannot find symbol
boolean b="True";//C.E:incompatible types
boolean b=0;//C.E:incompatible types
```

Example 2:



char data type:

In old languages like C & C++ are ASCII code based the no. of ASCII code characters are **< 256** to represent these 256 characters 8 - bits enough hence char size in old languages 1 byte.

In java we are allowed to use any worldwide alphabets character and java is Unicode based and no. of unicode characters are **> 256 and <= 65536** to represent all these characters one byte is not enough compulsory we should go for 2 bytes.

```
Size: 2 bytes
Range: 0 to 65535
```

Example:

```
char ch1=97;
char ch2=65536;//C.E:possible loss of precision
```

Java is pure object oriented programming or not?

Java is not considered as pure object oriented programming language because several oops features (like multiple inheritance, operator overloading) are not supported by java.

Moreover we are depending on primitive data types which are non objects.

Summary of java primitive data type:

data type	Size	Range	Corresponding Wrapper class	Default value
byte	1 byte	-2^7 to 2^7-1(-128 to 127)	Byte	0
short	2 bytes	-2^{15} to $2^{15}-1$ (-32768 to 32767)	Short	0

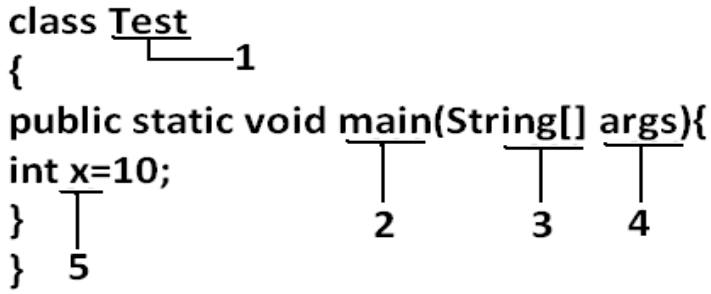
int	4 bytes	-2^{31} to $2^{31}-1$ (-2147483648 to 2147483647)	Integer	0
long	8 bytes	-2^{63} to $2^{63}-1$	Long	0
float	4 bytes	-3.4e38 to 3.4e38	Float	0.0
double	8 bytes	-1.7e308 to 1.7e308	Double	0.0
boolean	Not applicable	Not applicable(but allowed values true|false)	Boolean	false
char	2 bytes	0 to 65535	Character	0(represents blank space)

The default value for the object references is "null".

Identifier :

A name in java program is called identifier. It may be class name, method name, variable name.

Example:



Rules to define java identifiers:

Rule 1: The only allowed characters in java identifiers are:

```
1) a to z
2) A to Z
3) 0 to 9
4) _ (underscore)
5) $
```

Rule 2: If we are using any other character we will get compile time error.

Example:

```
1) total_number-------valid
2) Total#-------------------invalid
```

Rule 3: identifiers are not allowed to start with digit.

Example:

```
1) ABC123---------valid
2) 123ABC---------invalid
```

Rule 4: java identifiers are case sensitive. java language itself treated as case sensitive language.

Example:

```
class Test{
int number=10;
int Number=20;
int NUMBER=20; we can differentiate with case.
int NuMbEr=30;
}
```

Rule 5: There is no length limit for java identifiers but it is not recommended to take more than 15 lengths.

Rule 6: We can't use reserved words as identifiers.

Example:

```
int if=10; --------------invalid
```

Rule 7: **All predefined java class names and interface names we use as identifiers.**

Example 1:

```
class Test
{
public static void main(String[] args){
int String=10;
System.out.println(String);
}}
Output:
10
```

Example 2:

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

int Runnable=10;
System.out.println(Runnable);
}}
Output:
10
```

Even though it is legal to use class names and interface names as identifiers but it is not a good programming practice.

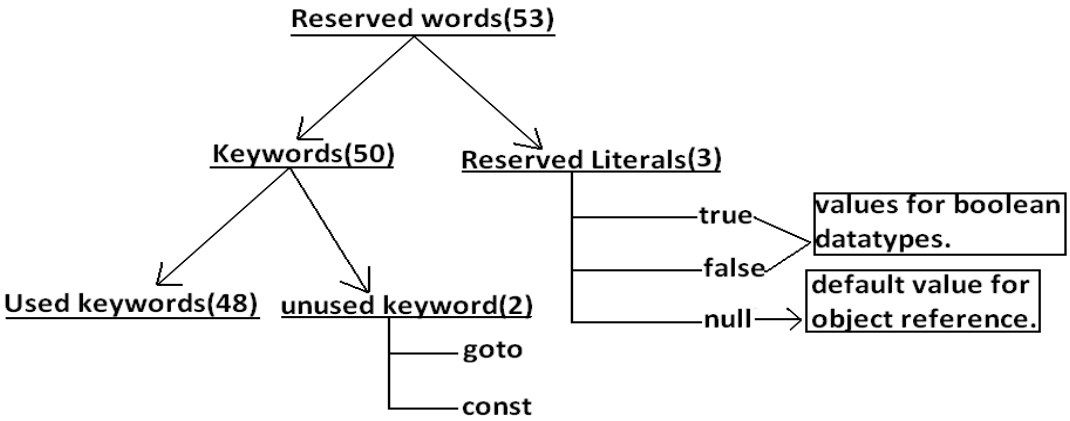
Which of the following are valid java identifiers?

1)_$_ (valid)
2)Ca$h (valid)
3)Java2share (valid)
4)all@hands (invalid)
5)123abc (invalid)
6)Total# (invalid)
7)Int (valid)
8)Integer (valid)
9)int (invalid)
10) tot123

Reserved words:

In java some identifiers are reserved to associate some functionality or meaning such type of reserved identifiers are called reserved words.

Diagram:



Reserved words for data types: (8)

```
1)     byte
2)     short
3)     int
4)     long
5)     float
6)     double
7)     char
8)     boolean
```

Reserved words for flow control:(11)

```
1)     if
2)     else
3)     switch
4)     case
5)     default
6)     for
7)     do
8)     while
9)     break
10)    continue
11)    return
```

Keywords for modifiers:(11)

```
1)     public
2)     private
3)     protected
4)     static
5)     final
6)     abstract
7)     synchronized
8)     native
9)     strictfp(1.2 version)
10)    transient
11)    volatile
```

Keywords for exception handling:(6)

```
1)     try
2)     catch
3)     finally
4)     throw
5)     throws
6)     assert(1.4 version)
```

Class related keywords:(6)

```
1)     class
2)     package
3)     import
4)     extends
5)     implements
6)     interface
```

Object related keywords:(4)

```
1)     new
2)     instanceof
3)     super
4)     this
```

Void return type keyword:

If a method won't return anything compulsory that method should be declared with the void return type in java but it is optional in C++.

```
1)     void
```

Unused keywords:

goto: Create several problems in old languages and hence it is banned in java.
Const: Use final instead of this.
By mistake if we are using these keywords in our program we will get compile time error.

Reserved literals:

```
1)     true   values for boolean data type.
2)     false
3)     null-----default value for object reference.
```

Enum:

This keyword introduced in 1.5v to define a group of named constants

```
Example:
enum Beer
{
KF, RC, KO, FO;
}
```

Conclusions :

1. All reserved words in java contain only lowercase alphabet symbols.
2. New keywords in java are:

```
strictfp-----------1.2v
assert-------------1.4v
enum---------------1.5v
```

3. In java we have only new keyword but not delete because destruction of useless objects is the responsibility of Garbage Collection.

```
instanceof  but not  instanceOf
strictfp    but not  strictFp
const       but not  Constant
syncronized but not  syncronize
extends     but not  extend
implements  but not  implement
import      but not  imports
int         but not  Int
```

Which of the following list contains only java reserved words ?

1. final, finally, finalize (invalid) //here finalize is a method in Object class.

2. throw, throws, thrown(invalid) //thrown is not available in java
3. break, continue, return, exit(invalid) //exit is not reserved keyword
4. goto, constant(invalid) //here constant is not reserved keyword
5. byte, short, Integer, long(invalid) //here Integer is a wrapper class
6. extends, implements, imports(invalid) //imports keyword is not available in java
7. finalize, synchronized(invalid) //finalize is a method in Object class
8. instanceof, sizeOf(invalid) //sizeOf is not reserved keyword
9. new, delete(invalid) //delete is not a keyword
10. None of the above(valid)

Which of the following are valid java keywords?

1. public(valid)
2. static(valid)
3. void(valid)
4. main(invalid)
5. String(invalid)
6. args(invalid)

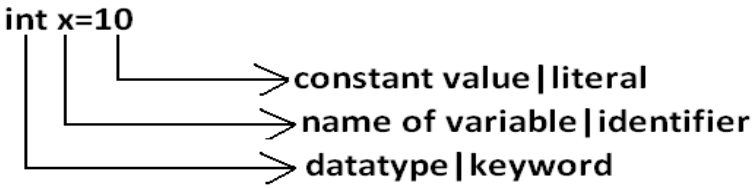
LITERALS

Agenda:

Literals
- **Integral Literals**
- **Floating Point Literals**
- **Boolean literals**
- **Char literals**
- **String literals**

Literals:

Any constant value which can be assigned to the variable is called literal.
Example:



Integral Literals:

For the integral data types (byte, short, int and long) we can specify literal value in the following ways.

1) Decimal literals: Allowed digits are 0 to 9.
Example: int x=10;

2) Octal literals: Allowed digits are 0 to 7. Literal value should be **prefixed with zero**.
Example: int x=**0**10;

3) Hexa Decimal literals:

- The allowed digits are 0 to 9, A to F.
- For the extra digits we can use both upper case and lower case characters.
- This is one of very few areas where java is not case sensitive.
- Literal value should be **prefixed with 0x(or)0X.**

Example: int x=**0x**10;

These are the only possible ways to specify integral literal.

Which of the following are valid declarations?

1. int x=0777; //(valid)
2. int x=0786; //C.E:integer number too large: 0786(invalid)
3. int x=**0x**FACE; (valid)
4. int x=**0x**beef; (valid)
5. int x=0xBeer; //C.E:';' expected(invalid) //:int x=0xBeer; ^// ^
6. int x=**0x**abb2cd;(valid)

Example:

```
int x=10;
int y=010;
int z=0x10;
System.out.println(x+"----"+y+"----"+z); //10----8----16
```

By default every integral literal is int type but we can specify explicitly as long type by **suffixing with small "l" (or) capital "L".**

Example:

```
int x=10;(valid)
long l=10L;(valid)
long l=10;(valid)
int x=10l;//C.E:possible loss of precision(invalid)
                found : long
                required : int
```

There is no direct way to specify byte and short literals explicitly. But whenever we are assigning integral literal to the byte variables and its value within the range of byte compiler automatically treats as byte literal. Similarly short literal also.

Example:

```
byte b=127;(valid)
byte b=130;//C.E:possible loss of precision(invalid)
short s=32767;(valid)
short s=32768;//C.E:possible loss of precision(invalid)
```

Floating Point Literals:

Floating point literal is by default double type but we can specify explicitly as float type by **suffixing with f or F.**

Example:

```
float f=123.456;//C.E:possible loss of precision(invalid)
float f=123.456f;(valid)
double d=123.456;(valid)
```

We can specify explicitly floating point literal as double type by suffixing with d or D.

Example:

```
double d=123.456D;
```

We can specify floating point literal only in decimal form and we can't specify in octal and hexadecimal forms.

Example:

```
double d=123.456;(valid)
double d=0123.456;(valid)  //it is treated as decimal value  but
not octal
double d=0x123.456;//C.E:malformed floating point
Literal(invalid)
```

Which of the following floating point declarations are valid?

1. float f=123.456; //C.E:possible loss of precision(invalid)
2. float f=123.456D; //C.E:possible loss of precision(invalid)
3. double d=0x123.456; //C.E:malformed floating point literal(invalid)
4. double d=0xFace; (valid)
5. double d=0xBeef; (valid)

We can assign integral literal directly to the floating point data types and that integral literal can be specified in decimal , octal and Hexa decimal form also.

Example:

```
double d=0xBeef;
System.out.println(d);//48879.0
```

But we can't assign floating point literal directly to the integral types.

Example:

```
int x=10.0;//C.E:possible loss of precision
```

We can specify floating point literal even in exponential form also(significant notation).

Example:

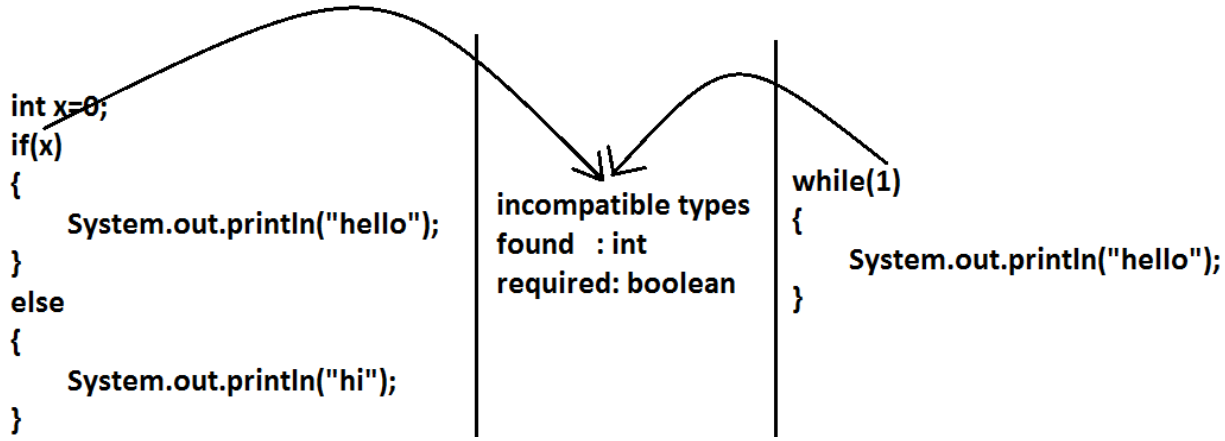
```
double d=10e2;//==>10*10^2(valid)
System.out.println(d);//1000.0
float f=10e2;//C.E:possible loss of precision(invalid)
float f=10e2F;(valid)
```

Boolean literals:

The only allowed values for the boolean type are true (or) false where case is important.
i.e., lower case

Example:

1. boolean b=true;(valid)
2. boolean b=0;//C.E:incompatible types(invalid)
3. boolean b=True;//C.E:cannot find symbol(invalid)
4. boolean b="true";//C.E:incompatible types(invalid)



Char literals:

1) A char literal can be represented as single character within **single quotes.**

Example:

1. char ch='a';(valid)
2. char ch=a;//C.E:cannot find symbol(invalid)
3. char ch="a";//C.E:incompatible types(invalid)
4. char ch='ab';//C.E:unclosed character literal(invalid)

2) We can specify a char literal as integral literal which represents Unicode of that character. We can specify that integral literal either in decimal or octal or hexadecimal form but allowed values range is 0 to 65535.

Example:

1. char ch=97; (valid)
2. char ch=0xFace; (valid)
   System.out.println(ch); //?
3. char ch=65536; //C.E: possible loss of precision(invalid)

3) We can represent a char literal by Unicode representation which is nothing but **'\uxxxx'** (4 digit hexa-decimal number) .

Example:

1. char ch='\ubeef';

2. char ch1='\u0061';
   System.out.println(ch1); //a
3. char ch2=\u0062; //C.E:cannot find symbol
4. char ch3='\iface'; //C.E:illegal escape character
5. Every escape character in java acts as a char literal.

Example:

```
1) char ch='\n';   //(valid)
2) char ch='\l';   //C.E:illegal escape character(invalid)
```

Escape Character	Description
\n	New line
\t	Horizontal tab
\r	Carriage return
\f	Form feed
\b	Back space character
\'	Single quote
\"	Double quote
\\	Back space

Which of the following char declarations are valid?

