

CHAPTER-1

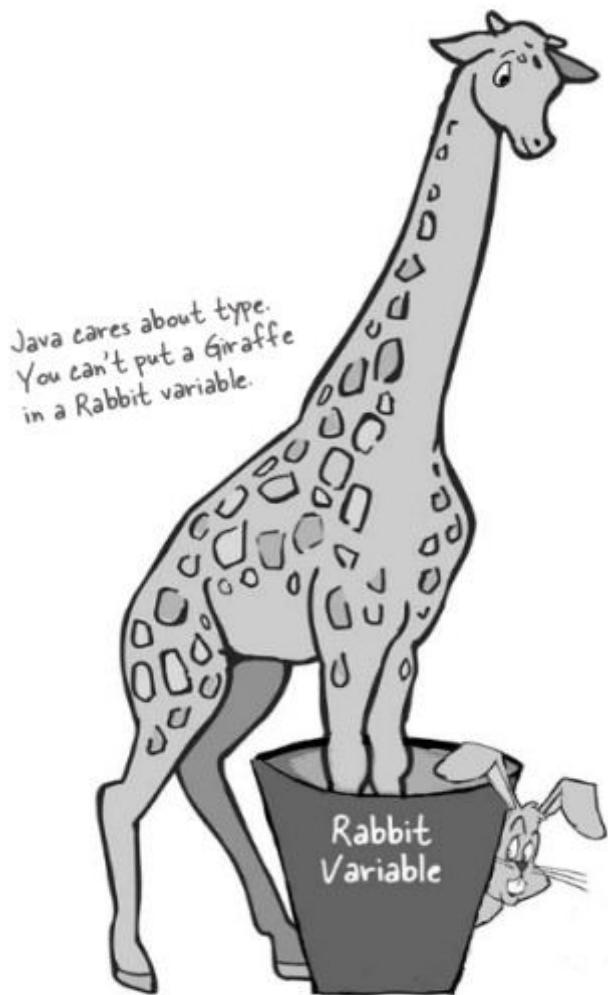
INTRODUCTION

FUNDAMENTAL DATA TYPES

DECISIONS

LOOPS

ARRAYS



1. FUNDAMENTALS OF OBJECT ORIENTED PROGRAMMING

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

Definition of Object Oriented Program is:

Object oriented programming is an approach that provides a way of modularizing 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 that an object is considered to be partitioned area of computer memory that stores data and set of operations that can access the data. Since the memory applications are independent, the objects can be used in a variety of different programs without modifications.

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.
- We can build programs from the standard working modules that communicate with one another, and to start writing the code from scratch. This shows to saving of development time and higher productivity.
- 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 to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach capture more details of a model in implementable form.
- Object oriented systems can be easily upgraded from small to large systems.
- Software complexity can be easily managed.

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:- <code>#include<stdio.h> void main() { printf(“hello world”); }</code>	CPP –sample application:- <code>#include<iostream.h> void main() {cout<<“hello world”; }</code>	

Java sample application:-

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

JAVA introduction:-

i. **Author :** James Gosling



- ii. **Vendor :** Sun Micro System(which has since merged into Oracle Corporation)
- iii. **Project name :** Green Project
- iv. **Type :** open source & free software
- v. **Initial Name :** OAK language



- vi. **Present Name :** java
- vii. **Extensions :** .java & .class & .jar
- viii. **Initial version :** jdk 1.0 (java development kit)
- ix. **Present version :** java 12, 2019
- x. **Operating System :** multi Operating System
- xi. **Implementation Lang :** c, cpp.....
- xii. **Symbol :** coffee cup with saucer



- xiii. **Objective :** To develop web applications
- xiv. **SUN :** Stanford Universally Network
- xv. **Slogan/Motto :** WORA(Write Once Run Anywhere)

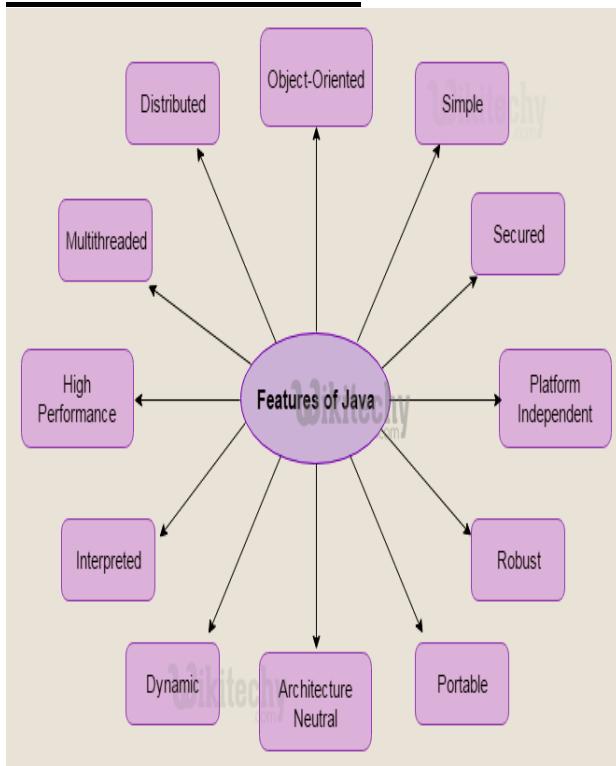
IMPORTANCE OF CORE JAVA:-

According to Sun, 3 billion devices run Java. There are many devices where Java is currently used. Some of them are as follows:

1. Desktop Applications such as acrobat reader, media player, antivirus, etc.
2. Web Applications such as irctc.co.in, redbus.in, etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System

6. Smart Card
7. Robotics
8. Games, etc.

Features of Java:



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

1. Simple:-

Java is a simple programming language because:

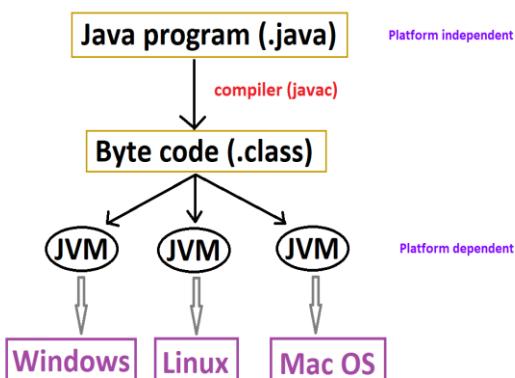
- Java technology has eliminated all the difficult and confusion oriented concepts like pointers, multiple inheritance in the java language.
- The c, cpp syntaxes easy to understand and easy to write. Java maintains C and CPP syntax mainly hence java is simple language.
- Java tech takes less time to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

Identifier :

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

Example:

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

1 2 3 4
 | | | |
 | 2 3 4
 | | | |
 | 5

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.

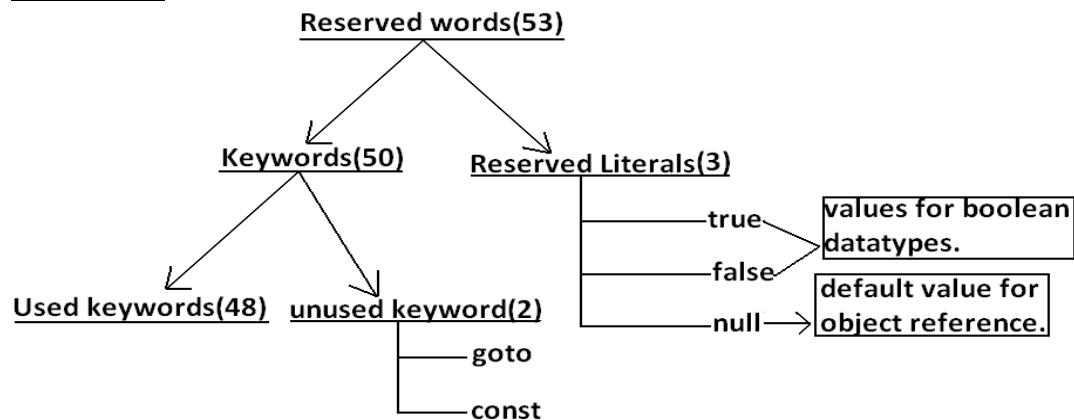
Which of the following are valid java identifiers?

- 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
synchronized but not synchronize
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**)

DATA TYPES

Agenda:

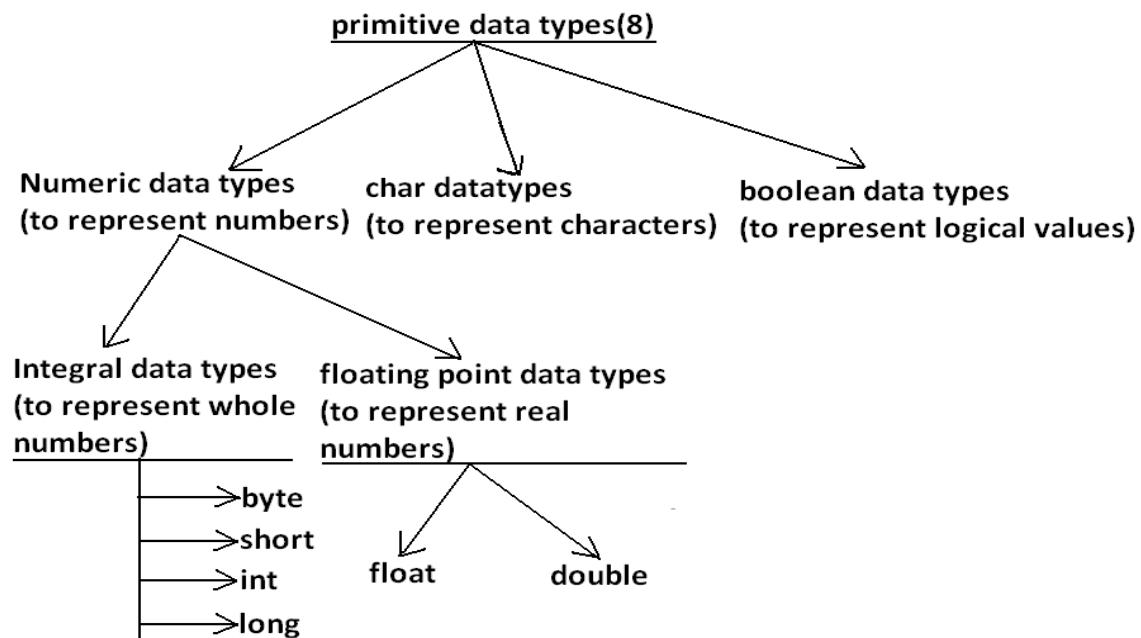
Data types

- 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:

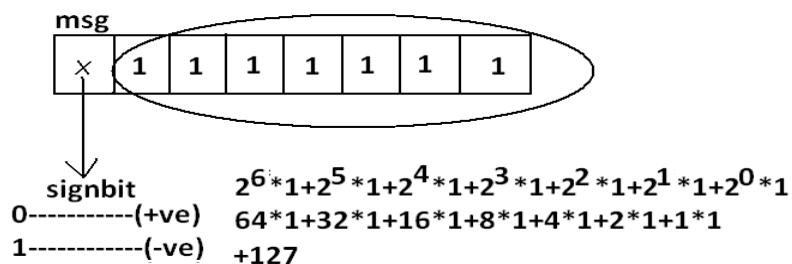


Except Boolean and char all remaining data types are considered as signed data types because we can represent both "+ve" and "-ve" numbers.