1. char ch=a; //C.E:cannot find symbol(invalid)
2. char ch='ab'; //C.E:unclosed character literal(invalid)
3. char ch=65536; //C.E:possible loss of precision(invalid)
4. char ch=\uface; //C.E:illegal character: \64206(invalid)
5. char ch='/n'; //C.E:unclosed character literal(invalid)
6. none of the above. (valid)

String literals:

Any sequence of characters with in double quotes is treated as String literal.

Example:
String s="Ashok"; (valid)

Operator:

Operator is a symbol that performs certain operations.

Java provides the following set of operators

1. Arithmetic operators
2. Increment and decrement operators
3. Relational or comparison operators
4. Bitwise operators, Shift operators
5. Short circuit Logical operators
6. Assignment operators
7. Conditional (?:) operator

1. Arithmetic operators:

+	**addition**
-	**subtraction**
*****	**multiplication**
/	**Division operator**
%	**Modulo operator**

If we apply any arithmetic operation b/w 2 variables a & b, the result type is always

max (int, type of a , type of b)

Ex:

byte + byte = int
byte + short = int
short + short = int
short + long = long
double + float = double
int + double = double
char + char = int
char + int = int
char + int = int
char + double = double

In integral arithmetic (byte, int , short, long) there is no way to represent infinity, if infinity is the result we will get the ArithmeticException/ by zero

```
System.out.println(10/0);
//output ArithmeticException/ by zero
```

But in floating point arithmetic (float,double) , there is a way represents infinity.

```
System.out.println(10/0.0);
//output:       infinity
```

Arithmetic exception:

1. It is a RuntimeException but not compile time error
2. It occurs only in integral arithmetic but not in floating point arithmetic.
3. The only operations which cause ArithmeticException are: '/' and '%'

2. Increment & decrement operators:

Increment operator	**Pre-increment**	**Ex: y = ++x ;**
	Post-increment	**Ex: y = x++ ;**

Decrement operator	**Pre-decrement**	**Ex: y = --x ;**
	Post-decrement	**Ex: y = x--;**

The following table will demonstrate the use of increment and decrement operators.

Expression	Initial value of x	Value of y	Final value of x
y = ++x	**20**	**21**	**21**
y = x++	**20**	**20**	**21**
y = --x	**20**	**19**	**19**
y= x--	**20**	**20**	**19**

1. Increment and decrement operators we can apply only for variables but not for constant values. Otherwise we will get compile time error.

2. We can apply increment or decrement operators even for primitive data types except boolean.

3. Relational operators

(<, <= , > , >=)

We can apply relational operators for every primitive type except boolean .

Ex:

```
System.out.println(10>10.5);  //false
```

4. Equality operators:

(==, !=)

We can apply equality operators for every primitive type including boolean type also

Ex:

```
System.out.println(10==20); //false
```

5. Shift operators:

<< left shift operator:

After shifting the empty cells we have to fill with zero.

```
System.out.println(10<<2);        //40
```

0	0	0	0	1	0	1	0

0	0	1	0	1	0	0	0

>> right shift operator :

After shifting the empty cells, we have to fill with sign bit. (0 for +ve and 1 for –ve)

```
System.out.println(10>>2);          //2
```

0	0	0	0	1	0	1	0

0	0	0	0	0	0	1	0

6. Bitwise operators(& ,| ,^)

operator	Description
&(AND)	If both arguments are true then only result is true.
|(OR)	if atleast one argument is true. Then the result is true.
^(X-OR)	if both are different arguments. Then the result is true
~	Bitwise complement operator i.e, 1 means 0 and 0 means 1

Ex:

```
System.out.println(true&false); //false
```

We can apply bitwise operators even for integral types also.

```
System.out.println(4&5);      //4
```

Bitwise complement operator: (~)

We can apply this operator only for integral types but not for boolean types.

We have to apply complement for total bits.

Ex:

```
System.out.println(~4);           //-5
```

Note: The most significant bit acts as sign bit. O value represents +ve number where as 1 represents –ve value

Positive numbers will be represented directly in the memory where as –ve numbers will be represented indirectly in 2's complement form

Boolean complement operator (!)

This operator is applicable only for Boolean types but not for integral types

```
System.out.println(!true);          //false
```

7. Short circuit logical operators (&& , ||)

Applicable only for Boolean types but not for integral types.

x&&y : y will be evaluated if and only if x is true.

x || y : y will be evaluated if and only if x is false.

8. Assignment operator:

There are 3 types of assignment operators

1. simple assignment:

Example:

```
int x = 10 ;
```

2. chained assignment :

Example:

```
a=b=c=d=20;
```

3.compound assignment :

Example:

```
+= , -= , *= , /= , %=
&= , |= , ^=
>>= , >>>= , <<=
```

9. Conditional operator(?:)

The only possible ternary operator in java is conditional operator.

Syntax:

```
X=firstValue if condition else secondValue
```

If condition is True then firstValue will be considered else secondValue will be considered

Ex 1:

```
int x=(10>20)?30:40;
System.out.println(x);          //40
```

Note: nesting of ternary operator is possible.

[ ] operator: we can use this operator to declare under construct/ create arrays.

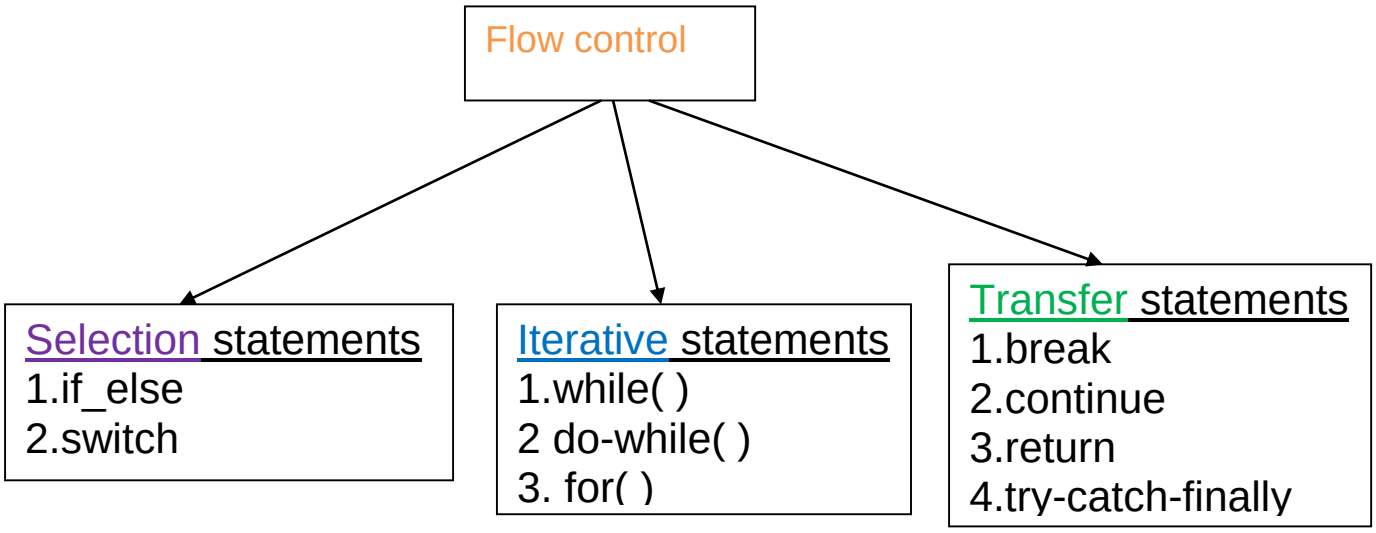
Java operator precedence:

1. unary	**Highest** **( ), [ ],** **++, --, ~,!**
2. Arithmetic	***, /, %,+, -**
3. Shift	**>>,>>>,<<**
4. Relational	**<,<=,>,>=**
5. Equality	**==,!=**
6. Bitwise	**&, ^, |**
7. Short circuit logical	**&&, ||**
8. Conditional	**?:**
9. Assignment	**=** **Lowest**

Flow control:

Flow control describes the order in which all the statements will be executed at run time.

Diagram:



java flow control Statements:-

There are three types of flow control statements in java

1) *Selection Statements*
2) *Iteration statements*
3) *Transfer statements*

1. *Selection Statements*

a. If b. If-else c. switch

If syntax:-

```
if (condition)
{       true body;      }
else
{       false body;     }
```

- If is taking condition that condition must be Boolean condition otherwise compiler will raise compilation error.
- The curly brasses are optional whenever we are taking single statements and the curly brasses are mandatory whenever we are taking multiple statemen

Ex-1:-

```
class Test
{       public static void main(String[ ] args)
        {       int a=10;       int b=20;
                if (a<b)
                {       System.out.println("if body/true body");      }
                else
                {       System.out.println("else body/false body ");     }
        }
}
```

Ex -2:- For the if the condition it is possible to provide Boolean values.

```
class Test
{       public static void main(String[ ] args)
        {       if (true)
                {       System.out.println("true body");       }
                else
                {       System.out.println("false body");       }
        }
}
```

Ex-3:-in c-language 0-false & 1-true but these convensions are not allowed in java.

```
class Test
{       public static void main(String[ ] args)
        {       if (0)
                {       System.out.println("true body");       }
                else
                {       System.out.println("false body");       }
        }
}
```

Switch statement:-

1) Switch statement is used to declare multiple selections.
2) Inside the switch It is possible to declare any number of cases but is possible to declare only one default.
3) Switch is taking the argument the allowed arguments are
   a. **byte b. short c. int d.char e.String(allowed in 1.7 version)**
4) float , double and long is not allowed for a switch argument because these are having more number of possibilities (float and double is having infinity number of possibilities) hence inside the switch statement it is not possible to provide float and double and long as a argument.
5) Based on the provided argument the matched case will be executed if the cases are not matched default will be executed.

Syntax:-

```
switch(argument)
{
   case  label1 : System.out.println(" ");break;
    case  label2 : System.out.println (" ");break;
            |                       |
    default      : System.out.println (" ");  break;
}
```

Eg-1:Normal input and normal output.

```
class Test
{       public static void main(String[ ] args)
        {       int a=10;
                switch (a)
                {
                case 10:System.out.println("sunny");          break;
                case 20:System.out.println("bunny");          break;
                case 30:System.out.println("chinny");          break;
                default:System.out.println("vinny");          break;
                }
        }
}
```

Ex-2:-Inside the switch the case labels must be unique; if we are declaring duplicate case labels the compiler will raise compilation error "duplicate case label".

```
class Test
{       public static void main(String[ ] args)
        {       int a=10;
                switch (a)
                {
                case 10:System.out.println("sunny");          break;
                case 10:System.out.println("bunny");          break;
                case 30:System.out.println("chinny");          break;
                default:System.out.println("vinny");          break;
                }       }}
```

Ex-3:Inside the switch for the case labels it is possible to provide expressions(10+10+20 , 10*4 , 10/2).

```
class Test
{       public static void main(String[ ] args)
        {       int a=100;
                switch (a)
                {
                case 10+20+70:System.out.println("sunny");          break;
                case 10+5:System.out.println("bunny");          break;
                case 30/6:System.out.println("chinny");          break;
                default:System.out.println("vinny");          break;
                }
        }
}
```

Eg-4:- Inside the switch the case label must be constant values. If we are declaring variables as a case labels the compiler will show compilation error "constant expression required".

```
class Test
{       public static void main(String[ ] args)
        {       int a=10;       int b=20;       int c=30;
                switch (a)
                {
                case a:System.out.println("sunny");          break;
                case b:System.out.println("bunny");          break;
                case c:System.out.println("chinny");          break;
                default:System.out.println("vinny");          break;
                }
        }
}
```

Ex-5:-inside the switch the default is optional.

```
class Test
{       public static void main(String[ ] args)
        {       int a=10;
                switch (a)
                {       case 10:System.out.println("10");       break;
                }
        }
};
```

Ex 6:-Inside the switch cases are optional part.

```
class Test
{       public static void main(String[ ] args)
        {       int a=10;
                switch (a)
                {       default: System.out.println("default");             break;
                }
        }
};
```

Ex 7:-inside the switch both cases and default Is optional.

```
public class Test
{    public static void main(String[ ] args)
    { int a=10;
       switch(a)
       {       }
    }
}
```

Ex -8:-inside the switch independent statements are not allowed. If we are declaring the statements that statement must be inside the case or default.

```
public class Test
{    public static void main(String[ ] args)
    {       int x=10;
       switch(x)
       {            System.out.println("Hello World");
       }
    }
}
```

Ex-9:-internal conversion of char to integer.

Unicode values a-97 A-65

```
class Test
{       public static void main(String[ ] args)
        {       int a=65;
                switch (a)
                {
                case 66:System.out.println("10");       break;
                case 'A':System.out.println("20");      break;
                case 30:System.out.println("30");       break;
```

```
                default: System.out.println("default");    break;
                }
        }
};
```

Ex -10: internal conversion of integer to character.

```
class Test
{       public static void main(String[ ] args)
        {       char ch='d';
                switch (ch)
                {
                case  100:System.out.println("10");   break;
                case 'A':System.out.println("20");          break;
                case 30:System.out.println("30");          break;
                default: System.out.println("default"); break;
                }
        }
};
```

Ex-11:-Inside the switch statement beak is optional. If we are not providing break statement at that situation from the matched case onwards up to break statement is executed if no beak is available up to the end of the switch is executed. This situation is called as fall though inside the switch case.

```
class Test
{       public static void main(String[ ] args)
        {       int a=10;
                switch (a)
                {
                case 10:System.out.println("10");
                case 20:System.out.println("20");
                case 40:System.out.println("40");                break;
                default: System.out.println("default");          break;
                }
        }
};
```

Ex-12:- inside the switch the case label must match with provided argument data type otherwise compiler will raise compilation error "incompatible types".

```
class Test
{       public static void main(String[ ] args)
        {       char ch='a';
                switch (ch)
                {
                case "aaa"      :System.out.println("sunny");                   break;
                case  65        :System.out.println("bunny");                   break;
                case 'a'        :System.out.println("chinny");                  break;
                default         :System.out.println("default")                  break;
                }
        }
}
```

Ex-13 :-inside the switch we are able to declare the default statement starting or middle or end of the switch.

```
class Test
{       public static void main(String[ ] args)
        {       int a=100;
                switch (a)
                {
                default: System.out.println("default");
                case 10:System.out.println("10");
                case 20:System.out.println("20");
                }
        }
};
```

Ex -14:-The below example compiled and executed only in above 1.7 version because switch is taking String argument from 1.7 version.

```
class Sravya
{    public static void main(String[] args)
    { String str = “aaa”;
       switch (str)
        {       case "aaa" : System.out.println("Hai");          break;
                case "bbb" : System.out.println("Hello");        break;
                case "ccc" : System.out.println("how");          break;
                 default   : System.out.println("what");         break;
        }
    }
}
```

Ex-15:-inside switch the case labels must be within the range of provided argument data type otherwise compiler will raise compilation error “possible loss of precision”.

```
class Test
{       public static void main(String[] args)
        {       byte b=125;
                switch (b)
                {
                case 125:System.out.println("10");
                case 126:System.out.println("20");
                case 127:System.out.println("30");
                case 128:System.out.println("40");
                default:System.out.println("default");
                }
        }
};
```

***Iteration Statements*:-**

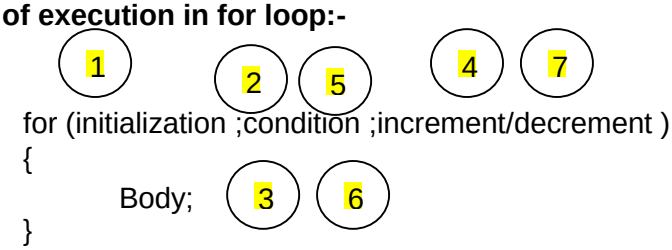
By using iteration statements we are able to execute group of statements repeatedly or more number of times.

1) For 2) while 3) do-while

for syntax:-

```
for (initialization ;condition ;increment/decrement )
{       Body;
}
```

Flow of execution in for loop:-



The above process is repeated until the condition is false. If the condition is false the loop is stopped.

Initialization part:-

1) Initialization part it is possible to take the singe initialization it is not possible to take the more than one initialization.

With out for loop

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

By using for loop

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

***Initialization*:-**

Ex1: Inside the for loop initialization part is optional.

```
class Test
{       public static void main(String[ ] args)
        {       int i=0;
                for (;i<10;i++)
                {       System.out.println("gangamma");
                }
        }
}
```

Ex 2:- Instead of initialization it is possible to take any number of System.out.println("babu") statements and each and every statement is separated by commas(,) .

```
class Test
```

```
{       public static void main(String[] args)
        {       int i=0;
                for (System.out.println("krishna");i<10;i++)
                {       System.out.println("gangamma");
                }
        }
}
```

Ex 3:- compilation error more than one initialization not possible.

```
class Test
{
        public static void main(String[] args)
        {
        for (int i=0,double j=10.8;i<10;i++)
                {
                System.out.println("gangamma"
                );
                }
        }
```

Ex :-declaring two variables possible.

```
class Test
{
        public static void main(String[] args)
        {
                for (int i=0,j=0;i<10;i++)
                {

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

***Conditional part*:-**

Ex 1:-inside for loop conditional part is optional if we are not providing condition at the time of compilation compiler will provide true value.

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

***Increment/decrement*:-**

Ex1:- Inside the for loop increment/decrement part is optional.

```
class Test
{       public static void main(String[ ] args)
        {       for (int i=0;i<10;)
                {    System.out.println("gangamma");
                }
        }
}
```

Ex 2:- Instead of increment/decrement it is possible to take the any number of SOP() that and each and every statement is separated by commas(,).

```
class Test
{       public static void main(String[] args)
        {       for(int i=0;i<10;System.out.println("manam"),System.out.println("krishna"))
                {       System.out.println("Ramaya");
                        i++;
                }
        }
}
```

Note : Inside the for loop each and every part is optional.
for(;;)----------→represent infinite loop because the condition is always true.

Example :-

```
class Test
{       static boolean foo(char ch)
        {       System.out.println(ch);
                return true;
        }
        public static void main(String[] args)
        {       int i=0;
                for (foo('A');foo('B')&&(i<2);foo('C'))
                {       i++;
                        foo('D');
                }
        }
};
```

Ex:- compiler is unable to identify the unreachable statement.

```
class Test
{
        public static void main(String[] args)
        {
                for (int i=1;i>0;i++)
                {
System.out.println("infinite times ratan");
                }
        System.out.println("rest of the code");
        }
}
```

ex:- compiler able to identify the unreachable Statement.

```
class Test
{
        public static void main(String[] args)
        {
                for (int i=1;true;i++)
                {

        System.out.println("ratan");
                }
        System.out.println("rest of the code");
        }
}
```

While loop:-

Syntax:-

```
while (condition)    //condition must be Boolean & mandatory.
{       body;
}
```

Ex :-

```
class Test
{       public static void main(String[ ] args)
        {       int i=0;
                while(i<10)
                {       System.out.println("ramaya");
                        i++;
                }
        }
}
```

Ex :- compilation error unreachable statement

```
class Test
{       public static void main(String[] args)
        {       int i=0;
                while(false)
                {       //unreachable statement
                        System.out.println("ramaya");
                        i++;
                }
        }
}
```

do-while:-

1) If we want to execute the loop body at least one time then we should go for do-while statement.
2) In the do-while first body will be executed then only condition will be checked.
3) In the do-while, the while must be ends with semicolon otherwise we are getting compilation error.
4) do is taking the body and while is taking the condition and the condition must be boolean condition.

Syntax:-

```
do
{
//body of loop
} while(condition);
```

Example :-

```
class Test
        public static void main(String[ ] args)
        {       int i=0;
                do
                {       System.out.println("ramaya");
```

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

Example :- unreachable statement

```
class Test
{       public static void main(String[ ] args)
        {       int i=0;
                do
                {       System.out.println("ramaya");
                }while (true);
                System.out.println("ramainfotech");//unreachable statement
        }
}
```

Example :-

```
class Test
{       public static void main(String[] args)
        {       int i=0;
                do
                {       System.out.println("ramaya");
                }while (false);
                System.out.println("ramainfotech");
        }
}
```

Transfer statements:-By using transfer statements we are able to transfer the flow of execution from one position to another position**.**

1. **break 2. Continue 3. Return 4. Try**

***break*:-** Break is used to stop the execution.

We are able to use the break statement only two places.

a. Inside the switch statement. b. Inside the loops.

if we are using any other place the compiler will generate compilation error message **" break outside switch or loop".**

Example :-*break means stop the execution come out of loop.*

```
class Test
{       public static void main(String[ ] args)
        {       for (int i=0;i<10;i++)
                {       if (i==5)
                        break;
                        System.out.println(i);
                }
        }}
```

Example :-if we are using break outside switch or loops the compiler will raise compilation error ***"break outside switch or loop"***

```
class Test
{       public static void main(String[ ] args)
        {       if (true)
                {       System.out.println("rao");
                        break;
                        System.out.println("naidu");
                }
        }};
```

Continue:-(skip the current iteration and it is continue the rest of the iterations normally)

```
class Test
{       public static void main(String[ ] args)
        {       for (int i=0;i<10;i++)
                {       if (i==5)
                        continue;
                        System.out.println(i);
                }
        }
}
```

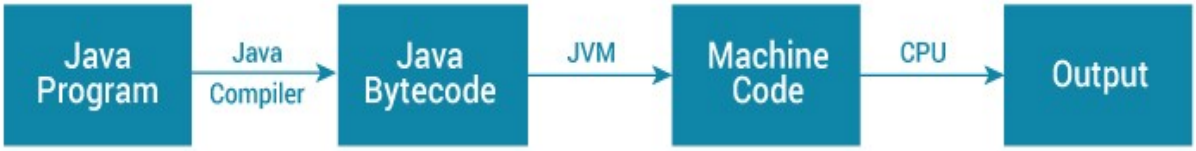
Java JDK, JRE and JVM

What is JVM?

JVM (Java Virtual Machine) is an abstract machine that enables your computer to run a Java program.

When you run the Java program, Java compiler first compiles your Java code to bytecode. Then, the JVM translates bytecode into native machine code (set of instructions that a computer's CPU executes directly).

Java is a platform-independent language. It's because when you write Java code, it's ultimately written for JVM but not your physical machine (computer). Since JVM executes the Java bytecode which is platform-independent, Java is platform-independent.



Working of Java Program

What is JRE?

JRE (Java Runtime Environment) is a software package that provides Java class libraries, Java Virtual Machine (JVM), and other components that are required to run Java applications.

JRE is the superset of JVM.

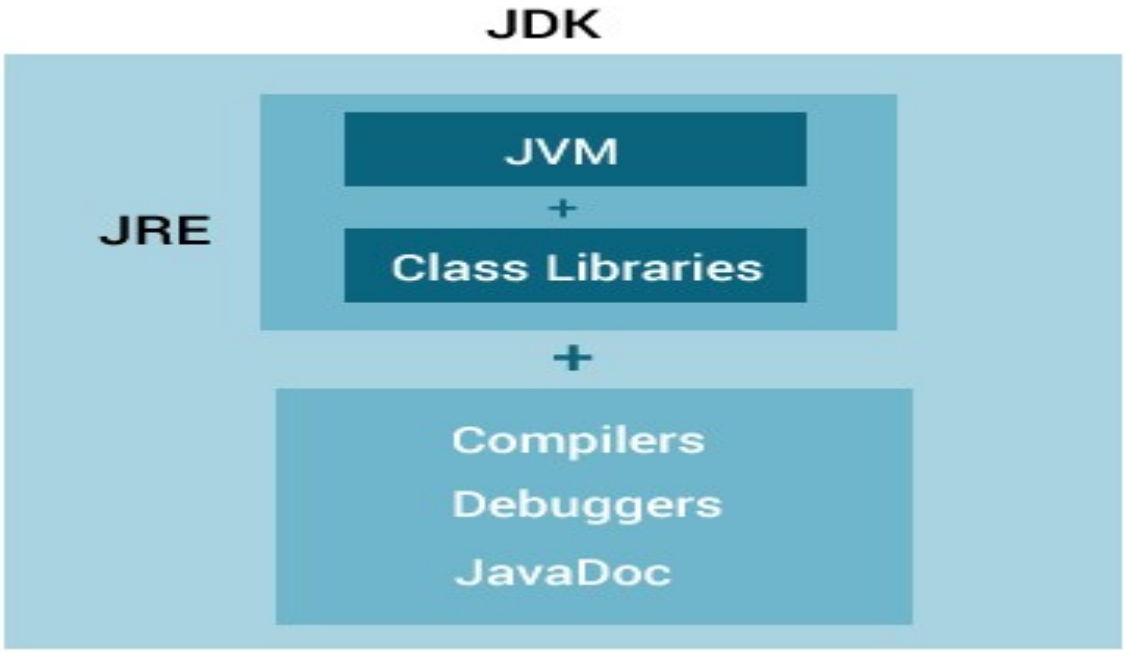
What is JDK?

JDK (Java Development Kit) is a software development kit required to develop applications in Java. When you download JDK, JRE is also downloaded with it.

In addition to JRE, JDK also contains a number of development tools (compilers, JavaDoc, Java Debugger, etc).

Java Development Kit

Relationship between JVM, JRE, and JDK.



Relationship between JVM, JRE, and JDK

Java Basic Input and Output

The simple ways to display output to users and take input from users in Java.

Java Output

In Java, you can simply use

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

System.out.print(); or

System.out.printf();
```

to send output to standard output (screen).

Here,

- `System` is a class
- `out` is a `public` `static` field: it accepts output data.

Let's take an example to output a line.

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

        System.out.println("Java programming is interesting.");
    }
}
```

Output:

```
Java programming is interesting.
```

Here, we have used the `println()` method to display the string.

Difference between println(), print() and printf()

- `print()` - It prints string inside the quotes.
- `println()` - It prints string inside the quotes similar like `print()` method. Then the cursor moves to the beginning of the next line.
- `printf()` - Tt provides string formatting (similar to printf in C/C++ programming).

Example: print() and println()

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

        System.out.println("1. println ");
        System.out.println("2. println ");

        System.out.print("1. print ");
        System.out.print("2. print");
    }
}
```

Output:

```
1. println
2. println
1. print 2. print
```

Example: Printing Variables and Literals

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

        Double number = -10.6;

        System.out.println(5);
        System.out.println(number);
    }
}
```

When you run the program, the output will be:

```
5
-10.6
```

Here, we can see that we have not used the quotation marks. It is because to display integers, variables and so on, we don't use quotation marks.

Example: Print Concatenated Strings

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

        Double number = -10.6;

        System.out.println("I am " + "awesome.");
        System.out.println("Number = " + number);
    }
}
```

Output:
```
I am awesome.
Number = -10.6
```

Here, we have used the + operator to concatenate (join) the two strings: *"I am "* and *"awesome."*.

And also, the line,

```
System.out.println("Number = " + number);
```

Here, first the value of variable *number* is evaluated. Then, the value is concatenated to the string: *"Number = "*.

Java Input

Java provides different ways to get input from the user. Here we will learn to get input from user using the object of `Scanner` class.

In order to use the object of `Scanner`, we need to import `java.util.Scanner` package.

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

Then, we need to create an object of the `Scanner` class. We can use the object to take input from the user.

```
// create an object of Scanner
Scanner input = new Scanner(System.in);

// take input from the user
int number = input.nextInt();
```

Example: Get Integer Input From the User

```java
import java.util.Scanner;

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

        Scanner input = new Scanner(System.in);

        System.out.print("Enter an integer: ");
        int number = input.nextInt();
        System.out.println("You entered " + number);

        // closing the scanner object
        input.close();
    }
}
```

Output:

```
Enter an integer: 23
You entered 23
```

In the above example, we have created an object named *input* of the `Scanner` class. We then call the `nextInt()` method of the `Scanner` class to get an integer input from the user.

Similarly, we can use `nextLong()`, `nextFloat()`, `nextDouble()`, and `next()` methods to get `long`, `float`, `double`, and `string` input respectively from the user.

Note: We have used the `close()` method to close the object. It is recommended to close the scanner object once the input is taken.

Example: Get float, double and String Input

```
import java.util.Scanner;

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

        Scanner input = new Scanner(System.in);

        // Getting float input
        System.out.print("Enter float: ");
        float myFloat = input.nextFloat();
        System.out.println("Float entered = " + myFloat);

        // Getting double input
        System.out.print("Enter double: ");
        double myDouble = input.nextDouble();
        System.out.println("Double entered = " + myDouble);

        // Getting String input
        System.out.print("Enter text: ");
        String myString = input.next();
        System.out.println("Text entered = " + myString);
    }
}
```

Output:

```
Enter float: 2.343
Float entered = 2.343
Enter double: -23.4
Double entered = -23.4
Enter text: Hey!
Text entered = Hey!
```

Scope and lifetime of Variables in Java

In programming, **scope of variable** defines how a specific variable is accessible within the program or across classes. In this section, we will discuss the **scope of variables in Java**.

Scope of a Variable

In programming, a variable can be declared and defined inside a class, method, or block. It defines the scope of the variable i.e. the visibility or accessibility of a variable. Variable declared inside a block or method are not visible to outside. If we try to do so, we will get a compilation error. Note that the scope of a variable can be nested.

- We can declare variables anywhere in the program but it has limited scope.
- A variable can be a parameter of a method or constructor.
- A variable can be defined and declared inside the body of a method and constructor.
- It can also be defined inside blocks and loops.
- Variable declared inside main() function cannot be accessed outside the main() function

Variable Type	Scope	Lifetime
Instance variable	Troughout the class except in static methods	Until the object is available in the memory
Class variable	Troughout the class	Until the end of the program
Local variable	Within the block in which it is declared	Until the control leaves the block in which it is declared

Types of Variables

Division 1 : Based on the type of value represented by a variable all variables are divided into 2 types. They are:

1. Primitive variables
2. Reference variables

Primitive variables:

Primitive variables can be used to represent primitive values.

Example:

```
int x=10;
```

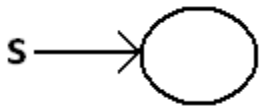
Reference variables:

Reference variables can be used to refer objects.

Example:

```
Student s=new Student();
```

Diagram:



Division 2 : Based on the behaviour and position of declaration all variables are divided into the following 3 types.

1. Instance variables
2. Static variables
3. Local variables

1. Instance variables:

- If the value of a variable is varied from object to object such type of variables are called instance variables.
- For every object a separate copy of instance variables will be created.
- Instance variables will be created at the time of object creation and destroyed at the time of object destruction hence the scope of instance variables is exactly same as scope of objects.
- Instance variables will be stored on the heap as the part of object.
- Instance variables should be declared with in the class directly but outside of any method or block or constructor.
- Instance variables can be accessed directly from Instance area. But cannot be accessed directly from static area.
- But by using object reference we can access instance variables from static area.

Example:

```
class Test
{
      int i=10;
      public static void main(String[] args)
      {
            //System.out.println(i);
```

```
		//C.E:non-static variable i cannot be referenced from a
	static context(invalid)
		Test t=new Test();
		System.out.println(t.i);//10(valid)
		t.methodOne();
	}
	public void methodOne()
	{
		System.out.println(i);//10(valid)
	}
}
```

For the instance variables it is not required to perform initialization JVM will always provide default values.

Example:

```
class Test
{
	boolean b;
	public static void main(String[] args)
	{
		Test t=new Test();
		System.out.println(t.b);//false
	}
}
```

Instance variables also known as object level variables or attributes.

2. Static variables:

- If the value of a variable is not varied from object to object such type of variables is not recommended to declare as instance variables. We have to declare such type of variables at class level by using static modifier.
- In the case of instance variables for every object a separate copy will be created but in the case of static variables for entire class only one copy will be created and shared by every object of that class.
- Static variables will be crated at the time of class loading and destroyed at the time of class unloading hence the scope of the static variable is exactly same as the scope of the .class file.
- Static variables will be stored in method area. Static variables should be declared with in the class directly but outside of any method or block or constructor. Static variables can be accessed from both instance and static areas directly.
- We can access static variables either by class name or by object reference but usage of class name is recommended.
- But within the same class it is not required to use class name we can access directly.

java TEST

1. Start JVM.
2. Create and start Main Thread by JVM.
3. Locate(find) Test.class by main Thread.
4. Load Test.class by main Thread. // static variable creation
5. Execution of main() method.