Integral data types :

byte:

```
Size: 1byte (8bits)
Maxvalue: +127
Minvalue:-128
Range:-128to 127 [-27 to 27-1]
```



- 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 (-215 to 215-1)
```

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 (-231 to 231-1)
```

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:-263 to 263-1
```

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 to 1.7e308.
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?

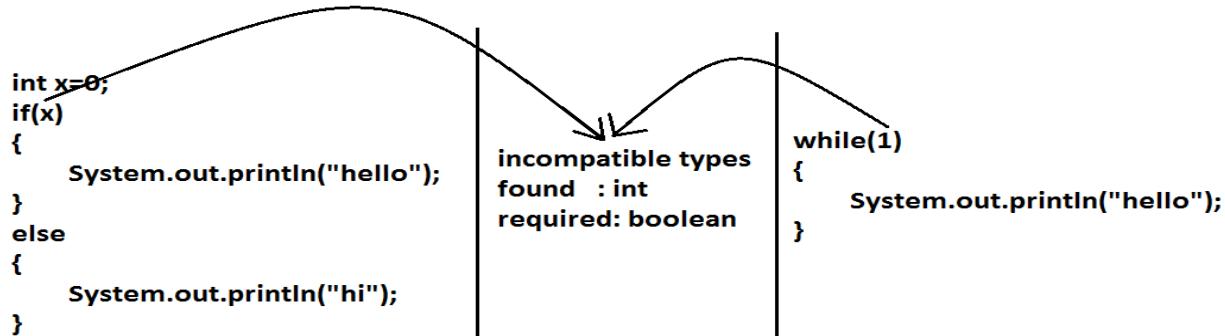
Example 1:

```

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 65536

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

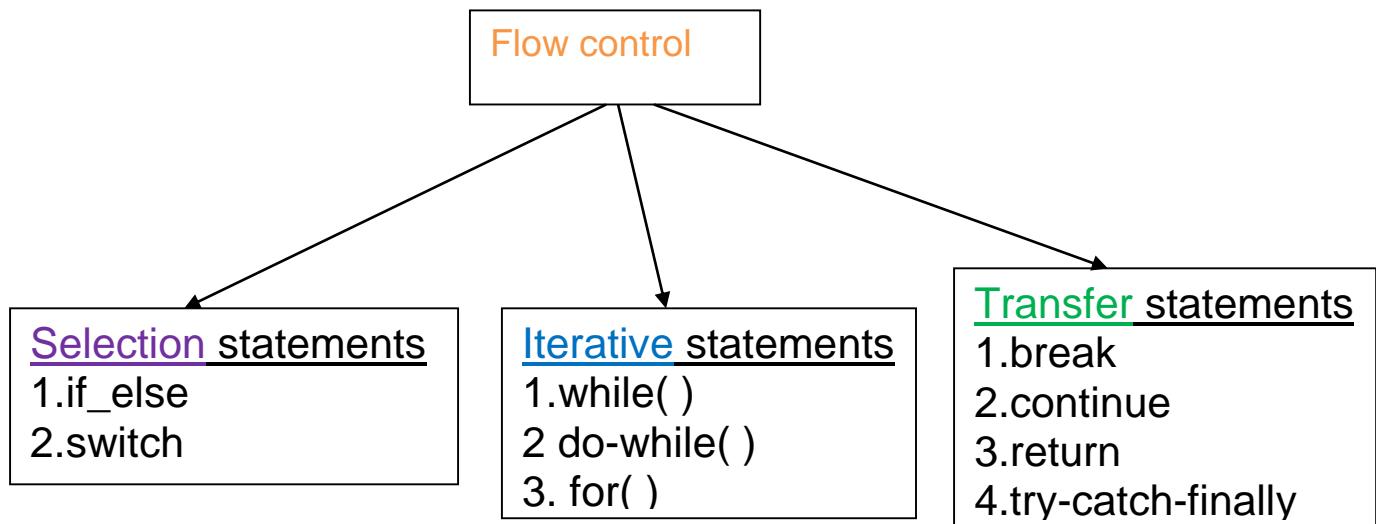
int	4 bytes	-2 ³¹ to 2 ³¹ -1 (-2147483648 to 2147483647)	Integer	0
long	8 bytes	-2 ⁶³ to 2 ⁶³ -1	Long	0
float	4 bytes	-3.4e38 to 3.4e38	Float	0.0
double	8 bytes	-1.7e308 to 1.7e308	Double	0.0
boolean	Not applicable	Not applicable(but allowed values true false)	Boolean	false
char	2 bytes	0 to 65535	Character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

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

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 braces are optional whenever we are taking single statements and the curly braces are mandatory whenever we are taking multiple statements

Ex-1:-

```
class Test
{
    public static void main(String[ ] args)
    {
        int a=10;      int b=20;
        if (a<b)
        {
            System.out.println("ifbody/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");
            case20:System.out.println("bunny");
            case 30:System.out.println("chinny");
            default:System.out.println("vinny");
        }
    }
}
```

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");
            case 10:System.out.println("bunny");
            case 30:System.out.println("chinny");
            default:System.out.println("vinny");
        }
    }
}
```

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");
            case 10+5:System.out.println("bunny");
            case 30/6:System.out.println("chinny");
            default:System.out.println("vinny");
        }
    }
}
```

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");
            case b:System.out.println("bunny");
            case c:System.out.println("chinny");
            default:System.out.println("vinny");
        }
    }
}
```

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)
    {
        charch='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 break 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 break is available up to the end of the switch is executed. This situation is called as fall through 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)
    {
        charch='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:-

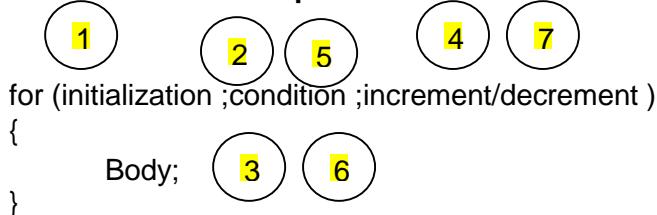
By using iteration statements we are able to execute group of statements repeatedly or more number of times.

- 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 single 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);
        }
    }
}
```

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. 0 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:

<code>+= , -= , *= , /= , %=</code>
<code>&= , = , ^=</code>
<code>>>= , >>>= , <<=</code>

9. Conditional operator(?:)

The only possible ternary operator in java is conditional operator.

Syntax:

<code>X=firstValue if condition else</code>
<code>secondValue</code>

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	<code>*, /, %, +, -</code>
3. Shift	<code>>>, >>>, <<</code>
4. Relational	<code><, <=, >, >=</code>
5. Equality	<code>==, !=</code>
6. Bitwise	<code>&, ^, </code>
7. Short circuit logical	<code>&&, </code>
8. Conditional	<code>?:</code>
9. Assignment	= Lowest

Summary:

The various operators are tabulated as below

Operator	Result
+	Addition
-	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
--	Decrement

table:Arithmetic operators

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

table: bitwise operators

Operator	Result
&	Logical AND
	Logical OR
^	Logical XOR (exclusive OR)
	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
:?	Ternary if-then-else

table :logical operators

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

table: relational operators

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:

```
int x=10
```

constant value | literal
name of variable | identifier
datatype | keyword

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=010;

3) Hexa Decimal literals:

- The allowed digits are 0 to 9, A to Z.
- 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=0x10;

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=0xFACE; (valid)
4. int x=0xbeef; (valid)
5. int x=0xBeer; //C.E: ';' expected(invalid) //:int x=0xBeer; ^// ^
6. int x=0xabb2cd;(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 "I" (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:

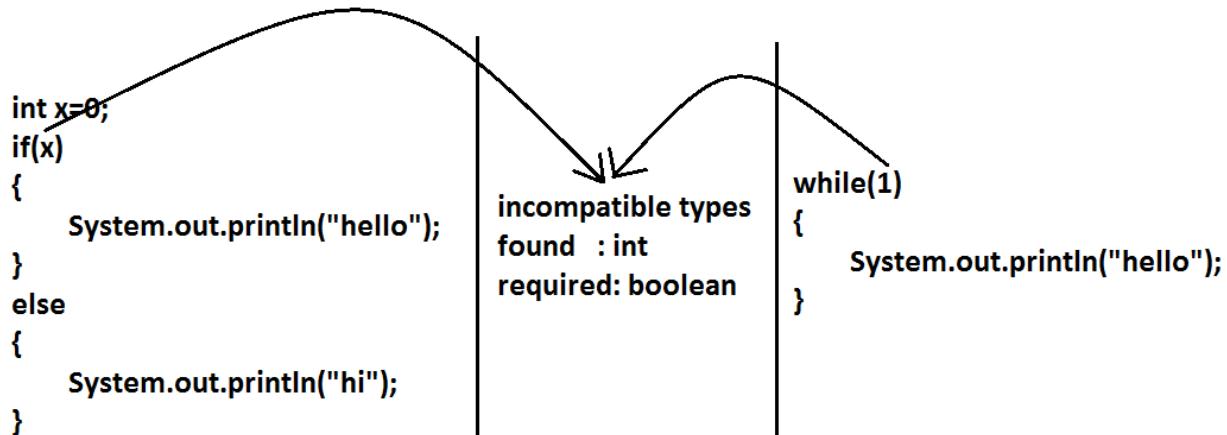
```
double d=10e2; //==>10*102 (valid)
System.out.println(d); //1000.0
float f=10e2; //C.E:possible loss of precision (invalid)
float f=10e2F; (valid)
```