6. Unload Test.class // static variable destruction
7. Terminate main Thread.
8. Shutdown JVM.

Example:

```
class Test
{
	static int i=10;
	public static void main(String[] args)
	{
		Test t=new Test();
		System.out.println(t.i);//10
		System.out.println(Test.i);//10
		System.out.println(i);//10
	}
}
```

For the static variables it is not required to perform initialization explicitly, JVM will always provide default values.

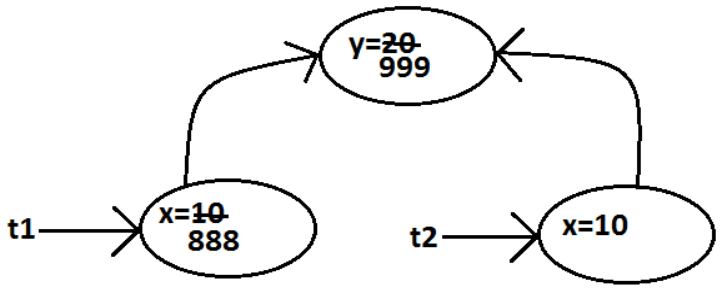
Example:

```
class Test
{
	static String s;
	public static void main(String[] args)
	{
		System.out.println(s);//null
	}
}
```

Example:

```
class Test
{
	int x=10;
	static int y=20;
	public static void main(String[] args)
	{
		Test t1=new Test();
		t1.x=888;
		t1.y=999;
		Test t2=new Test();
		System.out.println(t2.x+"----"+t2.y);//10----999
	}
}
```

Diagram:



Static variables also known as class level variables or fields.

3. Local variables:

- Some times to meet temporary requirements of the programmer we can declare variables inside a method or block or constructors such type of variables are called local variables or automatic variables or temporary variables or stack variables.
- Local variables will be stored inside stack.
- The local variables will be created as part of the block execution in which it is declared and destroyed once that block execution completes. Hence the scope of the local variables is exactly same as scope of the block in which we declared.

Example 1:

```
class Test
{
      public static void main(String[] args)
      {
            int i=0;
            for(int j=0;j<3;j++)
            {
                  i=i+j;
            }

                  System.out.println(i+"----"+j);
```

C.E

```
javac Test.java
Test.java:10: cannot find symbol
symbol  : variable j
location: class Test
```

```
      }
}
```

- The local variables will be stored on the stack.
- For the local variables JVM won't provide any default values compulsory we should perform initialization explicitly before using that variable.

```
class Test
{
    public static void main(String[] args)
    {
        int x;
        System.out.println("hello");//hello
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        int x;
      System.out.println(x);//C.E:variable x might not have been initialized
    }
}
```

Example:

```
class Test
{

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

```
            int x;
            if(args.length>0)
            {
                  x=10;
            }
            System.out.println(x);
            //C.E:variable x might not have been initialized
      }
}
```

Example:

```
class Test
{
      public static void main(String[] args)
      {
            int x;
            if(args.length>0)
            {
                  x=10;
            }
            else
            {
                  x=20;
            }
            System.out.println(x);
      }
}
```

Output:

java Test x
10
java Test x y
10
java Test
20

- It is never recommended to perform initialization for the local variables inside logical blocks because there is no guarantee of executing that block always at runtime.
- It is highly recommended to perform initialization for the local variables at the time of declaration at least with default values.

Note: The only applicable modifier for local variables is final. If we are using any other modifier we will get compile time error.

Example:

```
class Test
{
public static void main(String[] args)
{
      public int x=10; //(invalid)
      private int x=10; //(invalid)
      protected int x=10; //(invalid) C.E: illegal start of expression
      static int x=10; //(invalid)
      volatile int x=10; //(invalid)
      transient int x=10; //(invalid)
```

```
        final int x=10;//(valid)
}
}
```

Conclusions:

1. For the static and instance variables it is not required to perform initialization explicitly JVM will provide default values. But for the local variables JVM won't provide any default values compulsory we should perform initialization explicitly before using that variable.
2. For every object a separate copy of instance variable will be created whereas for entire class a single copy of static variable will be created. For every Thread a separate copy of local variable will be created.
3. Instance and static variables can be accessed by multiple Threads simultaneously and hence these are not Thread safe but local variables can be accessed by only one Thread at a time and hence local variables are Thread safe.
4. If we are not declaring any modifier explicitly then it means default modifier but this rule is applicable only for static and instance variables but not local variable.

Differences between local variables, instance variables, and static variables:

Characteristic	Local variable	instance variable	static variables
where declared	inside method or Constructor or block.	inside the class outside Of methods	inside the class outside Of methods
Use	within the method	inside the class all methods and constructors.	inside the class all methods and constructors.
When memory allocated	When method starts	when objectcreated	when .class file loading
When memory destroyed	when method ends.	When object destroyed	when .class unloading.
Initial value s	none, must initialize the value before first use.	default values are Assigned by JVM	default values are Assigned by JVM
Relation with Object	no way related to object.	for every object one copy Of instance variable created It means memory.	for all objects one copy is created. Single memory
Accessing	directly possible.	By using object name. Test t = new Test(); System.out.println(t.a);	by using class name. System.out.println(Test .a);
Memory	stored in stack memory	Stored in heap memory	non-heap memory.

Arrays

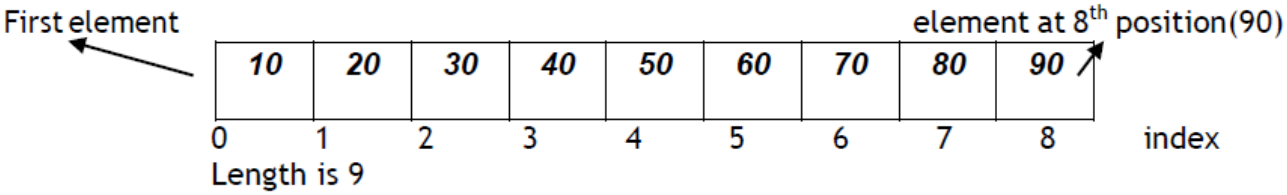
INTRODUCTION:

i. An array is an indexed collection of fixed number of homogenous data elements.

ii. The main advantage of arrays is we can represent multiple values with the same name so that readability of the code will be improved.

iii. But the main disadvantage of arrays is: fixed in size. It means once we created an array, there is no chance of increasing or decreasing the size based on our requirement that is to use arrays concept compulsory, we should know the size in advance which may not possible always.

iv. We can resolve this problem by using collections.



ARRAY DECLARATIONS:

Single dimensional array declaration:

int[ ] a;//recommended to use because name is clearly separated from the type

int [ ]a;

int a[ ];

At the time of declaration, we can't specify the size otherwise we will get compile time error

Example:

int[ ] a; //valid

int[5] a; //invalid

Two dimensional array declaration:

Example:

```
int[ ][ ]   a;
int    [ ][ ]a;
int   a[ ][ ];
int[ ]   [ ]a;
int[ ]   a[ ];
int   [ ]a[ ];
```

All are valid. (6 ways)

Three dimensional array declaration:

Example:

```
int[ ][ ][ ]      a;
int      [ ][ ][ ]a;
int      a[ ][ ][ ];
int[ ]      [ ][ ]a;
int[ ]     a[ ][ ];
int[ ]      [ ]a[ ];
int[ ][ ]     a[ ];
int[ ][ ]     [ ]a;
int     [ ]a[ ][ ];
int      [ ][ ]a[ ];
```

All are valid(10 ways)

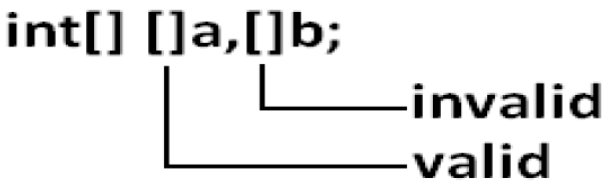
which of the following declarations are valid?

1) int[ ] a1,b1; (valid)
2) int[ ] a2[ ],b2; (valid)
3) int[ ] [ ]a3,b3; (valid)
4) int[ ] a,[ ]b; (invalid)

Note:

- If we want to specify the dimension before the variable that rule is applicable only for the 1st variable
- 2nd variable onwards we cant apply in the same declaration.

Example:

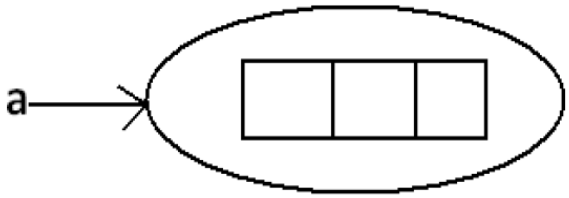


ARRAY CREATION:

Every array in java is an object hence we can create by using new operator.

Ex:

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



For every array type corresponding classes are available but these classes are part of java language and not available to the programmer level.

Array Type	Corresponding class name
int[ ]	[I
int[ ][ ]	[[I
Double[ ]	[D

RULE 1: At the time of array creation, compulsory we should specify the size otherwise we will get compile time error.

Example:

```
int[ ]   a=new  int[3];
int[ ]   a=new  int[ ];     //array dimension missing
```

RULE 2: It is legal to have an array with **size zero** in java.

Example:

```
int[ ]   a=new  int[0];
System.out.println(a.length);//0
```

RULE 3: If we are taking array size with –ve int value then we will get runtime exception saying NegativeArraySizeException

Example:

```
int[ ]   a=new  int[-3];//NegativeArraySizeException
```

RULE 4: The allowed datatypes to specify array size are **byte,short,char,int.**

By mistake if we are using any other type we will get compile time error.

Example:

```
int[ ]   a=new  int['a']; //valid
byte     b=10;
int[ ]   a=new int[b];//valid
short    s=20;
int[ ]   a=new  int[s]; //valid
int[ ]   a=new  int[10.5];//possible loss of precision  //invalid
```

RULE 5: The maximum allowed array size in java is maximum value of int size [2147483647].

Example:

```
int[ ]  a1=new  int[2147483647];//valid
int[ ]  a2=new  int[2147483648];// integer number too large:2147483648(invalid)
```

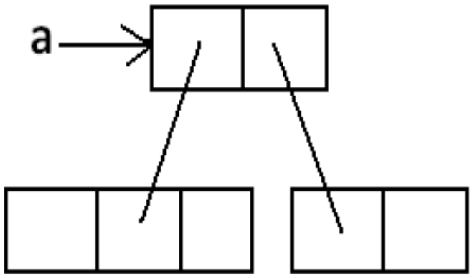
MULTIDIMENSIONAL ARRAY CREATION:

- In java, multidimensional arrays are implemented as array of arrays approach but not matrix form.
- The main advantage of this approach is to improve memory utilization.

Example:

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

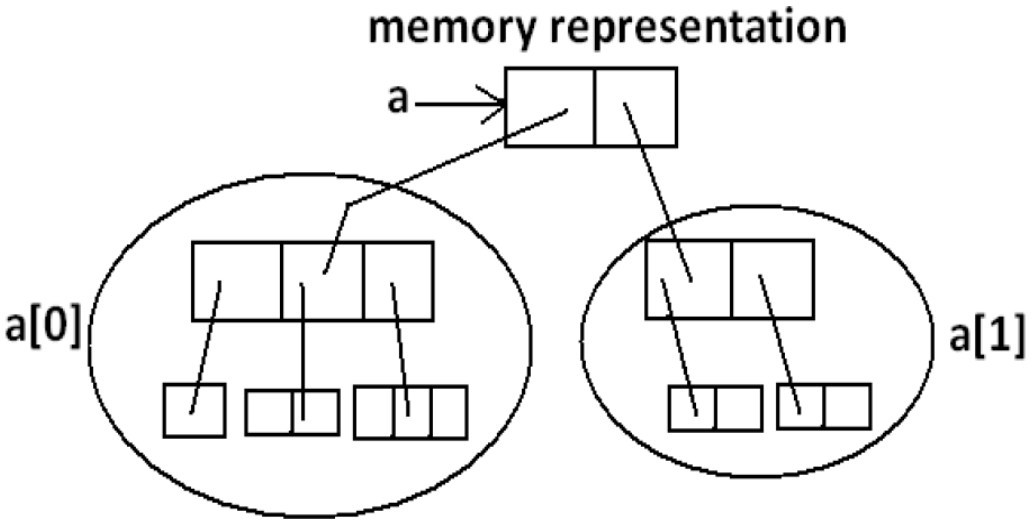
Diagram of memory representation:



Example 2:

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

Diagram of memory representation:



Which of the following declarations are valid?

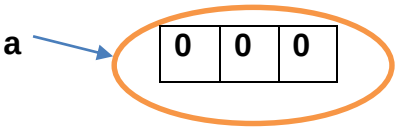
1) `int[ ]  a=new  int[  ];` //array dimension is missing (invalid)
2) `int[ ][ ]  a=new  int[3][4];` (valid)
3) `int[ ][ ]  a=new  int[3][ ];` (valid)
4) `int[ ][ ]  a=new  int[ ][4];` (invalid)
5) `int[ ][ ][ ]  a=new  int[3][4][5];` (valid)
6) `int[ ][ ][ ]  a=new  int[3][ ][5];` (invalid)

ARRAY INITIALIZATION:

Whenever we are creating an array every element is initialized with default value automatically.

Example1:

```
int[ ]  a= new  int[3];
System.out.println(a);    //[I@3e25a25
System.out.println(a[0]);//0
```



Note:

Whenever we are trying to print any object reference internally toString() method will be executed which is implemented by default to return the following.

classname@hexadecimalstringrepresentationofhashcode. Eg: [I @ 3e25a25

Example 2:

```
int[ ][ ]    a=new int[2][3];       //here 2 is base size
```

System.out.println(a);//[[I@3e25a5

System.out.println(a[0]);//[I@19821f

System.out.println(a[0][0]);//0

Example 3:

int[ ][ ] a=new int[2][];

System.out.println(a);//[[I@3e25a5

System.out.println(a[0]);//null

System.out.println(a[0][0]);//R.E:NullPointerException

Once we created an array all its elements by default initialized with default values. If we are not satisfied with those default values then we can replace with our customized values.

Example:

int[] a=new int[4];

a[0]=10;

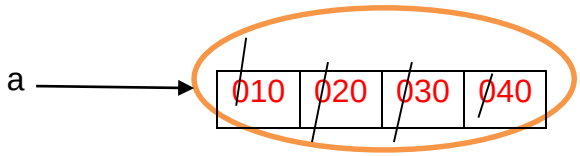
a[1]=20;

a[2]=30;

a[3]=40;

a[4]=50;//R.E:ArrayIndexOutOfBoundsException: 4

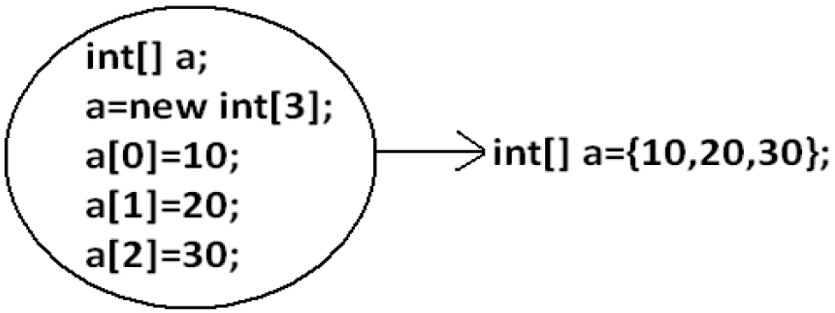
a[-4]=60;//R.E:ArrayIndexOutOfBoundsException: -4



Note: if we are trying to access array element with out of range index we will get Runtime Exception saying ArrayIndexOutOfBoundsException.

DECLARATION, CONSTRUCTION AND INITIALIZATION OF AN ARRAY IN A SINGLE LINE:

We can perform declaration, construction and initialization of an array in a single line.



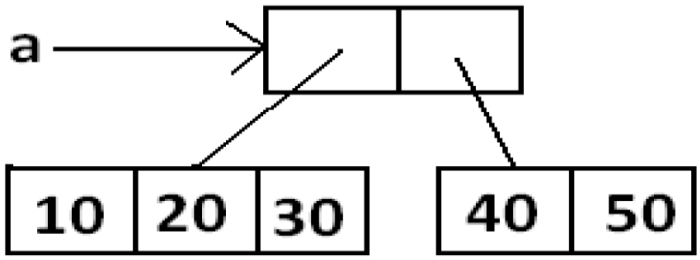
char[ ] ch={'a','e','i','o','u'};(valid)

String[ ] s={"balayya","venki","nag","chiru"};(valid)

We can extend this short cut even for multi dimensional arrays also.

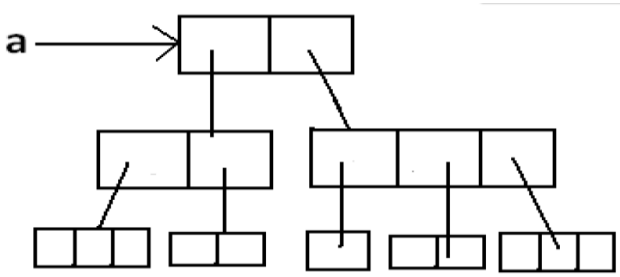
Example:

```
int[ ][ ] a={{10,20,30},{40,50}};
```



Example:

```
int[][][] a={{{10,20,30},{40,50}},{{60},{70,80},{90,100,110}}};
```



int[ ][ ][ ] a={{{10,20,30},{40,50}},{{60},{70,80},{90,100,110}}};

System.out.println(a[0][1][1]);//50(valid)

System.out.println(a[1][0][2]);//R.E:ArrayIndexOutOfBoundsException: 2(invalid)

System.out.println(a[1][2][1]);//100(valid)

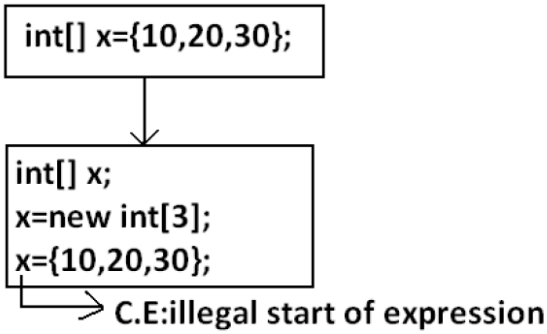
System.out.println(a[1][2][2]);//110(valid)

System.out.println(a[2][1][0]);//R.E:ArrayIndexOutOfBoundsException:2(invalid)

System.out.println(a[1][1][1]);//80(valid)

- If we want to use this short cut compulsory we should perform declaration, construction and initialization in a single line.
- If we are trying to divide into multiple lines then we will get compile time error.

Example:



length Vs length( ):

length:

1. It is the final variable applicable only for arrays.
2. It represents the size of the array.

Example:

```
int[] x=new int[3];
System.out.println(x.length());//C.E: cannot find symbol
System.out.println(x.length);//3
```

length( ) method:

1. It is a final method applicable for String objects.
2. It returns the no of characters present in the String.

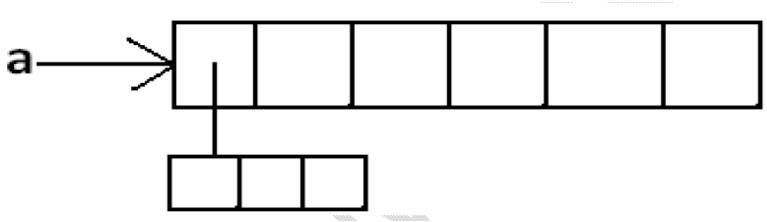
Example:

```
String s="bhaskar";
System.out.println(s.length);//C.E:cannot find symbol
System.out.println(s.length());//7
```

In multidimensional arrays length variable represents only base size but not total size.

Example:

```
int[][] a=new int[6][3];
System.out.println(a.length);//6
System.out.println(a[0].length);//3
```



- length variable applicable only for arrays where as length()method is applicable for String objects.
- There is no direct way to find total size of multi dimentional array but indirectly we can find as follows

  x[0].length +x[1].length + x[2].length + ...

PROGRAMS

Eg 1 :- Taking array elements from dynamic input by using scanner class.

```
import java.util.*;
class Test
{       public static void main(String[ ] args)
        {       int[ ] a=new int[5];
                Scanner s=new Scanner(System.in);
                System.out.println("enter values");
                for (int i=0;i<a.length;i++)
                {       System.out.println("enter "+i+"value");
                        a[i]=s.nextInt();
                }
                for (int a1:a)
                {       System.out.println(a1);
                }
        }
}
```

Eg 2:-Find the sum of the array elements.

```
class Test
{       public static void main(String[] args)
       {        int[] a={10,20,30,40};
                int sum=0;
                for (int a1:a)
                {
                        sum=sum+a1;
                }
                System.out.println("Array Element sum is="+sum);
        }
}
```

Eg 3:-To get the class name of the array:-

```
class Test
{       public static void main(String[] args)
        {       int[] a={10,20,30};
                System.out.println(a.getClass().getName());
        }
}
```

Eg :Finding minimum & maximum element of the array:-

```
class Test
{
	public static void main(String[] args)
	{	int[]a=new int[]{10,20,5,70,4};
		for (int a1:a)
		{	System.out.println(a1);
		}
		//minimum element of the Array
		int min=a[0];
		for (int i=1;i<a.length;i++)
		{	if (min>a[i])
			{	min=a[i];
			}
		}
		System.out.println("minimum value is ="+min);
		//maximum element of the Array
		int max=a[0];
		for (int i=1;i<a.length;i++)
		{	if (max<a[i])
			{	max=a[i];
			}
		}
		System.out.println("maximum value is ="+max);
	}
}
```

Eg:- Finding null index values.

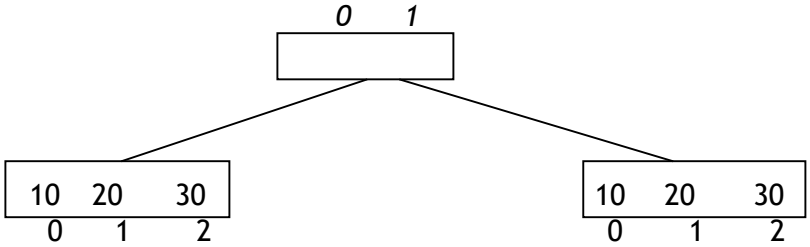
```
class Test
{	public static void main(String[] args)
	{	String[]str= new String[5];
		str[0]="ratan";
		str[1]="anu";
		str[2]=null;
		str[3]="sravya";
		str[4]=null;
		for (int i=0;i<str.length;i++)
		{	if (str[i]==null)
			{	System.out.println(i);
			}
		}
	}
}
```

DECLARATION OF MULTI-DIMENSIONAL ARRAY:-

```
int[ ][ ]  a;
int   [ ][ ]a;
int  a[ ][ ];
int [ ]a[ ];
```

Example :-

```
class Test
{       public static void main(String[] args)
        {       int[][]  a={{10,20,30},{40,50,60}};
                System.out.println(a[0][0]);//10
                System.out.println(a[1][0]);//40
                System.out.println(a[1][1]);//50
        }
}
```



a[0][0]------→10 a[0][1]------→20 a[0][2]-----→30 ,
a[1][0]-----→40 a[1][1]-----→50 a[1][2]-----→60

Example:-

```
class Test
{       public static void main(String[] args)
        {       String[][]str={{"A.","B.","C."},{"ratan","ratan","ratan"}};
                System.out.println(str[0][0]+str[1][0]);
                System.out.println(str[0][1]+str[1][1]);
                System.out.println(str[0][2]+str[1][2]);
        }
}
```

Example: fibonacci series

```
importjava.util.Scanner;
class Test
{       public static void main(String[ ] args)
        {       System.out.println("enter start series of fibonacci");
                int  x = new Scanner(System.in).nextInt();
                int[] feb = new
                int[x];
                feb[0]=0;
                feb[1]=1;
                for (int i=2;i<x;i++)
                {       feb[i]=feb[i-1]+feb[i-2];
                }
                //print the data
                for (int feb1 : feb)
                {
                        System.out.print(" "+feb1);
                }
        }
}
```

PRE-INCREMENT & POST-INCREMENT :-

PRE-INCREMENT:- it increases the value by 1 then it will execute statement.

POST INCREMENT: it executes the statement then it will increase value by 1

```
class Test
{       public static void main(String[] args)
        {       //post increment
                int a=10;
                System.out.println(a);      //10
                System.out.println(a++);  //10
                System.out.println(a); //11
               //pre increment
               int b=20;
               System.out.println(b);       //20
               System.out.println(++b);    //21
               System.out.println(b); //21
               System.out.println(a++ + ++a + a++ + ++a); //11 13 13  15
        }
}
```

PRE-DECREMENT & POST-DECREMENT:

PRE-DECREMENT:- it decrements the value by 1 then it will execute statement.

POST-DECREMENT: It executes the statement then it will increase value by 1

```
Class Test
{
        public static void main(String args[ ])
        {
                //post decrement
                int a=10;
                System.out.println(a);      //10
                System.out.println(a--);    //10
                System.out.println(a); //9
                //pre decrement
                int b=20;
                System.out.println(b);      //20
                System.out.println(b);      //19
                System.out.println(b);      //19
                System.out.println(a-- + --a + a-- + --a);//9    7      7      5

         }
}
```

Java Type Casting

Java Type Casting

Type casting is when you assign a value of one primitive data type to another type.

In Java, there are two types of casting:

- **Widening Casting** (automatically) - converting a smaller type to a larger type size
  `byte` -> `short` -> `char` -> `int` -> `long` -> `float` -> `double`

- **Narrowing Casting** (manually) - converting a larger type to a smaller size type
  `double` -> `float` -> `long` -> `int` -> `char` -> `short` -> `byte`

Widening Casting

Widening casting is done automatically when passing a smaller size type to a larger size type:

Example

```
public class MyClass {
  public static void main(String[] args) {
    int myInt = 9;
    double myDouble = myInt; // Automatic casting: int to double

    System.out.println(myInt);      // Outputs 9
    System.out.println(myDouble);   // Outputs 9.0
  }
}
```

Narrowing Casting

Narrowing casting must be done manually by placing the type in parentheses in front of the value:

Example

```
public class MyClass {
  public static void main(String[] args) {
    double myDouble = 9.78;
    int myInt = (int) myDouble; // Manual casting: double to int

    System.out.println(myDouble);   // Outputs 9.78
    System.out.println(myInt);      // Outputs 9
  }
}
```

CONSTRUCTORS

Agenda:

Constructors

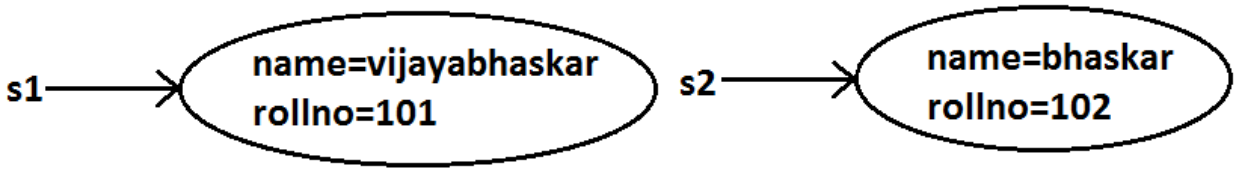
- **Constructor vs instance block**
- **Rules to write constructors**
- **Default constructor**
- **Prototype of default constructor**
- **super( ) vs this( )**
- **Overloaded constructors**

Constructors :

1. Object creation is not enough compulsory we should perform initialization then only the object is in a position to provide the response properly.
2. Whenever we are creating an object some piece of the code will be executed automatically to perform initialization of an object this piece of the code is nothing but constructor.
3. Hence the main objective of constructor is to perform initialization of an object.

```
Example:
class Student
{
        String name;
        int rollno;
        Student(String name,int rollno)  //Constructor
        {
                this.name=name;
                this.rollno=rollno;
        }
        public static void main(String[] args)
        {
                Student s1=new Student("vijayabhaskar",101);
                Student s2=new Student("bhaskar",102);
                System.out.println("NAME:" + s1.name + "\t" + "ROLLNO:" + s1.rollno);
                System.out.println("NAME:" + s2.name + "\t" + "ROLLNO:" + s2.rollno);
        }
}
```

Diagram:



Constructor Vs instance block:

1. Both instance block and constructor will be executed automatically for every object creation but instance block 1st followed by constructor.
2. The main objective of constructor is to perform initialization of an object.
3. Other than initialization if we want to perform any activity for every object creation we have to define that activity inside instance block.
4. Both concepts having different purposes hence replacing one concept with another concept is not possible.
5. Constructor can take arguments but instance block can't take any arguments hence we can't replace constructor concept with instance block.
6. Similarly we can't replace instance block purpose with constructor.

Demo program to track no of objects created for a class:

```
class Test
{
	static int count=0;
	{
		count++;	//instance block
	}
	Test()
	{}
	Test(int i)
	{}
	public static void main(String[] args)
	{
		Test t1=new Test();
		Test t2=new Test(10);
		Test t3=new Test();
		System.out.println(count);//3
	}
}
```

Rules to write constructors:

1. Name of the constructor and name of the class must be same.

2. Return type concept is not applicable for constructor even void also by mistake if we are declaring the return type for the constructor we won't get any compile time error and runtime error compiler simply treats it as a method.

```
Example:
class Test
{
  void Test() //it is not a constructor and it is a method
  {}
}
```

3. It is legal (but stupid) to have a method whose name is exactly same as class name.
4. The only applicable modifiers for the constructors are public, default, private, protected.
5. If we are using any other modifier we will get compile time error.

```
Example:
class Test
{
       static Test()
       {}
}
Output:
Modifier static not allowed here
```

Default constructor:

1. For every class in java including abstract classes also constructor concept is applicable.
2. If we are not writing at least one constructor then compiler will generate default constructor.
3. If we are writing at least one constructor then compiler won't generate any default constructor. Hence every class contains either compiler generated constructor (or) programmer written constructor but not both simultaneously.

Prototype of default constructor:

1. It is always no argument constructor.
2. The access modifier of the default constructor is same as class modifier. (This rule is applicable only for public and default).
3. Default constructor contains only one line. super( ); it is a no argument call to super class constructor.

Programmers code	Compiler generated code

class Test { }	**class Test {
	Test()
	{
	super();
	}**

	}

public class Test { }	public class Test {
	public Test()
	{
	super();
	}
	
	}

class Test
{
void Test(){}
 	
}
	class Test
	{
	Test()
	{
	super();
	}
	void Test()
	{}
	
	}

class Test
{
Test(int i)
{}
 	
}
	class Test
	{
	Test(int i)
	{
	super();
	}
	
	}

class Test
{
Test()
{
super();
}
 	
}
	class Test
	{
	Test()
	{
	super();
	}
	
	}

class Test
{
Test(int i)
{
this();
}
Test()
{}
 	
}
	class Test
	{
	Test(int i)
	{
	this();
	}
	Test()
	{
	super();
	}
	}

super( ) vs this( ):

The 1st line inside every constructor should be either super( ) or this( ) if we are not writing anything compiler will always generate super( ).

Case 1: We have to take super() (or) this() only in the 1st line of constructor. If we are taking anywhere else we will get compile time error.