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)

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;
```

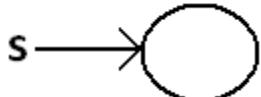
Reference variables:

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 created 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.

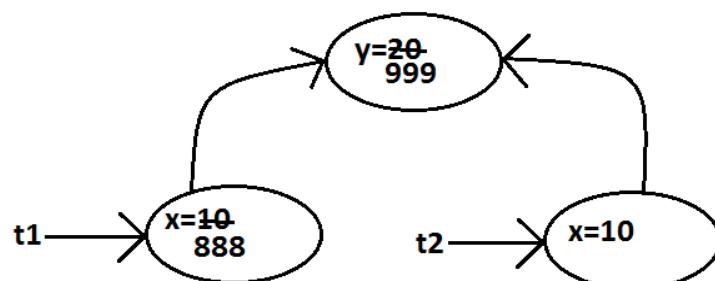
Example:

```
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);
    }
}
```

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 object created	when .class file loading
When memory destroyed	when method ends.	When object destroyed	when .class unloading.
Initial values	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.

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()` - It 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

```
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!
```

MODIFIERS

Modifiers:

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

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

Table 1.1 Modifiers for classes, interfaces, and enums

Modifier	Meaning
<code>private</code>	It is accessible only from within its own class.
<code>protected</code>	It is accessible only from within its own class and its extensions.
<code>public</code>	It is accessible from all classes.

Table 1.2 Constructor modifiers

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

Table 1.3 Field modifiers

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

Table 1.4 Method modifiers

Modifier	Meaning
<code>final</code>	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

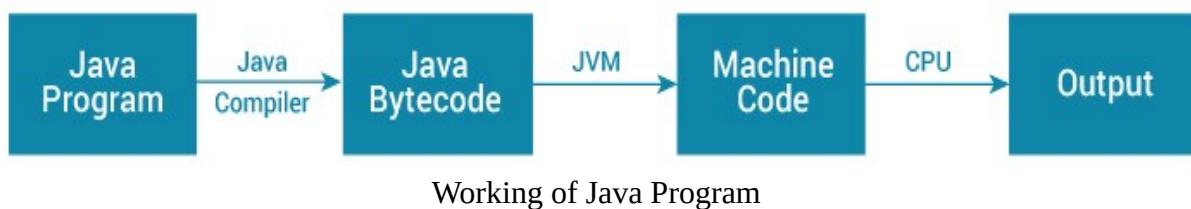
Java JDK, JRE and JVM

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.



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).

JDK

JRE

+ Compilers + Debuggers ...

Java Development Kit

Relationship between JVM, JRE, and JDK.

JDK

JRE

JVM

+

Class Libraries

+

Compilers

Debuggers

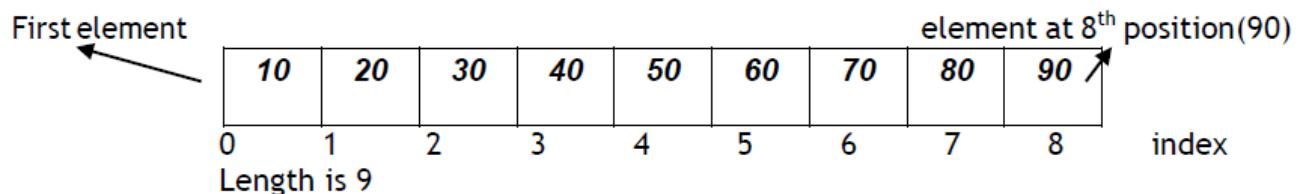
JavaDoc

Relationship between JVM, JRE, and JDK

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:

<code>int[] a;</code>	<code>//valid</code>
<code>int[5] a;</code>	<code>//invalid</code>

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:

`int[] []a,[]b;`

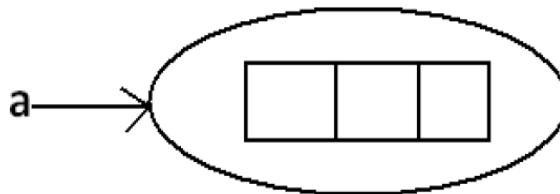
invalid
valid

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)
```

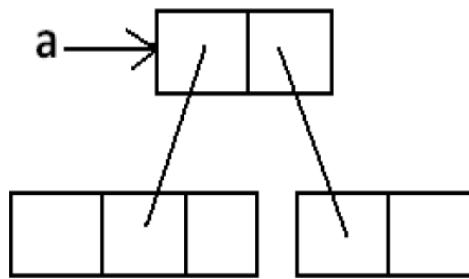
MULTIDIMENSIONAL ARRAY CREATION:

- 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];
```

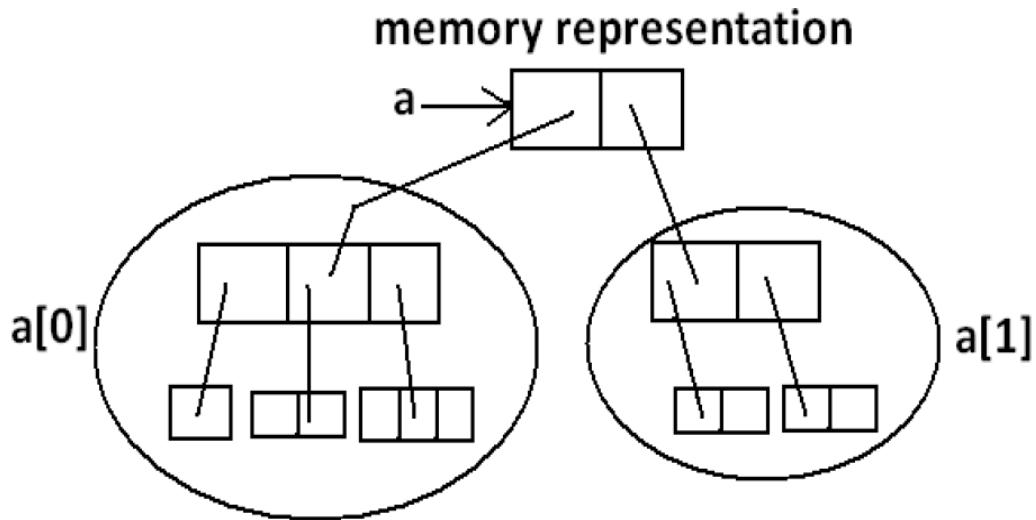
Diagram of memory representation:



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?

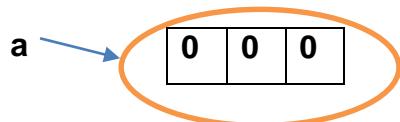
- 1) `int[] a=new int[];` //array dimension is missing (invalid)
- 2) `int[][] a=new int[3][4];` (valid)
- 3) `int[][] a=new int[3][];` (valid)
- 4) `int[][] a=new int[][4];` (invalid)
- 5) `int[][][] a=new int[3][4][5];` (valid)
- 6) `int[][][] a=new int[3][][5];` (invalid)

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:

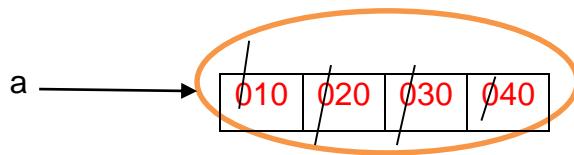
```
int[ ][ ] a=new int[2][];
System.out.println(a); // [[I@3e25a5
System.out.println(a[0]); // null
System.out.println(a[0][0]); // R.E:NullPointerException
```

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;
```

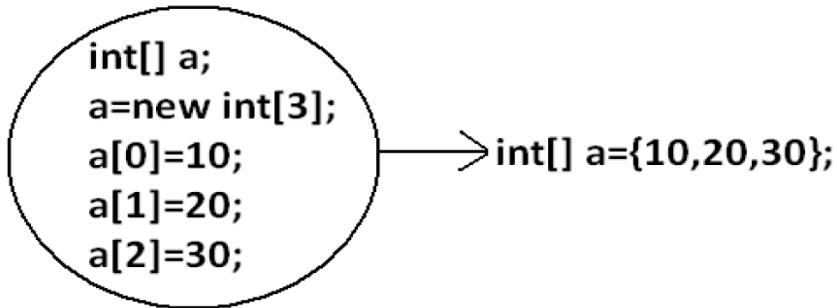
```
a[1]=20;  
a[2]=30;  
a[3]=40;  
a[4]=50;//R.E:ArrayIndexOutOfBoundsException: 4  
a[-4]=-60;//R.E:ArrayIndexOutOfBoundsException: -4
```



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.



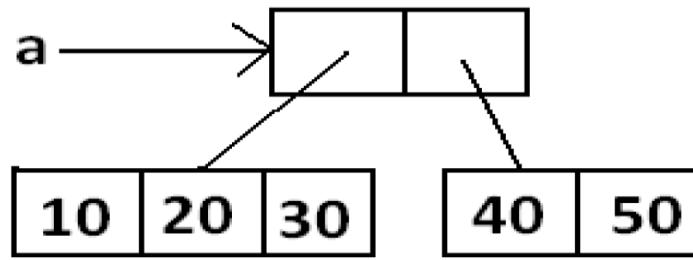
`char[] ch={'a','e','i','o','u'}; (valid)`

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

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

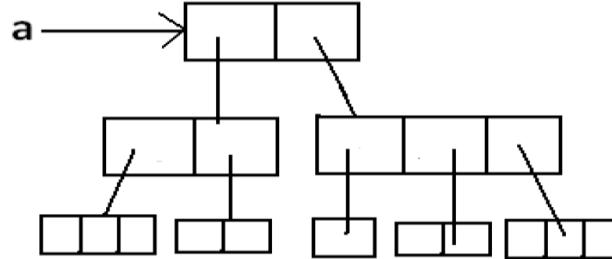
Example:

`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:

```
int[] x={10,20,30};
```

```
int[] x;  
x=new int[3];  
x={10,20,30};
```

→ C.E:illegal start of expression

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.

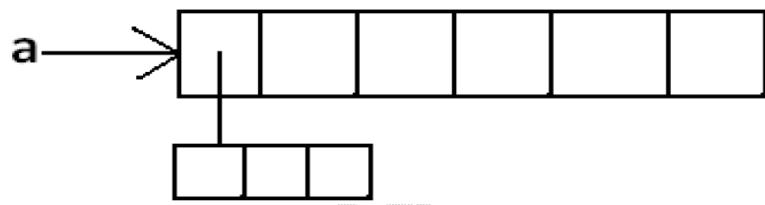
Example:

```
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 whereas **length()** method is applicable for String objects.
- There is no direct way to find total size of multi dimensional 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.

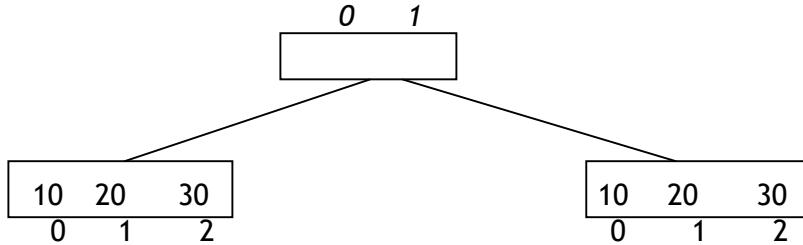
```
class Test
{
    public static void main(String[] args)
    {
        String[] str = new String[5];
        str[0] = "ratan";
        str[1] = "anu";
        str[2] = null;
        str[3] = "sravya";
        str[4] = null;
        for (int i = 0; i < str.length; i++)
        {
            if (str[i]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

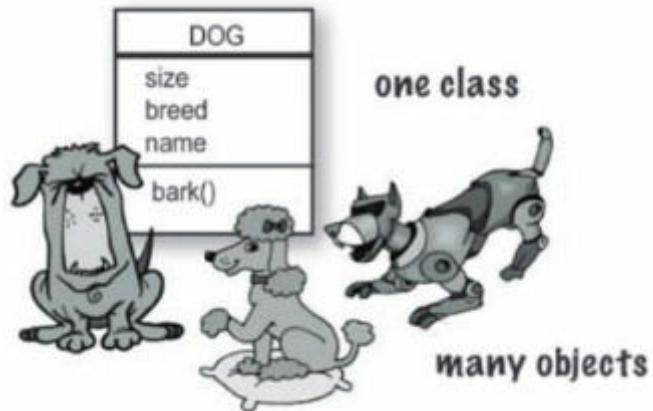
CHAPTER-2

OBJECTS and CLASSES

METHODS

STRINGS

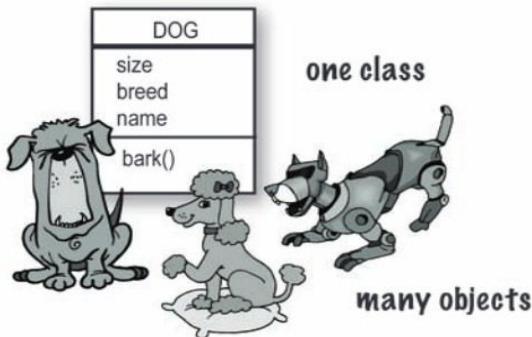
LIBRARY



CLASS and OBJECT

Class: Class is collection of instance variables and methods. Things an object knows about itself are called **instance variables**. Things an object can do are called **methods**.

Ex: Dog is the class name. The size, breed, name are instance variables, bark() is the method that Dog class consists of, as shown in the figure below.



A class is a blueprint for an object. It tells the virtual machine how to make an object of that particular type. Each object made from that class can have its own values for the instance variables of that class. For example, you might use the Button class to make dozens of different buttons, and each button might have its own color, size, shape, label, and so on.

Object: object is an instance of class.

The syntax to create an object is:

classname object=new classname();

The dot operator (.) gives you access to an object's state and behavior (instance variables and methods).

Dog d = new Dog();

d.bark();

d.size = 40;

METHODS

Java Methods (behaviors):-

- Methods are used to write the business logics of the project.
- Coding convention of method is method name starts with lower case letter if method contains more than one word then every inner word starts with uppercase letter.
Example:- post() , charAt() , toUpperCase() , compareToIgnoreCase().....etc

There are two types of methods

1. Instance method
 2. Static method
- Inside the class it is possible to declare any number of instance methods & static methods based on the developer requirement.
 - It will improve the reusability of the code and we can optimize the code.

Note :- Whether it is an instance method or static method the methods are used to provide business logics of the project.

instance method :-

```
void m1()    //instance method
{
    //body instance area
}
```

Note: - for the instance members memory is allocated during object creation hence access the instance members by using object(reference-variable).

Syntax:-

```
Void m1() { }           //instance method
Test t = new Test();
Objectname.instancemethod( );    //calling instance method
t.m1( );
```

static method:-

```
static void m1()           //instance method
{
    //body static method
}
```

Note: - for the static member's memory allocated during .class file loading hence access the static members by using class-name.

Syntax:-

```
Static void m2() { }       //static method
classname.staticmethod( ); // call static method by using class name
Test.m2( );
```

Syntax:-

[modifiers-list] return-Type Method-name (parameters list) throws Exception

Modifiers-list	----->	represent access permissions.---- → [optional]
Return-type	----->	functionality return value---- → [mandatory]
Method name	----->	functionality name ---- → [mandatory]
Parameter-list	----->	input to functionality ---- → [optional]
Throws Exception	----->	representing exception handling--- → [optional]

Example:-

```
public void m1()
private int m2(int a,int b)
```

Method Signature:-

Method Signature is nothing but name of the method and parameters list. Return type and modifiers list not part of a method signature.

Syntax:- Method-name(parameter-list)

Ex:- m1(int a)
 m1(int a,int b)

Every method contains two parts.

1. Method declaration
2. Method implementation (logic)

Ex:-

```
void m1()                         -----> method declaration
{
    Body (Business logic); -----> method implementation
}
```

Example-1 :- instance methods without arguments.

Instance methods are bounded with objects hence call the instance methods by using object name(reference variable).

```
class Test
{
    //instance methods
    void sravya()
    {
        System.out.println("Sravya");
    }
    void soft()
    {
        System.out.println("software solutions");
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        t.sravya();                         //calling of instance method by using object name [ t ]
        t.soft();                             //calling of instance method by using object name [ t ]
    }
}
```

Example-2:-instance methods with parameters.

- If the method is taking parameters at that situation while calling that method must provide parameter values then only that method will be executed.
- Parameters of methods is nothing but inputs to method.
- While passing parameters number of arguments and argument order is important.

void m1(int a)	-->t.m1(10);	-->valid
void m2(int a,int b)	-->t.m2(10,'a');	-->invalid
void m3(int a,char ch,float f)	-->t.m3(10,'a',10.6);	-->invalid
void m4(int a,char ch,float f)	-->t.m4(10,10,10.6);	-->invalid
void m5(int a,char ch,float f)	-->t.m3(10,'c');	-->invalid

```

class Test
{
    //instance methods
    void m1(int i,char ch) //local variables
    {
        System.out.println(i+"-----"+ch);
    }
    void m2(double d ,String str) //local variables
    {
        System.out.println(d+"-----"+str);
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1(10,'a');           //m1() method calling
        t.m2(10.2,"ratna");    //m2() method calling
    }
}

```

Example-3 :- static methods without parameters.

Static methods are bounded with class hence call the static members by using class name.
 class Test

```

{
    //static methods
    static void m1()
    {
        System.out.println("m1 static method");
    }
    static void m2()
    {
        System.out.println("m2 static method");
    }
    public static void main(String[] args)
    {
        Test.m1();
        Test.m2();
        //call the static method by using class name
        //call the static method by using class name
    }
}

```

Example -4 :-static methods with parameters.

```

class Test
{
    //static methods
    static void m1(String str,char ch,int a)           //local variables
    {

```

```

        System.out.println(str+"---"+ch+"---"+a);
    }
    static void m2(boolean b1,double d)           //local variables
    {
        System.out.println(b1+"---"+d);
    }
    public static void main(String[] args)
    {
        Test.m1("ratan",'a',10);               //static m1() calling by using class name
        Test.m2(true,10.5);                   //static m2() calling by using class name
    }
}

```

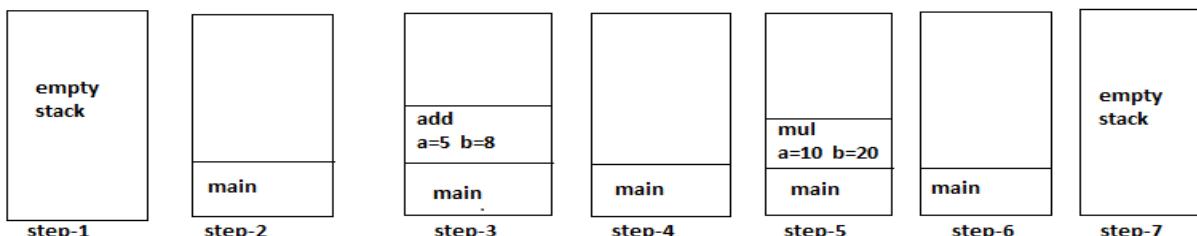
Stack Mechanism:-

- In java program execution starts from main method, just before program execution JVM creates one empty stack for that application.
- Whenever JVM calling particular method then that method entry and local variables of that method stored in stack memory.
- When the method exists, that particular method entry and local variables of that method are deleted from memory that memory becomes available to other called methods.
- Based on 2 & 3 the local variables are stored in stack memory and for these variables memory is allocated when method starts and memory is deleted when program ends.
- The intermediate calculations are stored in stack memory at final if all methods are completed that stack will become empty then that empty stack is destroyed by JVM just before program completes.
- The empty stack is created by JVM and at final empty stack is destroyed by JVM.

```

class Test
{
    void add(int a,int b)
    {
        System.out.println(a+b);
    }
    void mul(int a,int b)
    {
        System.out.println(a+b);
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.add(5,8);
        t.mul(10,20);
    }
}

```



CONSTRUCTORS

Agenda:

Constructors

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

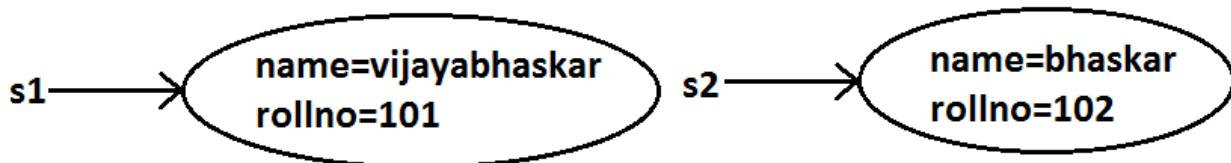
Constructors :

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.