```
Example:
class Test
{
	Test()
	{
		System.out.println("constructor");
		super();
	}
}
Output:
Compile time error.
Call to super must be first statement in constructor
```

Case 2: We can use either super( ) (or) this( ) but not both simultaneously.

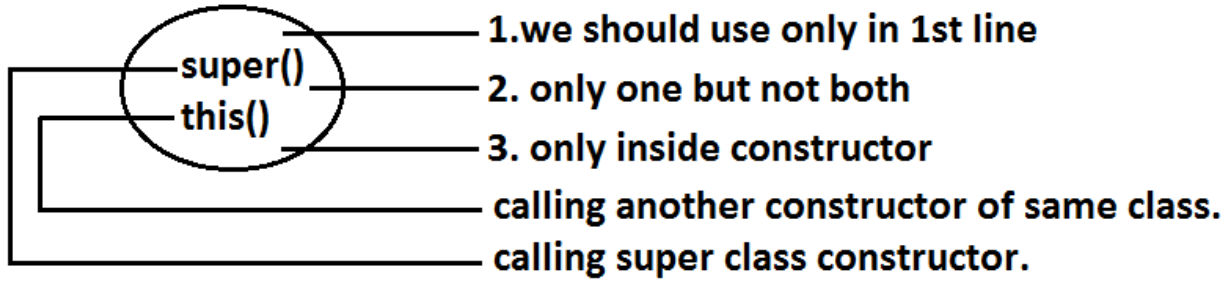
```
Example:
class Test
{
	Test()
	{
		super();
		this();
	}
}
Output:
Compile time error.
Call to this must be first statement in constructor
```

Case 3: We can use super( ) (or) this( ) only inside constructor. If we are using anywhere else we will get compile time error.

```
Example:
class Test
{
	public void methodOne()
	{
		super();
	}
}
Output:
Compile time error.
Call to super must be first statement in constructor
```

That is we can call a constructor directly from another constructor only.

Diagram:



Example:

super(), this()	super, this
These are constructors calls.	These are keywords
We can use these to invoke super class & current constructors directly	We can use refers parent class and current class instance members.
We should use only inside constructors as first line, if we are using outside of constructor we will get compile time error.	We can use anywhere (i.e., instance area) except static area , other wise we will get compile time error .

```
Example:
class Test
{
	public static void main(String[] args)
	{
		System.out.println(super.hashCode());
	}
}
Output:
Compile time error.
Non-static variable super cannot be referenced from a static context.
```

Overloaded constructors :

A class can contain more than one constructor and all these constructors having the same name but different arguments and hence these constructors are considered as overloaded constructors.

```
Example:
class Test
{
	Test(double d)
	{
		System.out.println("double-argument constructor");
	}
	Test(int i)
	{
		this(10.5);
		System.out.println("int-argument constructor");
	}
```

```
        Test()
        {
                this(10);
                System.out.println("no-argument constructor");
        }
        public static void main(String[] args)
        {
        Test t1=new Test();  //no-argument constructor/int-argument
                             //constructor/double-argument constructor
        Test t2=new Test(10);
                //int-argument constructor/double-argument constructor
        Test t3=new Test(10.5);//double-argument constructor
        }
}
```

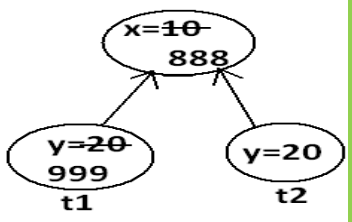
- Parent class constructor by default won't available to the Child. Hence Inheritance concept is not applicable for constructors and hence overriding concept also not applicable to the constructors. But constructors can be overloaded.
- We can take constructor in any java class including abstract class also but we can't take constructor inside interface.

Static modifier:

- Static is the modifier applicable for methods, variables and blocks.
- We can't declare a class with static but inner classes can be declaring as the static.
- In the case of instance variables, for every object a separate copy will be created but in the case of static variables, a single copy will be created at class level and shared by all objects of that class.

Example:

```
class Test{
static int x=10;
int y=20;
public static void main(String args[]){
Test t1=new Test();
t1.x=888;
t1.y=999;
Test t2=new Test();
System.out.println(t2.x+"....."+t2.y);
}
}
```



Output:

```
D:\Java>javac Test.java
D:\Java>java Test
888.....20
```

- Instance variables can be accessed only from instance area directly and we can't access from static area directly.
- But static variables can be accessed from both instance and static areas directly.

```
1)      int x=10;

2)      static int x=10;

3)      public void m1( )
        {
          System.out.println(x);
        }

4)      public static void m1( )
        {
          System.out.println(x);
        }
```

Which are the following declarations are allow within the same class simultaneously ?

a) 1 and 3:

Example:

```
class Test
{
        int x=10;
        public void m1( )
        {
                System.out.println(x);
        }
}
Output:
Compile successfully.
```

b) 1 and 4:

Example:

```
class Test
{
        int x=10;
        public static void methodOne()
        {
                System.out.println(x);
        }
}
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:5: non-static variable x cannot be referenced from a static context
System.out.println(x);
```

c) 2 and 3:

Example:

```
class Test
{
        static int x=10;
        public void methodOne()
        {
                System.out.println(x);
        }
}
Output:
Compile successfully.
```

d) 2 and 4 :

Example:

```
class Test
{
        static int x=10;
        public static void m1()
```

```
        {
                System.out.println(x);
        }
}
```
Output:
Compile successfully.

e) 1 and 2:

Example:

```
class Test
{
        int x=10;
        static int x=10;
}
```
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:4: x is already defined in Test
static int x=10;

f) 3 and 4:

Example:

```
class Test
{
        public void methodOne()
        {
                System.out.println(x);
        }
        public static void methodOne()
        {
                System.out.println(x);
        }
}
```
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:5: methodOne() is already defined in Test
public static void methodOne(){

For static methods implementation should be available but for abstract methods implementation is not available hence static abstract combination is illegal for methods.

case 1:

Overloading concept is applicable for static method including main method also. But JVM will always call **String[] args** main method .
The other overloaded method we have to call explicitly then it will be executed just like a normal method call .

Example:

```
class Test{
public static void main(String args[]){
System.out.println("String() method is called");
}
public static void main(int args[]){
System.out.println("int() method is called");
}
}
```

This method we have to call explicitly.

Output :
String() method is called

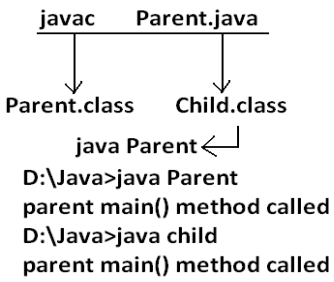
case 2:

Inheritance concept is applicable for static methods including main() method hence while executing child class, if the child doesn't contain main() method then the parent class main method will be executed.

Example:

```
class Parent
{
        public static void main(String args[])
        {
                System.out.println("parent main() method called");
        }
}
class child extends Parent
{
}
```

Output:

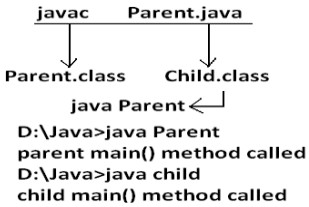


Example:

```
class Parent{
public static void main(String args[]){
System.out.println("parent main() method called");
}
}
class child extends Parent{
public static void main(String args[]){
System.out.println("child main() method called");
}
```

it is not overriding but method hiding.

Output:




```
D:\Java>java Parent
parent main() method called
D:\Java>java child
child main() method called
```

- It seems to be overriding concept is applicable for static methods but it is not overriding it is method hiding.

MODIFIERS

Modifiers:

Modifiers are used in the declaration of class members and local variables. These are summarized in the following tables.

Modifier	Meaning
`abstract`	The class cannot be instantiated.
`final`	The class cannot be extended.
`public`	Its members can be accessed from any other class.
`strictfp`	Floating point results will be platform independent.

Table 1.1 Modifiers for classes, interfaces, and enums

Modifier	Meaning
`private`	It is accessible only from within its own class.
`protected`	It is accessible only from within its own class and its extensions.
`public`	It is accessible from all classes.

Table 1.2 Constructor modifiers

Modifier	Meaning
`final`	It must be initialized and cannot be changed.
`private`	It is accessible only from within its own class.
`protected`	It is accessible only from within its own class and its extensions.
`public`	It is accessible from all classes.
`static`	The same storage is used for all instances of the class.
`transient`	It is not part of the persistent state of an object.
`volatile`	It may be modified by asynchronous threads.

Table 1.3 Field modifiers

Modifier	Meaning
`abstract`	Its body is absent; to be defined in a subclass.
`final`	It cannot be overridden in class extensions.
`native`	Its body is implemented in another programming language.
`private`	It is accessible only from within its own class.
`protected`	It is accessible only from within its own class and its extensions.
`public`	It is accessible from all classes.
`static`	It has no implicit argument.
`strictfp`	Its floating point results will be platform independent.
`synchronized`	It must be locked before it can be invoked by a thread.
`volatile`	It may be modified by asynchronous threads.

Table 1.4 Method modifiers

Modifier	Meaning
`final`	It must be initialized and cannot be changed.

Table 1.5 Local variable modifier

The three access modifiers, public, protected, and private, are used to specify where the declared member (class, field, constructor, or method) can be used. If none of these is specified, then the entity has *package access*, which means that it can be accessed from any class in the same package.

The modifier final has three different meanings, depending upon which kind of entity it modifies. If it modifies a class, final means that the class cannot be extended to a subclass. If it modifies a field or a local variable, it means that the variable must be initialized and cannot be changed, that is, it is a constant. If it modifies a method, it means that the method cannot be overridden in any subclass.

The modifier static means that the member can be accessed only as an agent of the class itself, as opposed to being bound to a specific object instantiated from the class.

A static method is also called a *class method*; a non static method is also called an *instance method*. The object to which an instance method is bound in an invocation is called its *implicit argument* for that invocation

ACCESS CONTROL

Controlling Access to Members of a Class

Access level modifiers determine whether other classes can use a particular field or invoke a particular method. There are two levels of access control:

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

A class may be declared with the modifier `public`, in which case that class is visible to all classes everywhere. If a class has no modifier (the default, also known as *package-private*), it is visible only within its own package (packages are named groups of related classes — you will learn about them in a later lesson.)

At the member level, you can also use the `public` modifier or no modifier (*package-private*) just as with top-level classes, and with the same meaning. For members, there are two additional access modifiers: `private` and `protected`. The `private` modifier specifies that the member can only be accessed in its own class. The `protected` modifier specifies that the member can only be accessed within its own package (as with *package-private*) and, in addition, by a subclass of its class in another package.

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

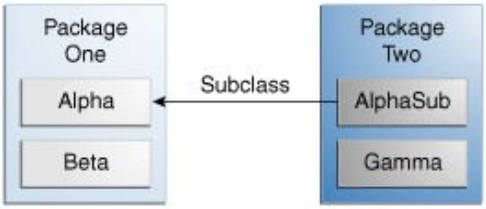
Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

The first data column indicates whether the class itself has access to the member defined by the access level. As you can see, a class always has access to its own members. The second column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member. The third column indicates whether subclasses of the class declared outside this package have access to the member. The fourth column indicates whether all classes have access to the member.

Access levels affect you in two ways. First, when you use classes that come from another source, such as the classes in the Java platform, access levels determine which members of those classes your own classes can use. Second, when you write a class, you need to decide what access level every member variable and every method in your class should have.

Let's look at a collection of classes and see how access levels affect visibility. The following figure shows the four classes in this example and how they are related.



Classes and Packages of the Example Used to Illustrate Access Levels

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

Visibility

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

Tips on Choosing an Access Level:

If other programmers use your class, you want to ensure that errors from misuse cannot happen. Access levels can help you do this.

- Use the most restrictive access level that makes sense for a particular member. Use `private` unless you have a good reason not to.
- Avoid `public` fields except for constants. (Many of the examples in the tutorial use public fields. This may help to illustrate some points concisely, but is not recommended for production code.) Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.

Garbage Collection

The concept of removing unused or unreferenced objects from the memory location is known as a **garbage collection**.

- While executing the program if garbage collection takes place, then more memory space is available for the program and rest of the program execution becomes faster.
- Garbage collector is a predefined program, which removes the unused or unreferenced objects from the memory location.
- Any object reference count becomes zero, then we call that object as a **unused** or **unreferenced** object.
- The number of reference variables which are pointing the object is known as a reference count of the object.
- While executing the program if any object reference count becomes zero, the internally java interpreter call the garbage collector and garbage collector will remove that object from memory location.

Introduction:

- In old languages like C++, **programmer is responsible** for **both creation and destruction of objects**. Usually programmer is taking very much care while creating object and neglect destruction of useless objects .Due to his negligence at certain point of time for creation of new object sufficient memory may not be available and entire application may be crashed due to memory problems.
- But **in java**, **programmer is responsible only for creation of new object** and he is not responsible for **destruction of objects**.
- Sun people **provided one assistant** which is always running in the background for destruction of useless objects. Due to this assistant the chance of failing java program is very rare because of memory problems.
- This assistant is nothing but garbage collector. Hence the **main objective of GC** is **to destroy useless objects**.

The ways to make an object eligible for GC:

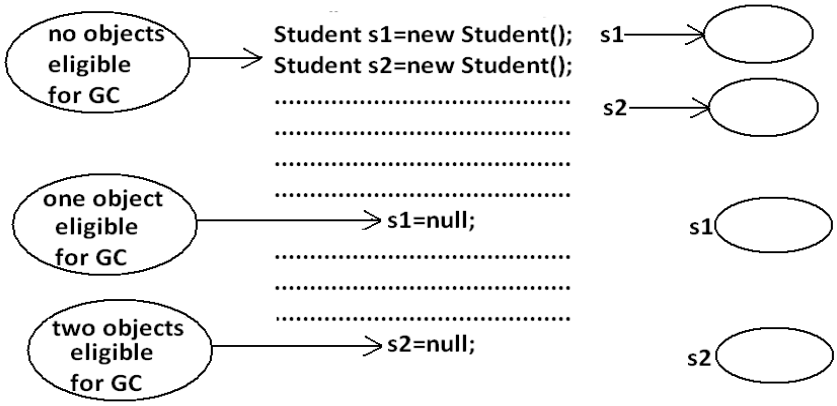
- Even through programmer is not responsible for destruction of objects but **it is always a good programming practice to make an object eligible for GC** if it is no longer required.
- An object is eligible for GC if and only if it does not have any references.

The following are various possible ways to make an object eligible for GC:

1.Nullifying the reference variable:

If an object is no longer required then we can make eligible for GC by assigning "**null**" to all its reference variables.

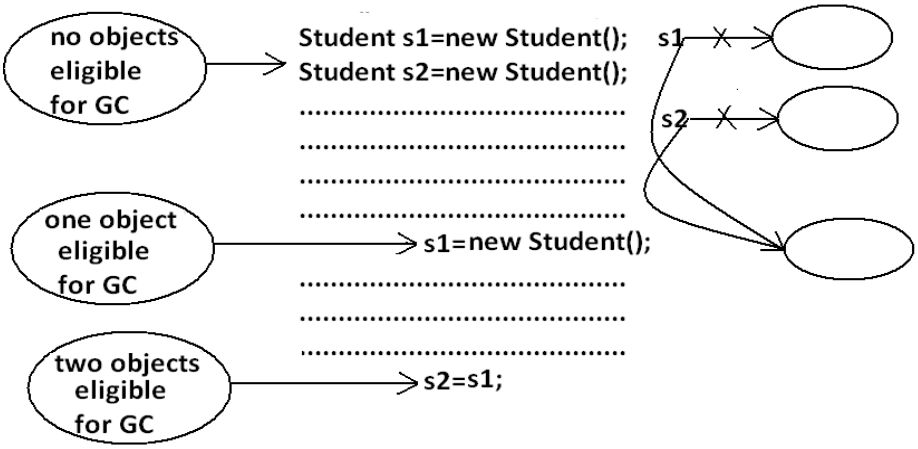
Example:



2.Reassign the reference variable:

If an object is no longer required then reassign all its reference variables to some other objects then old object is by default eligible for GC.

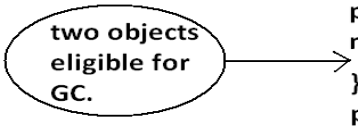
Example:



3. Objects created inside a method:

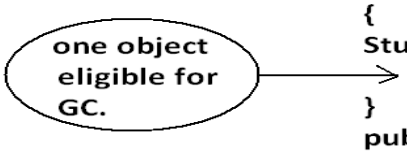
Objects created inside a method are by default eligible for GC once method completes.

Example 1:



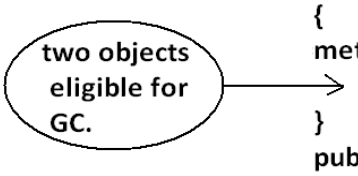

```
class Test
{
public static void main(String args[]){
methodOne();
}
public static void methodOne()
{
Student s1=new Student();
Student s2=new Student();
//.....
//.....
//.....
}
}
```

Example 2:



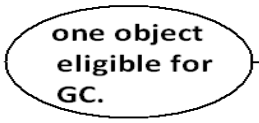

```
class Test
{
public static void main(String args[])
{
Student s=methodOne();
}
public static Student methodOne()
{
Student s1=new Student();
Student s2=new Student();
return s1;
}
}
```

Example 3:



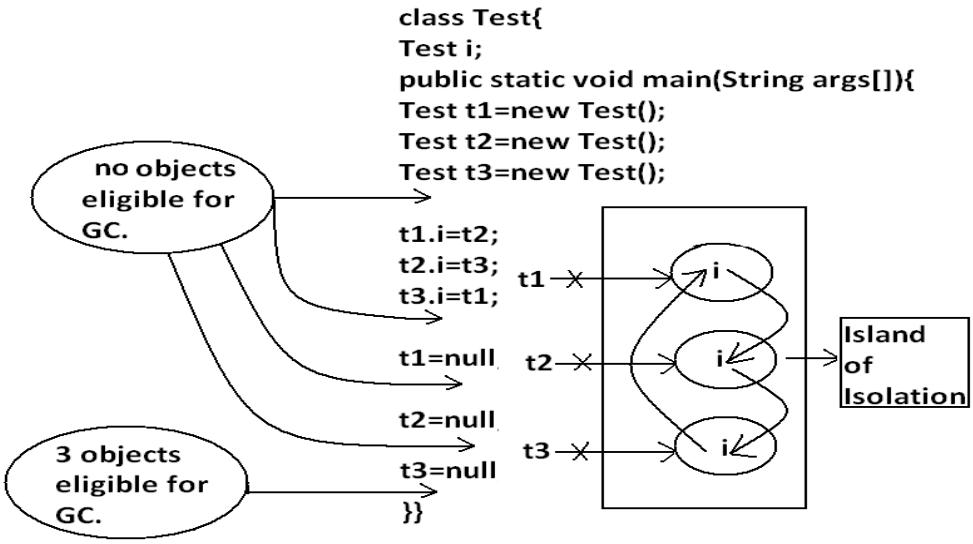

```
class Test
{
public static void main(String args[])
{
methodOne();
}
public static Student methodOne()
{
Student s1=new Student();
Student s2=new Student();
return s1;
}
}
```

Example 4:




```
class Test  {
 static Student s1 ;
public static void main(String args[])
{
 methodOne();
}
public static   void    methodOne()
{
Student s1=new Student();
Student s2=new Student();

}
}
```

4.Island of Isolation:



Note: If an object doesn't have any reference then it always eligible for GC.
Note: Even though object having reference still it is eligible for GC some times.

Example:

island of isolation. (Island of Isolation all references are internal references)

The methods for requesting JVM to run GC:

- Once we made an object eligible for GC it may not be destroyed immediately by the GC. Whenever jvm runs GC then only object will be destroyed by the GC. But when exactly JVM runs GC we can't expert it is vendor dependent.
- We can request jvm to run garbage collector programmatically, but whether jvm accept our request or not there is no guaranty. But most of the times JVM will accept our request.

The following are various ways for requesting jvm to run GC:

By System class:

System class contains a static method GC for this purpose.

Example:
System.gc();

By Runtime class:

- A java application can communicate with jvm by using Runtime object.
- Runtime class is a singleton class present in java.lang. Package.
- We can create Runtime object by using factory method getRuntime().

Example:

Runtime r=Runtime.getRuntime();

Once we got Runtime object we can call the following methods on that object.

freeMemory(): returns the free memory present in the heap.
totalMemory(): returns total memory of the heap.
gc(): for requesting jvm to run gc.

Example:

```
import java.util.Date;
class RuntimeDemo
{
	public static void main(String args[])
	{
		Runtime r=Runtime.getRuntime();
		S.o.p("total memory of the heap :"+r.totalMemory());
		S.o.p("free memory of the heap :"+r.freeMemory());
		for(int i=0;i<10000;i++)
		{
			Date d=new Date();
			d=null;
		}
		S.o.p("free memory of the heap :"+r.freeMemory());
		r.gc();
		S.o.p("free memory of the heap :"+r.freeMemory());
	}
}
Output:
Total memory of the heap: 5177344
Free memory of the heap: 4994920
Free memory of the heap: 4743408
Free memory of the heap: 5049776
```

Note : Runtime class is a singleton class so not create the object to use constructor.

Which of the following are valid ways for requesting jvm to run GC ?

```
System.gc(); (valid)
Runtime.gc(); (invalid)
(new Runtime).gc(); (invalid)
Runtime.getRuntime().gc(); (valid)
```

Note: gc() method present in System class is static, where as it is instance method in Runtime class.
Note: Over Runtime class gc() method , System class gc() method is recommended to use.
Note: in java it is not possible to find size of an object and address of an object.

Finalization:

- Just before destroying any object gc always calls finalize() method to perform cleanup activities.
- If the corresponding class contains finalize() method then it will be executed otherwise Object class finalize() method will be executed.

which is declared as follows.
protected void finalize() throws Throwable
Case 1:Just before destroying any object GC calls finalize() method on the object which is eligible for GC then the corresponding class finalize() method will be executed.

For Example if String object is eligible for GC then String class finalize()method is executed but not Test class finalize()method.

Example:

```
class Test
{
	public static void main(String args[])
	{
		String s=new String("bhaskar");
		Test t=new Test();
		s=null;
		System.gc();
		System.out.println("End of main.");
	}
	public void finalize()
	{
		S.o.p("finalize() method is executed");
	}
}
Output:
End of main.
```

In the above program String class finalize()method got executed. Which has empty implementation.

If we replace String object with Test object then Test class finalize() method will be executed .

The following program is an Example of this.

Example:

```
class Test
{
	public static void main(String args[])
	{
		String s=new String("bhaskar");
		Test t=new Test();
		t=null;
		System.gc();
		System.out.println("End of main.");
	}
	public void finalize()
	{
		S.o.p("finalize() method is executed");
	}
}
Output:
finalize() method is executed
End of main
```

Case 2:

We can call finalize() method explicitly then it will be executed just like a normal method call and object won't be destroyed. But before destroying any object GC always calls finalize() method.

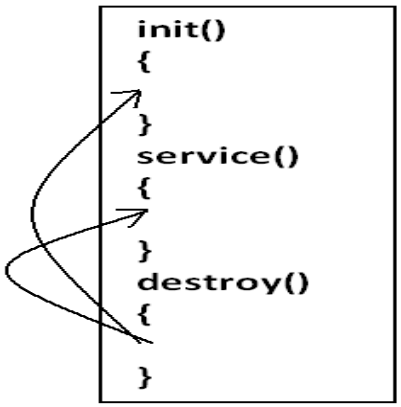
Example:

```
class Test
{
	public static void main(String args[])
	{
		Test t=new Test();
		t.finalize();
		t.finalize();
		t=null;
		System.gc();
		System.out.println("End of main.");
	}
	public void finalize()
	{
		System.out.println("finalize() method called");
	}
```

```
}
Output:
finalize() method called.
finalize() method called.
finalize() method called.
End of main.
```

In the above program finalize() method got executed 3 times in that 2 times explicitly by the programmer and one time by the gc.
Note: In Servlets we can call destroy() method explicitly from init() and service() methods. Then it will be executed just like a normal method call and Servlet object won't be destroyed.

Diagram:



Case 3:

finalize() method can be call either by the programmer or by the GC .

If the programmer calls explicitly finalize() method and while executing the finalize() method if an exception raised and uncaught then the program will be terminated abnormally.
If GC calls finalize() method and while executing the finalize()method if an exception raised and uncaught then JVM simply ignores that exception and the program will be terminated normally.

Example:

```
class Test
{
	public static void main(String args[])
	{
		Test t=new Test();
		//t.finalize();-------line(1)
		t=null;
		System.gc();
		System.out.println("End of main.");
	}
	public void finalize()
	{
		System.out.println("finalize() method called");
```

```
                System.out.println(10/0);
        }
}
```

If we are not comment line1 then programmer calling finalize() method explicitly and while executing the finalize()method ArithmeticException raised which is uncaught hence the program terminated abnormally.
If we are comment line1 then GC calls finalize() method and JVM ignores ArithmeticException and program will be terminated normally.

Which of the following is true?
While executing finalize() method JVM ignores every exception(invalid).
While executing finalize() method JVM ignores only uncaught exception(valid).

Case 4:

On any object GC calls finalize() method only once.

Example:

```
class FinalizeDemo
{
        static FinalizeDemo s;
        public static void main(String args[])throws Exception
        {
                FinalizeDemo f=new FinalizeDemo();
                System.out.println(f.hashCode());
                f=null;
                System.gc();
                Thread.sleep(5000);
                System.out.println(s.hashCode());
                s=null;
                System.gc();
                Thread.sleep(5000);
                System.out.println("end of main method");
        }
        public void finalize()
        {
                System.out.println("finalize method called");
                s=this;
        }
}
Output:
D:\Enum>java FinalizeDemo
4072869
finalize method called
4072869
End of main method
```

Note:
The behavior of the GC is vendor dependent and varied from JVM to JVM hence we can't expert exact answer for the following.

1. What is the algorithm followed by GC.
2. Exactly at what time JVM runs GC.
3. In which order GC identifies the eligible objects.
4. In which order GC destroys the object etc.
5. Whether GC destroys all eligible objects or not.

When ever the program runs with low memory then the JVM runs GC, but we can't except exactly at what time.
Most of the GC's followed **mark & sweap** algorithm , but it doesn't mean every GC follows the same algorithm.

Memory leaks:

- An object which is not using in our application and it is not eligible for GC such type of objects are called "memory leaks".
- In the case of memory leaks GC also can't do anything the application will be crashed due to memory problems.
- In our program if memory leaks present then certain point we will get **OutOfMemoryException**. Hence if an object is no longer required then it's highly recommended to make that object eligible for GC.
- By using monitoring tools we can identify memory leaks.

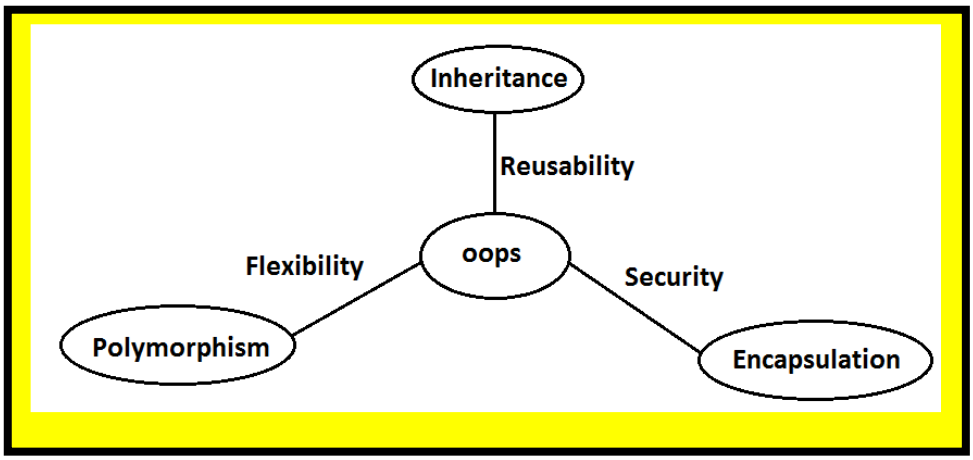
```
Example:

HPJ meter
HP ovo
IBM Tivoli            These are monitoring tools.
J Probe                      (or memory management tools)
Patrol and etc
```

Polymorphism

Agenda:

1. **Polymorphism**
   a) **Overloading**
      - **Automatic promotion in overloading**

Pillars of OOPS:



1) Inheritance talks about reusability.
2) Polymorphism talks about flexibility.
3) Encapsulation talks about security.

Polymorphism:

- Polymorphism is a Greek word poly means many and morphism means forms.
- Same name with different forms is the concept of polymorphism.

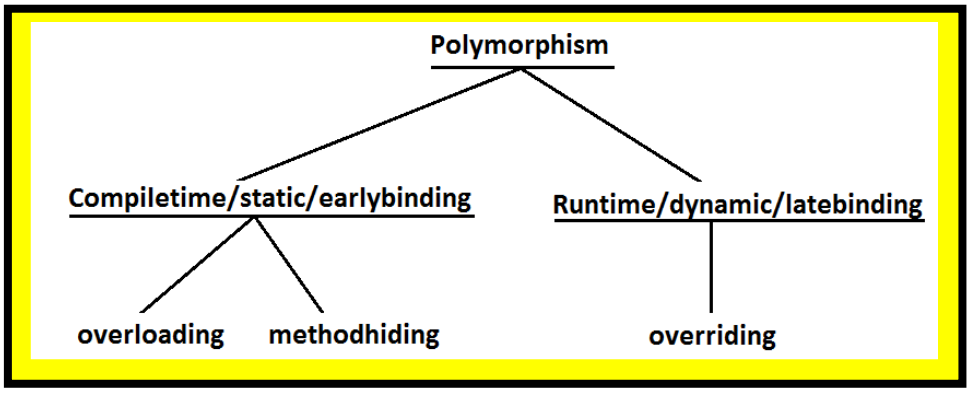
Ex 1: We can use **same abs( ) method** for int type, long type, float type etc.
Ex:

1. abs(int)
2. abs(long)
3. abs(float)

Ex 2: We can use the **parent reference** to hold any **child objects**. We can use the same List reference to hold ArrayList object, LinkedList object, Vector object, or Stack object.
Ex:

1. List l=new ArrayList( );
2. List l=new LinkedList( );
3. List l=new Vector( );
4. List l=new Stack( );

Diagram:

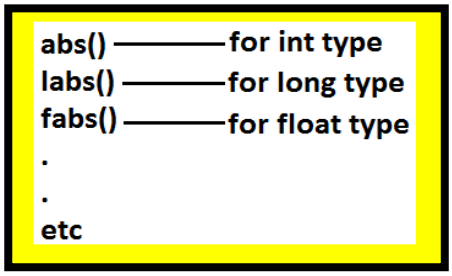


There are two types of polymorphism in java

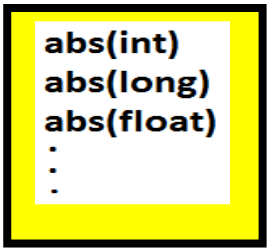
1. **Compile time polymorphism**: Its method execution decided at compilation time.
   Example :- method overloading.
2. **Runtime polymorphism**: Its method execution decided at runtime.
   Example :- method overriding.

Overloading:-

1) Two methods are said to be overload, if and only if both having the same name but different argument types.
2) In 'C' language, we can't take 2 methods with the same name and different types. If there is a change in argument type, compulsory we should go for new method name.
   Example:



3) Lack of overloading in "C" increases complexity of the programming.
4) But in java, we can take multiple methods with the same name and different argument types.
   Example:



5) Having the same name and different argument types is called method overloading.

6) All these methods are considered as overloaded methods.
7) Having overloading concept in java reduces complexity of the programming.