- 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
    {
    }
}
```

- It is legal (but stupid) to have a method whose name is exactly same as class name.
- The only applicable modifiers for the constructors are public, default, private, protected.
- 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:

- For every class in java including abstract classes also constructor concept is applicable.
- If we are not writing at least one constructor then compiler will generate default constructor.
- 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:

- It is always no argument constructor.
- The access modifier of the default constructor is same as class modifier. (This rule is applicable only for public and default).
- 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 { }	}
class Test { void Test(){} }	public class Test { public Test() { super(); } }
class Test { Test(int i) {} }	class Test { Test() { 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:

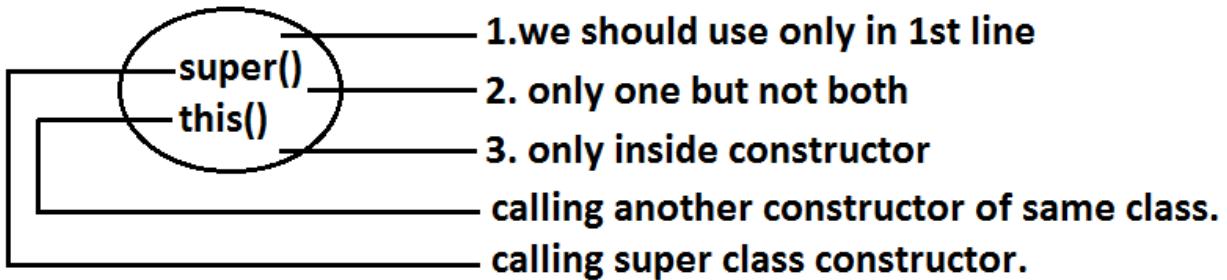
```
class Test
{
    public void methodOne()
    {
        super();
    }
}
```

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
}
}
```

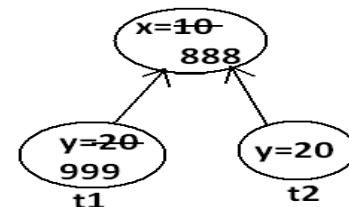
- Parent class constructor by default won't available to the Child. Hence Inheritance concept is not applicable for constructors and hence overriding concept also not applicable to the constructors. But constructors can be overloaded.
- We can take constructor in any java class including abstract class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

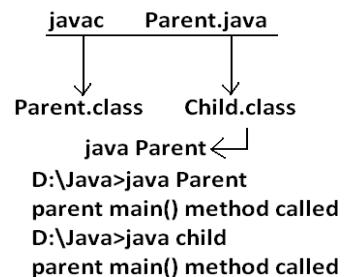
String() method is called

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.

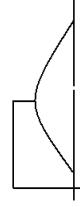
Example:

```
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:

```
javac Parent.java  
↓  
Parent.class Child.class  
java Parent ←  
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.

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.

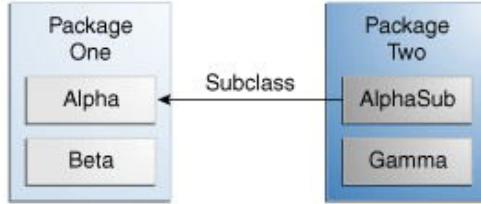
Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
<i>no modifier</i>	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
<i>no modifier</i>	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:

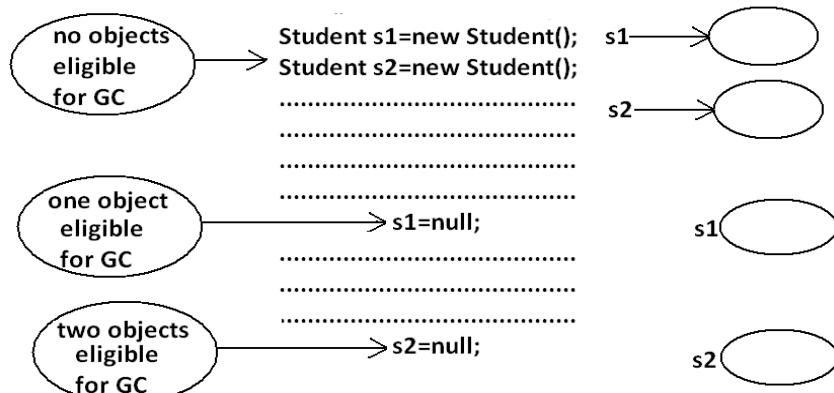
- Even though programmer is not responsible for destruction of objects but **it is always a good programming practice to make an object eligible for GC** if it is no longer required.
- An object is eligible for GC if and only if it does not have any references.

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.

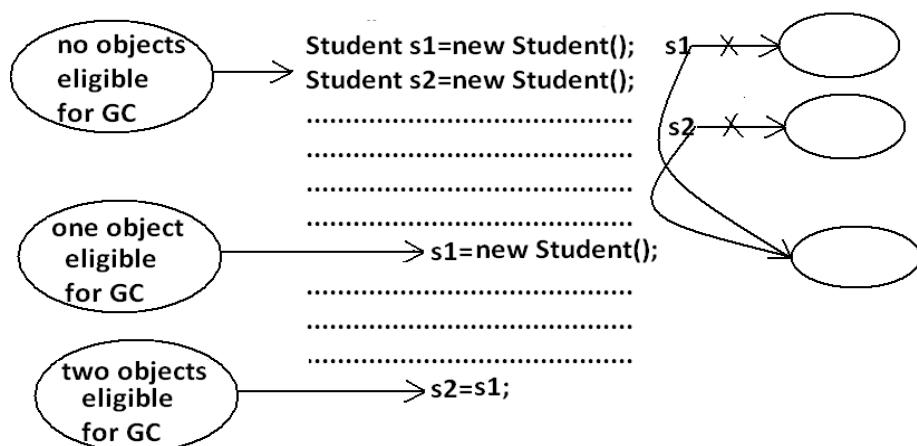
Example:



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:

two objects eligible for GC.

```

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

```

Example 2:

one object eligible for GC.

```

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:

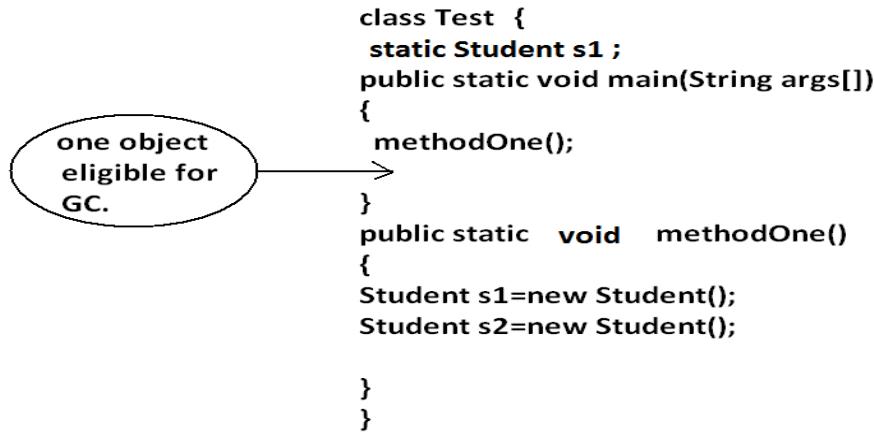
two objects eligible for GC.

```

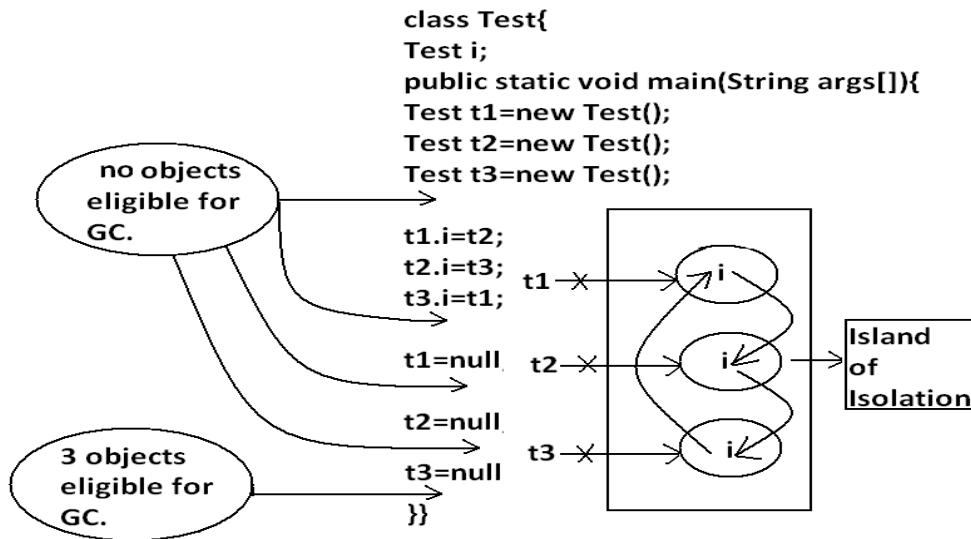
class Test
{
    public static void main(String args[])
    {
        methodOne();
    }
    public static Student methodOne()
    {
        Student s1=new Student();
        Student s2=new Student();
        return s1;
    }
}

```

Example 4:



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 expect 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()
    {
        System.out.println("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");
    }
}
```

```

}

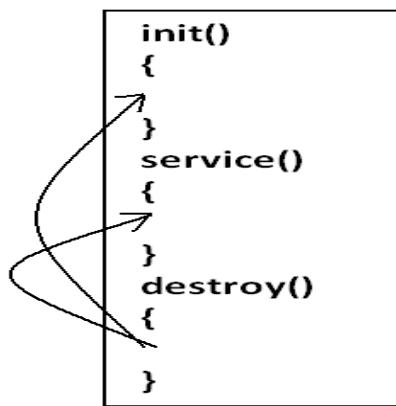
Output:
finalize() method called.
finalize() method called.
finalize() method called.
End of main.

```

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 expect 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 & sweep** 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.

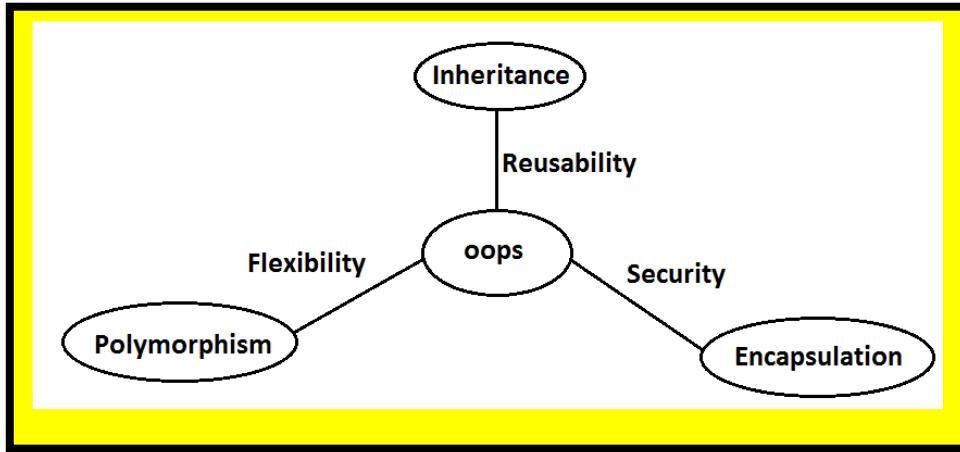
Example:

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:

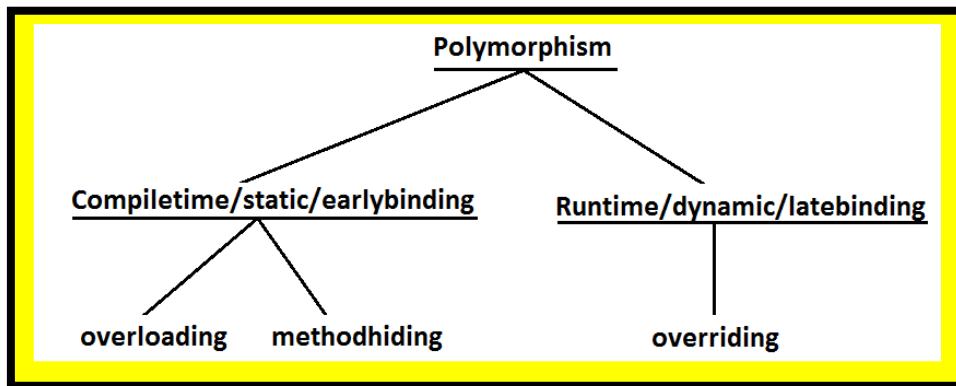
1. `abs(int)`
2. `abs(long)`
3. `abs(float)`

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:



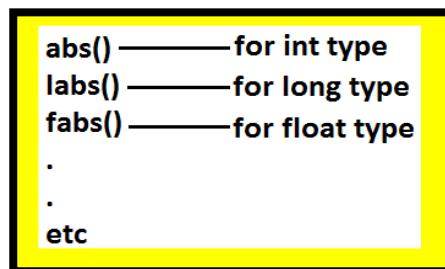
There are two types of polymorphism in java

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:-

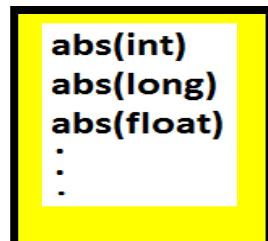
- 1) Two methods are said to be **overload**, if and only if both having the **same name** but **different argument types**.
- 2) In '**C**' language, we **can't take** 2 methods with the same name and different types. If there is a change in argument type, compulsory we should go for new method name.

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**.

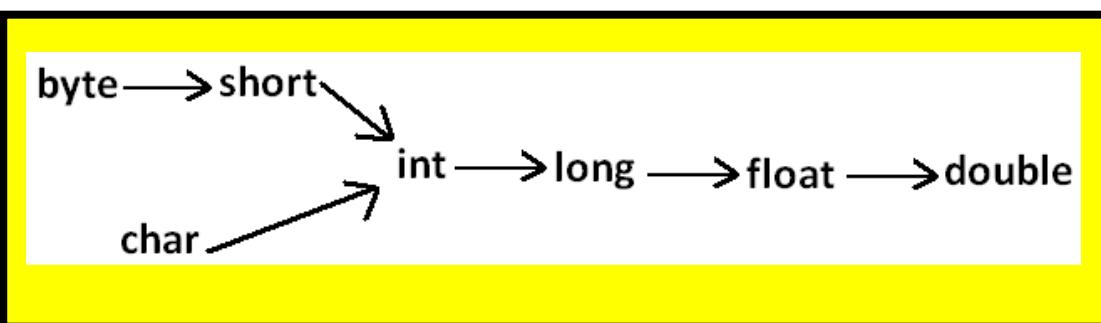
```
class Test
{
    public void methodOne()
    {
        System.out.println("no-arg method");
    }
    public void methodOne(int i)
    {
        System.out.println("int-arg method");
        //overloaded methods
    }
    public void methodOne(double d)
    {
        System.out.println("double-arg 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**.

Case 1: Automatic promotion in overloading.

- In overloading if compiler is unable to find the method with exact match we won't get any compile time error immediately.
- First compiler promotes the argument to the next level and checks whether the matched method is available or not. If it is available, then that method will be considered. If it is not available, then compiler promotes the argument once again to the next level. This process will be continued until all possible promotions still if the matched method is not available then we will get compile time error. This process is called automatic promotion in overloading.

The following are various possible automatic promotions in overloading.



Example:

```
class Test
{
    public void methodOne(int i)
    {
        System.out.println("int-arg method");
    }
    public void methodOne(float f) //overloaded methods
    {
        System.out.println("float-arg method");
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        //t.methodOne('a');//int-arg method
        //t.methodOne(10.1);//float-arg method
        t.methodOne(10.5); //C.E:cannot find symbol
    }
}
```

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);

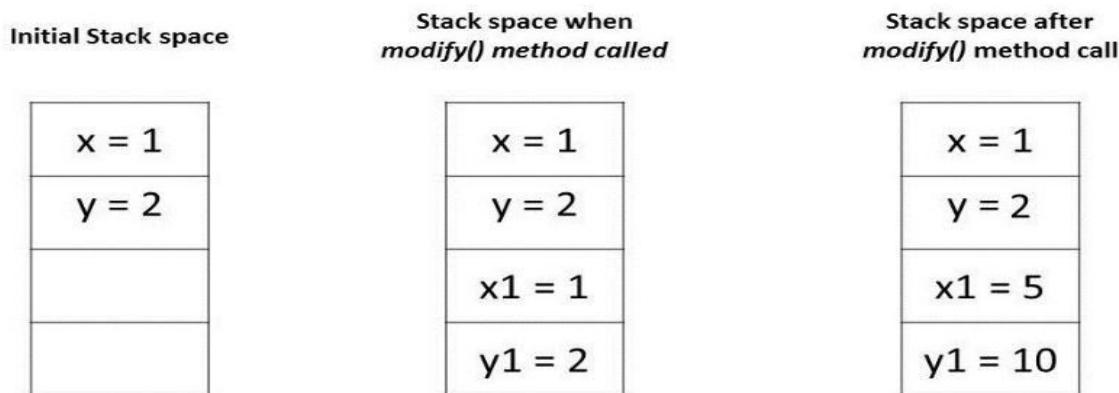
        modify(x,y);

        //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=15
        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

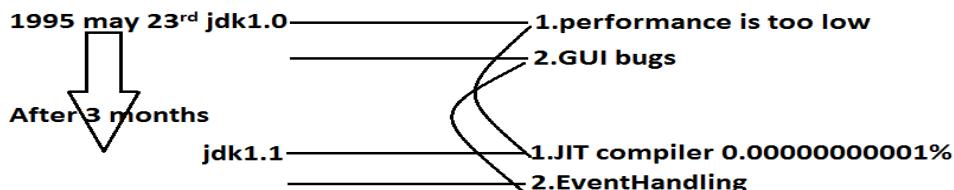
Agenda:

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
    {
    }
}
```

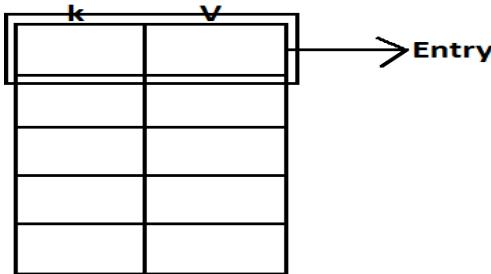
Example 3:

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

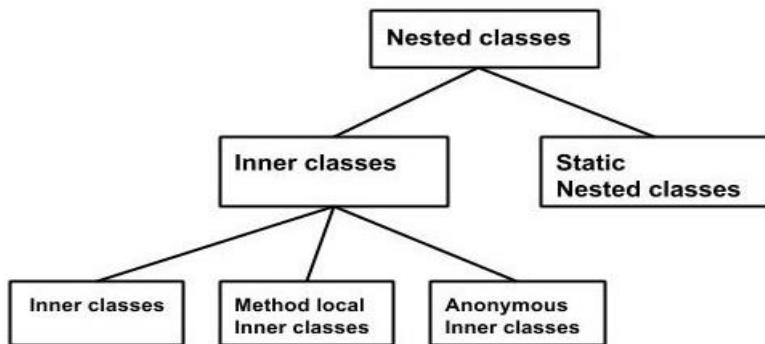
Inner classes:

- 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:

```
javac Outer.java
Outer.class      Outer$Inner.class
E:\scjp>java Outer
Exception in thread "main" java.lang.NoSuchMethodError: main
E:\scjp>java Outer$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(); } (or)
        i.methodOne();

        Outer.Inner i=new Outer().new Inner(); } (or)
        i.methodOne();

        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

```

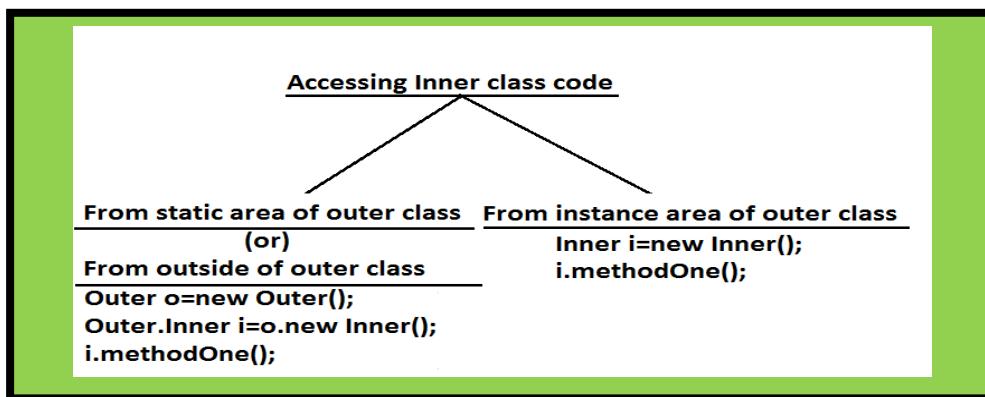
Accessing inner class code from outside of outer class:

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:

```
class A {  
    class B {  
        class C {  
            public void methodOne() {  
                System.out.println("Inner most method");  
            }  
        }  
    }  
  
    public static void main(String ar[]) {  
        A a = new A();  
        or  
        A.B b = a.new B();  
        or  
        A.B.C c = b.new C();  
        or  
        A.B.C c = b.new C();  
        c.methodOne();  
    }  
}
```

The diagram illustrates four different ways to create an object of class C, which is nested within class B, which is itself nested within class A. The code snippet shows these creation methods:

- A.B b = new A().new B(); (highlighted in green)
- A.B.C c = new A().new B().new C(); (highlighted in purple)
- A.B.C c = b.new C(); (highlighted in blue)
- new A().new B().new C().methodOne(); (highlighted in blue)

Red arrows point from the first two lines of code to the green box. Purple arrows point from the third line to the purple box. A blue arrow points from the fourth line to the blue box.

String manipulations

Java.lang.String:-

String is used to represent group of characters or character array enclosed with in the double quotes.

Case 1:-String vs StringBuffer

String & StringBuffer both classes are final classes present in java.lang package.

Case 2:-String vs StringBuffer

We are able to create String object in two ways.

- 1) Without using new operator

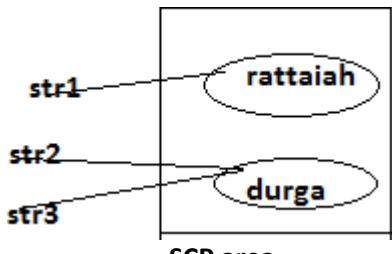
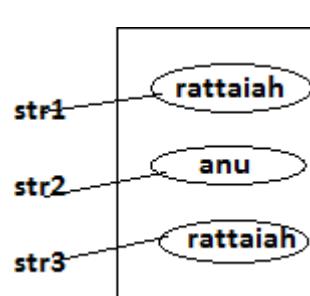
```
String str="krishna";
```

- 2) By using new operator

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

We are able to create StringBuffer object only one approach by using new operator.

```
StringBuffer sb = new StringBuffer("krify technologies");
```

<u>Creating a string object without using new operator :-</u>	<u>Creating a string object by using new operator</u>
<ul style="list-style-type: none"> ➤ When we create String object without using new operator the objects are created in SCP (String constant pool) area. ➤ <pre>String str1="rattaiah"; String str2="durga"; String str3="durga";</pre>  ➤ When we create object in SCP area then just before object creation it is always checking previous objects. ➤ If the previous object is available with the same content then it won't create new object that reference variable pointing to existing object. ➤ If the previous objects are not available then JVM will create new object. ➤ SCP area does not allow duplicate objects 	<ul style="list-style-type: none"> ➤ Whenever we are creating String object by using new operator the object created in heap area. <pre>String str1=new String("rattaiah"); String str2 = new String("anu"); String str3 = new String("rattaiah");</pre>  ➤ When we create object in Heap area instead of checking previous objects it directly creates objects. ➤ Heap memory allows duplicate objects

Example:-

```
class Test
{
    public static void main(String[] args)
    {
        //two approaches to create a String object
        String str1 = "krish";
        System.out.println(str1);
        String str2 = new String("anu");
        System.out.println(str2);

        //one approach to create StringBuffer Object (by using new operator)
        StringBuffer sb = new StringBuffer("krifysoft");
        System.out.println(sb);
    }
}
```

Case 3:- String

java.lang.String vs java.lang.StringBuffer:-

- String is **immutability class** it means once we are creating String objects it is not possible to perform modifications on existing object. (String object is fixed object)
- StringBuffer is a **mutability class** it means once we are creating StringBuffer objects on that existing object it is possible to perform modification.

Example :-

```
class Test
{
    public static void main(String[] args)
    {
        //immutability class (modifications on existing content not allowed)
        String str="ratan";
        str.concat("soft");
        System.out.println(str);

        //mutability class (modifications on existing content possible)
        StringBuffer sb = new StringBuffer("anu");
        sb.append("soft");
        System.out.println(sb);
    }
}
```

Java.lang.String class methods:-

Table 1: Some Most Commonly Used String Methods

Method Call	Task performed
s2 = s1.toLowerCase;	Converts the string s1 to all lowercase
s2 = s1.toUpperCase;	Converts the string s1 to all Uppercase
s2 = s1.replace('x', 'y');	Replace all appearances of x with y
s2 = s1.trim();	Remove white spaces at the beginning and end of the string s1
s1.equals(s2)	Returns 'true' if s1 is equal to s2
s1.equalsIgnoreCase(s2)	Returns 'true' if s1 = s2, ignoring the case of characters
s1.length()	Gives the length of s1
s1.charAt(n)	Gives nth character of s1
s1.compareTo(s2)	Returns negative if s1 < s2, positive if s1 > s2, and zero if s1 is equal s2
s1.concat(s2)	Concatenates s1 and s2
s1.substring(n)	Gives substring starting from n th character
s1.substring(n, m)	Gives substring starting from n th character up to m th (not including m th)
String.valueOf(p)	Creates a string object of the parameter p (simple type or object)
p.toString()	Creates a string representation of the object p
s1.indexOf('x')	Gives the position of the first occurrence of 'x' in the string s1
s1.indexOf('x', n)	Gives the position of 'x' that occurs after nth position in the string s1
String.valueOf(Variable)	Converts the parameter value to string representation

1) CompareTo(): -

- By using compareTo() we are comparing two strings character by character, such type of checking is called lexicographically checking or dictionary checking.
- compareTo() is return type is integer and it returns three values
 - a. zero ----- > if both String are **equal**
 - b. positive----->if first string first character Unicode value is **bigger than** second String first character Unicode value then it returns positive.
 - c. Negative ---> if first string first character Unicode value is **smaller than** second string first character Unicode value then it returns negative.
- compareTo() method comparing two string with **case sensitive**.

```
class Test
{
    public static void main(String... ratan)
    {
        String str1="ratan";
        String str2="Sravya";
        String str3="ratan";
        System.out.println(str1.compareTo(str2));//14
```

```

        System.out.println(str1.compareTo(str3));//0
        System.out.println(str2.compareTo(str1));//-13
        System.out.println("ratan".compareTo("RATAN"));//+ve
    }
}

```

Difference between `length()` method and `length` variable:-

- `length` variable used to find length of the Array.
- `length()` is method used to find length of the String.

Example :-

```

int [] a={10,20,30};
System.out.println(a.length);      //3

String str="rattaiah";
System.out.println(str.length());  //8

```

`charAt(int)`: By using above method we are able to extract the character from particular index position.

s1.charAt(n)

`trim()`: `trim()` is used to remove the trail and leading spaces. This method always used for memory saver.

s1.trim();

```

class Test
{
    public static void main(String[ ] args)
    {
        //charAt() method
        String str="ratan";
        System.out.println(str.charAt(1));
        //System.out.println(str.charAt(10));StringIndexOutOfBoundsException
        char ch="ratan".charAt(2);
        System.out.println(ch);
        //trim()
        String ss="ratan";
        System.out.println(ss.length());//7
        System.out.println(ss.trim());//ratan
        System.out.println(ss.trim().length());//5
    }
}

```

`replace()`:

`replace()` method used to replace the String or character.

`toUpperCase()` & `toLowerCase()`:

The above methods are used to convert lower case to upper case & upper case to lower case.

Example:-

```

class Test
{
    public static void main(String[] args)
    {
        String str="rattaiah how r u";
        System.out.println(str.replace('a','A'));//rAttAiAh
        System.out.println(str.replace("how","who"));//rattaiah who r u
        String str1="Sravya software solutions";
        System.out.println(str1);
        System.out.println(str1.replace("software","hardware"));// Sravya hardware solutions
        String str="ratan HOW R U";
        System.out.println(str.toUpperCase());
        System.out.println(str.toLowerCase());
        System.out.println("RATAN".toLowerCase());
        System.out.println("soft".toUpperCase());
    }
}

```

StringBuffer class methods:-

Table 2: Commonly Used StringBuffer Methods

Method	Task
s1.setCharAt(n, 'x')	Modifies the nth character to x
s1.append(s2)	Appends the string s2 to s1 at the end
s1.insert(n, s2)	Inserts the string s2 at the position n of the string s1
s1.setLength(n)	Sets the length of the string s1 to n. If n<s1.length() s1 is truncated. If n>s1.length() zeros are added to s1

reverse():-

```

class Test
{
    public static void main(String[] args)
    {
        StringBuffer sb=new StringBuffer("rattaiah");
        System.out.println(sb);
        System.out.println(sb.delete(1,3));
        System.out.println(sb);
        System.out.println(sb.deleteCharAt(1));
        System.out.println(sb.reverse());
    }
}

```

append():-

By using this method we can append the any values at the end of the string

Ex:-

```

class Test
{
    public static void main(String[] args)
    {
        StringBuffer sb=new StringBuffer("rattaiah");
        String str=" salary ";
        int a=60000;
        sb.append(str);
        sb.append(a);
    }
}

```

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

Insert():-
```

By using above method we are able to insert the string any location of the existing string.

```
class Test
{
    public static void main(String[] args)
    {
        StringBuffer sb=new StringBuffer("ratan");
        sb.insert(0,"hi ");
        System.out.println(sb);
    }
}
```

indexOf() and lastIndexOf():-

Ex:-

```
class Test
{
    public static void main(String[] args)
    {
        StringBuffer sb=new StringBuffer("hi ratan hi");
        int i;
        i=sb.indexOf("hi");
        System.out.println(i);
        i=sb.lastIndexOf("hi");
        System.out.println(i);
    }
}
```

replace():-

```
class Test
{
    public static void main(String[] args)
    {
        StringBuffer sb=new StringBuffer("hi ratan hi");
        sb.replace(0,2,"oy");
        System.out.println("after replaceing the string:-"+sb);
    }
}
```

Command Line Arguments

Main Method:-

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

String[] args → used to take command line arguments(the arguments passed from command prompt)

The arguments which are passed from command prompt is called **command line arguments**. We are passing command line arguments at the time program execution.

Example-1 :-

```
class Test
{
    public static void main(String[ ] leela)
    {
        System.out.println(leela[0]+" "+leela[1]);//printing command line arguments
        System.out.println(leela[0]+leela[1]);
        int a = Integer.parseInt(leela[0]); //conversion of String-int
        double d = Double.parseDouble(leela[1]); //conversion of String-double
        System.out.println(a+d);
    }
}
D:>java Test 100 200
100 200
100200
300.0
```

Example 2: To provide the command line arguments with spaces , then take that command line argument with in **double quotes**.

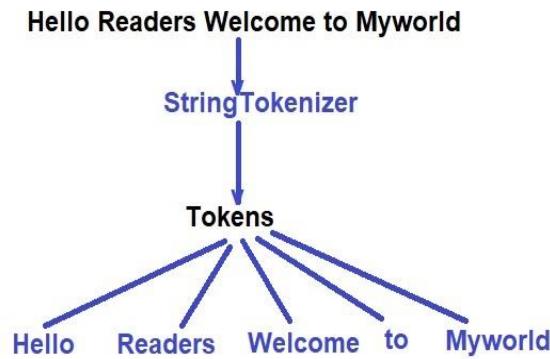
```
class Test
{
    public static void main(String[ ] leela)
    {
        System.out.println(leela[0]);
        System.out.println(leela[1]);
    }
}
D:>java Test corejava leela
corejava
leela
D:>java Test core java leela
core
java
D:>java Test "core java" leela
core java
leela
```

According to the Oracle, [***StringTokenizer in Java***](#) is a class that allows an application to break a string into tokens.

With the help of different StringTokenizer constructors and methods, we can break a string into tokens.

What is StringTokenizer in Java?

StringTokenizer class is used for creating tokens in Java. It allows an application to break or split into small parts. Each split string part is called *Token*.



A *StringTokenizer in Java*, object keeps the string in the present position as it is to be tokenized. By taking a substring of the string a token can return that utilize to make the StringTokenizer protest.

Table 1. Methods of the StringTokenizer Class

Method	Description
Constructors	
<code>StringTokenizer(String str)</code>	Constructs a string tokenizer for the specified string str; returns a reference the new object
<code>StringTokenizer(String str, String delim)</code>	Constructs a string tokenizer for the specified string str using delim as the delimiter set
<code>StringTokenizer(String str, String delim, boolean returnTokens)</code>	Constructs a string tokenizer for the specified string str using delim as the delimiter set; returns the tokens with the string if returnTokens is true
Instance Methods	
<code>countTokens()</code>	returns an int equal to the number of times that this tokenizer's <code>nextToken()</code> method can be called before it generates an exception
<code>hasMoreTokens()</code>	returns a boolean value: <code>true</code> if there are more tokens available from this tokenizer's string, <code>false</code> otherwise
<code>nextToken()</code>	returns a String equal to the next token from this string tokenizer
<code>nextToken(String delim)</code>	returns a String equal to the next token from this string tokenizer using delim as a new delimiter set

Java Random Class

- Random class is part of java.util package.
- An instance of java Random class is used to generate random numbers.
- This class provides several methods to generate random numbers of type integer, double, long, float etc.
- Random number generation algorithm works on the seed value. If not provided, seed value is created from system nano time.
- If two Random instances have same seed value, then they will generate same sequence of random numbers.
- Java Random class is thread-safe, however in multithreaded environment it's advised to use `java.util.concurrent.ThreadLocalRandom` class.
- Random class instances are not suitable for security sensitive applications, better to use `java.security.SecureRandom` in those cases.

Java Random Constructors

Java Random class has two constructors which are given below:

1. `Random()`: creates new random generator
2. `Random(long seed)`: creates new random generator using specified seed

Java Random Class Methods

Let's have a look at some of the methods of java Random class.

1. `nextBoolean()`: This method returns next pseudorandom which is a boolean value from random number generator sequence.
2. `nextDouble()`: This method returns next pseudorandom which is double value between 0.0 and 1.0.
3. `nextFloat()`: This method returns next pseudorandom which is float value between 0.0 and 1.0.
4. `nextInt()`: This method returns next int value from random number generator sequence.
5. `nextInt(int n)`: This method return a pseudorandom which is int value between 0 and specified value from random number generator sequence

Java Random Example

Let's have a look at the below java Random example program.

```
import java.util.Random;
//Java Random Number Example Program

public class RandomNumberExample {

    public static void main(String[] args) {
        //initialize random number generator
        Random random = new Random();
```

```
//generates boolean value  
System.out.println(random.nextBoolean());  
  
//generates double value  
System.out.println(random.nextDouble());  
  
//generates float value  
System.out.println(random.nextFloat());  
  
//generates int value  
System.out.println(random.nextInt());  
  
//generates int value within specific limit  
System.out.println(random.nextInt(20));  
}  
}
```

Output of the above program is:

```
false  
0.30986869120562854  
0.6210066  
-1348425743  
18
```

Important points:

Random class:

==>The Random class implements a *random number generator*, which produces sequences of numbers that appear to be random.

==>To generate random integers , we construct an object of Random class , and then apply to the nextInt() method.

For example, the call generator.nextInt(6) gives us a random number between 0 and 5

Example 1: Testing a Random Integer for Negativity

```
//java program to test a Random integer for negativity
import java.util.Random;
class Testnegativity
{
    public static void main(String[] args)
    {
        Random r=new Random();
        int n=r.nextInt();
        System.out.println("n="+n);
        if(n<0)
        {
            System.out.println("****n<0");
        }
        System.out.println("Goodbye");
    }
}
student@student-HP-245-G6-Notebook-PC:~$ javac Testnegativity.java
student@student-HP-245-G6-Notebook-PC:~$ java Testnegativity
n=1288870215
Goodbye
student@student-HP-245-G6-Notebook-PC:~$ javac Testnegativity.java
student@student-HP-245-G6-Notebook-PC:~$ java Testnegativity
n=-740627638
****n<0
Goodbye
```

This program uses a random number generator to generate a random integer. It then reports whether the integer is negative:

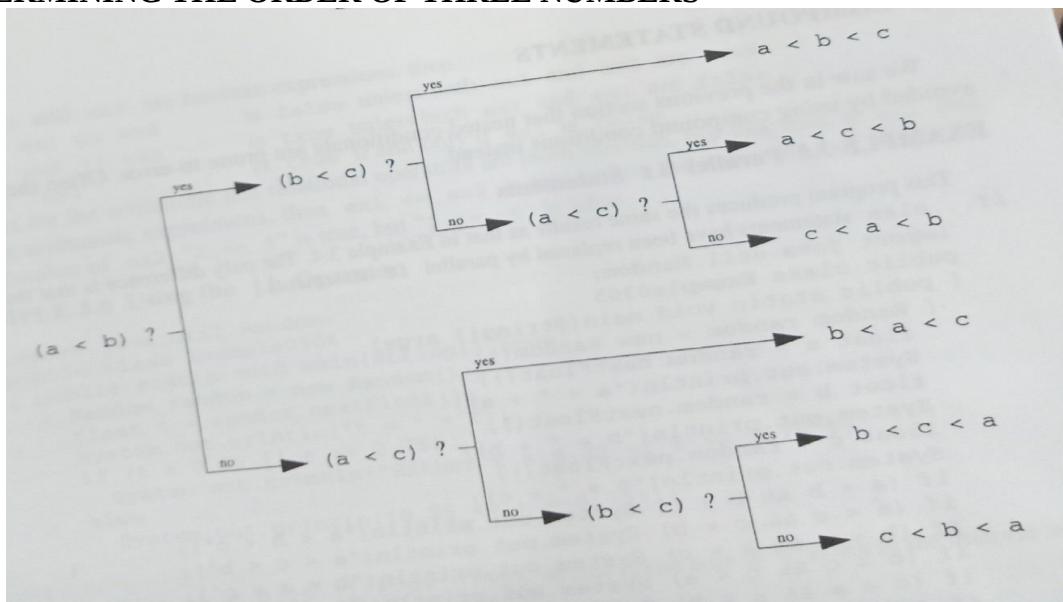
EXAMPLE 2: TESTING TWO RANDOM INTEGERS FOR THEIR MINIMUM USING if else STATEMENT:

```
//program to test 2 Random integers for their minimum
import java.util.Random;
class Testmin
{
    public static void main(String[] args)
    {
        Random r=new Random();
        int m=r.nextInt();
        System.out.println("m="+m);
        int n=r.nextInt();
        System.out.println("n="+n);
        if(m<n)
            System.out.println("The minimum is"+m);
        else
            System.out.println("The minimum is"+n);
    }
}
student@student-HP-245-G6-Notebook-PC:~$ javac Testmin.java
student@student-HP-245-G6-Notebook-PC:~$ java Testmin
m=593905876
n=20943430
The minimum is20943430
student@student-HP-245-G6-Notebook-PC:~$ javac Testmin.java
student@student-HP-245-G6-Notebook-PC:~$ java Testmin
m=-1626594304
n=431772941
The minimum is-1626594304
```

EXAMPLE 3: CHOOSING FROM FOUR ALTERNATIVES

```
//Java program for choosing among 4 alternatives
import java.util.Random;
class Testchoose
{
    public static void main(String[] args)
    {
        Random r=new Random();
        double t=r.nextDouble();
        