```
class Test
{
        public void methodOne()
        {
                System.out.println("no-arg method");
        }
        public void methodOne(int i)
        {
                System.out.println("int-arg method");
           //overloaded methods
        }
        public void methodOne(double d)
        {
                System.out.println("double-arg method");
        }
        public static void main(String[] args)
        {
                Test t=new Test();
                t.methodOne();        //no-arg method
                t.methodOne(10);      //int-arg method
                t.methodOne(10.5);//double-arg method
        }
}
```

Conclusion : In overloading, compiler is responsible to perform method resolution(decision) based on the reference type(but not based on run time object). Hence overloading is also considered as compile time polymorphism(or) static polymorphism (or)early biding.

Parameter Passing Mechanism in Java

1. Introduction

The two most prevalent modes of passing arguments to methods are “passing-by-value” and “passing-by-reference”. Different programming languages use these concepts in different ways. **As far as Java is concerned, everything is strictly *Pass-by-Value*.**

In this tutorial, we're going to illustrate how Java passes arguments for various types.

2. Pass-by-Value vs Pass-by-Reference

Let's start with some of the different mechanisms for passing parameters to functions:

- value
- reference
- result
- value-result
- name

The two most common mechanisms in modern programming languages are “Pass-by-Value” and “Pass-by-Reference”. Before we proceed, let's discuss these first:

2.1. Pass-by-Value

When a parameter is pass-by-value, the caller and the callee method operate on two different variables which are copies of each other. Any changes to one variable don't modify the other.

It means that while calling a method, **parameters passed to the callee method will be clones of original parameters.** Any modification done in callee method will have no effect on the original parameters in caller method.

2.2. Pass-by-Reference

When a parameter is pass-by-reference, the caller and the callee operate on the same object.

It means that when a variable is pass-by-reference, **the unique identifier of the object is sent to the method.** Any changes to the parameter’s instance members will result in that change being made to the original value.

3. Parameter Passing in Java

The fundamental concepts in any programming language are “values” and “references”. In Java, **Primitive variables store the actual values, whereas Non-Primitives store the reference variables which point to the addresses of the objects they're referring to.** Both values and references are stored in the stack memory.

Arguments in Java are always passed-by-value. During method invocation, a copy of each argument, whether its a value or reference, is created in stack memory which is then passed to the method.

In case of primitives, the value is simply copied inside stack memory which is then passed to the callee method; in case of non-primitives, a reference in stack memory points to the actual data which resides in the heap. When we pass an object, the reference in stack memory is copied and the new reference is passed to the method.

Let's now see this in action with the help of some code examples.

3.1. Passing Primitive Types

The Java Programming Language features eight primitive data types. **Primitive variables are directly stored in stack memory. Whenever any variable of primitive data type is passed as an argument, the actual parameters are copied to formal arguments and these formal arguments accumulate their own space in stack memory.**

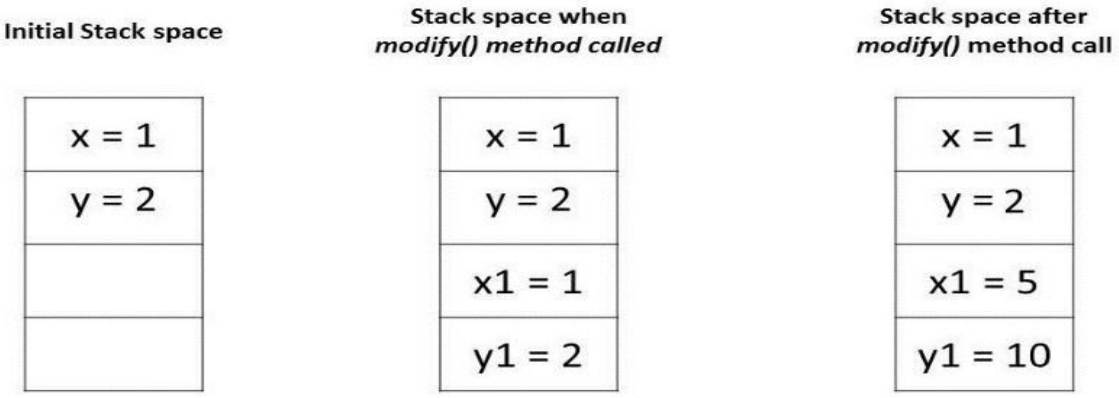
The lifespan of these formal parameters lasts only as long as that method is running, and upon returning, these formal arguments are cleared away from the stack and are discarded.

Let's try to understand it with the help of a code example:

```
public class Test74
{
        public void m1()
        {
                int x=1;
                int y=2;
                //Before modification
                System.out.println("x="+x);
                System.out.println("y="+y);

                modify(x,y);

                //after modification
                System.out.println("x="+x);
                System.out.println("y="+y);
        }
        public static void modify(int x1,int y1)
        {
                int sum1,sum2;
                sum1=x1+y1;                     //sum1=3
                System.out.println("sum1="+sum1);
                x1=5;
                y1=10;
                sum2=x1+y1;                     //sum2=15
                System.out.println("sum2="+sum2);

        }
        public static void main(String[] args)
        {
                Test74 obj=new Test74();
                obj.m1();
        }
}
```

Let's try to understand the assertions in the above program by analyzing how these values are stored in memory:

1. The variables "*x*" and "*y*" in the main method are primitive types and their values are directly stored in the stack memory
2. When we call method *modify()*, an exact copy for each of these variables is created and stored at a different location in the stack memory
3. Any modification to these copies affects only them and leaves the original variables unaltered



3.2. Passing Object References

In Java, all objects are dynamically stored in Heap space under the hood. These objects are referred from references called reference variables.

A Java object, in contrast to Primitives, is stored in two stages. The reference variables are stored in stack memory and the object that they're referring to, are stored in a Heap memory.

Whenever an object is passed as an argument, an exact copy of the reference variable is created which points to the same location of the object in heap memory as the original reference variable.

As a result of this, whenever we make any change in the same object in the method, that change is reflected in the original object. However, if we allocate a new object to the passed reference variable, then it won't be reflected in the original object.

Call by Value and Call by Reference in Java

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

Example of call by value in java.

In case of call by value original value is not changed. Let's take a simple example:

```
class Operation{
 int data=50;

 void change(int data){
 data=data+100;//changes will be in the local variable only
 }

 public static void main(String args[]){
   Operation op=new Operation();

   System.out.println("before change "+op.data);
   op.change(500);
   System.out.println("after change "+op.data);

 }
}
```

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

Another Example of call by value in java

In case of call by reference original value is changed if we made changes in the called method. If we pass object in place of any primitive value, original value will be changed. In this example we are passing object as a value. Let's take a simple example:

```
class Operation2{
int data=50;

void change(Operation2 op){
op.data=op.data+100;//changes will be in the instance variable
}



public static void main(String args[]){
  Operation2 op=new Operation2();

  System.out.println("before change "+op.data);
  op.change(op);//passing object
  System.out.println("after change "+op.data);

 }
}
```

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

INNER CLASSES

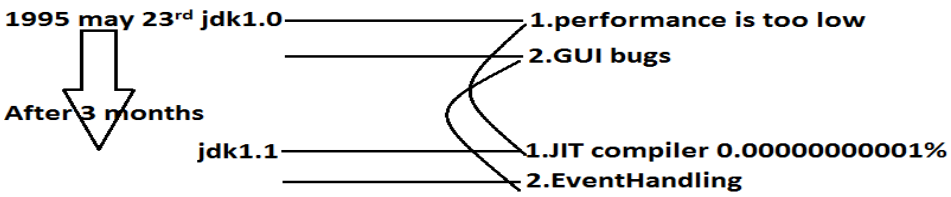
Agenda:

1. Introduction.
2. Applicable modifiers for outer classes and inner classes
3. Nested classes
4. Inner classes

Introduction:

- Sometimes we can declare a class inside another class such type of classes are called inner classes.

Diagram:



- Sun people introduced inner classes in 1.1 version as part of "Event Handling" to resolve GUI bugs.
- But because of powerful features and benefits of inner classes slowly the programmers starts using in regular coding also.
- Without existing one type of object if there is no chance of existing another type of object then we should go for inner classes.

Example1:

Without existing University object there is no chance of existing Department object hence we have to define Department class inside University class.

```
class University ———————— Outer class
{
     class Department ———— inner class
     {
     }
}
```

Example 2:

Without existing Bank object there is no chance of existing Account object hence we have to

define Account class inside Bank class.

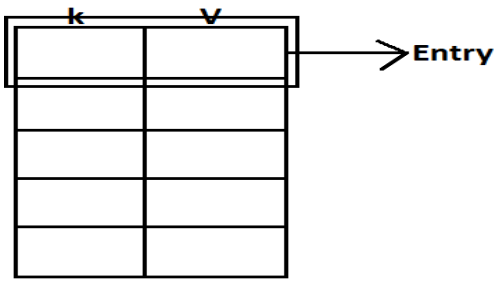
```
class Bank ——————————— Outer class
{
      class Account ———— inner class
      {
      }
}
```

Example 3:

Without existing Map object there is no chance of existing Entry object hence Entry interface is define inside Map interface.
Map is a collection of key-value pairs, each key-value pair is called an Entry.

```
interface Map ———— outer interface
{
      interface Entry —— inner interface
      {
      }
}
```

Diagram:



Note : Without existing Outer class Object there is no chance of existing Inner class Object.

Note: **The relationship between outer class and inner class is not IS-A relationship and it is Has-A relationship.**

The applicable modifiers for outer classes are:

1. public
2. default
3. final
4. abstract
5. strictfp

But for the inner classes in addition to this the following modifiers also allowed.
Diagram:

Nested classes:

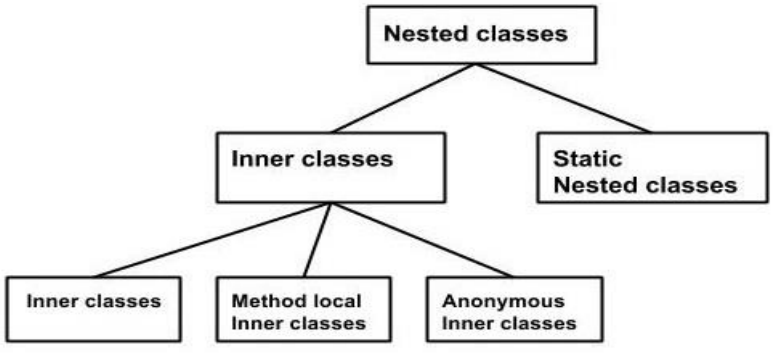
- It is a class defined within another class.
- **Nested classes** that are declared static are called **static nested classes**. A **nested class** is a member of its enclosing **class**.
- **Does not have access to instance members of containing class**

Inner classes:

- It is a class defined within another class.
- Non-static **nested classes** are called **inner classes**. Non-static **nested classes** (**inner classes**) have access to other members of the enclosing **class**, even if they are declared private.
- **Instances have access to instance members of containing class.**

Based on the purpose and position of declaration all **inner classes are divided into 3 types**. They are:

1. Normal or Regular inner classes
2. Method Local inner classes: It is a class defined within a method. It is only available within the method
3. Anonymous inner classes: It is a class defined with no class name! It is defined during reference declaration

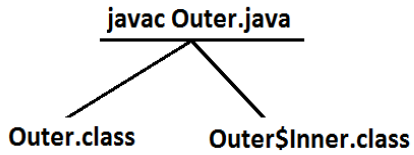


1. Normal (or) Regular inner class:

If we are declaring any named class inside another class directly **without static modifier** such type of inner classes are called normal or regular inner classes.

Example:

```
class Outer
{
        class Inner
        {
        }
}
Output:
```




```
E:\scjp>java Outer
Exception in thread "main" java.lang.NoSuchMethodError: main
E:\scjp>java Outer$Inner
Exception in thread "main" java.lang.NoSuchMethodError: main
```

Example:

```
class Outer
{
        class Inner
        {
        }
        public static void main(String[] args)
        {
                System.out.println("outer class main method");
        }
}
Output:
```

```
javac Outer.java
———— Outer.class
———— Outer$Inner.class
E:\scjp>java Outer
outer class main method
E:\scjp>java Outer$Inner
Exception in thread "main" java.lang.NoSuchMethodError: main
```

- Inside inner class we **can't declare static members**. Hence it is not possible to declare main() method and we can't invoke inner class directly from the command prompt.

Example:

```
class Outer
{
        class Inner
        {
```

```
                public static void main(String[] args)
                {
                        System.out.println("inner class main method");
                }
        }
}
Output:
E:\scjp>javac Outer.java
Outer.java:5: inner classes cannot have static declarations
                  public static void main(String[] args)
```

Accessing inner class code from static area of outer class:

Example:

```
class Outer
{
        class Inner
        {
                public void methodOne(){
                        System.out.println("inner class method");
                }
        }
        public static void main(String[] args)
        {
                Outer o=new Outer();
                Outer.Inner i=o.new Inner();
                i.methodOne();
                                                    (or)
                Outer.Inner i=new Outer().new Inner();
                i.methodOne();
                                                    (or)
                new Outer().new Inner().methodOne();
           }
}
```

Accessing inner class code from instance area of outer class:

Example:

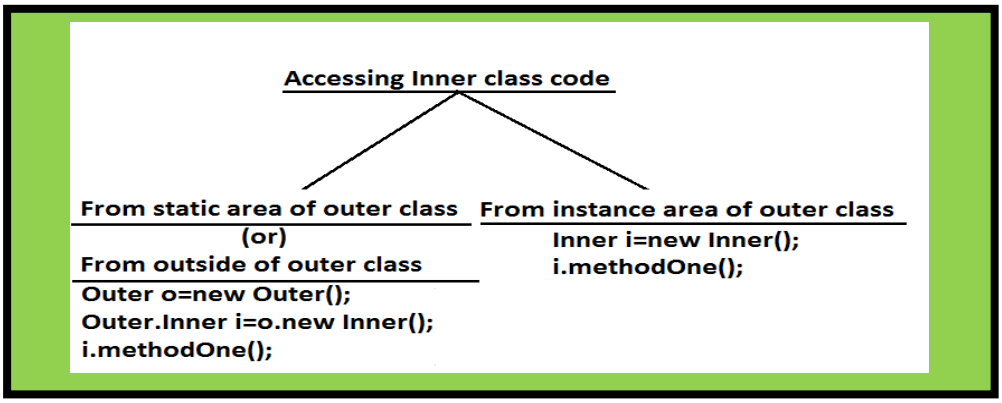
```
class Outer
{
        class Inner
        {
                public void methodOne()
                {
                        System.out.println("inner class method");
                }
        }
        public void methodTwo()
        {
                Inner i=new Inner();
                i.methodOne();
        }
        public static void main(String[] args)
        {
```

```
                Outer o=new Outer();
                o.methodTwo();
        }
}
Output:
E:\scjp>javac Outer.java
E:\scjp>java Outer
Inner class method
```

Accessing inner class code from outside of outer class:

Example:

```
class Outer
{
        class Inner
        {
                public void methodOne()
                {
                        System.out.println("inner class method");
                }
        }
}
class Test
{
        public static void main(String[] args)
        {
                new Outer().new Inner().methodOne();
        }
}
Output:
Inner class method
```



- From inner class we can access all members of outer class (both static and non-static, private and non private methods and variables) directly.

Example:

```
class Outer
{
        int x=10;
        static int y=20;
```

```
        class Inner{
                public void methodOne()
                {
                        System.out.println(x);      //10
                        System.out.println(y);      //20
                }
        }
        public static void main(String[] args)
        {
                new Outer().new Inner().methodOne();
        }
}
```

- Within the inner class "**this**" always refers current inner class object. To refer current outer class object we have to use "**outerclassname.this**".

Example:

```
class Outer
{
        int x=10;
        class Inner
        {
                int x=100;
                public void methodOne()
                {
                        int x=1000;
                        System.out.println(x);//1000
                        System.out.println(this.x);//100
                        System.out.println(Outer.this.x);//10
                }
        }
        public static void main(String[] args)
        {
                new Outer().new Inner().methodOne();
        }
}
```

Nesting of Inner classes :

We can declare an inner class inside another inner class.

Diagram: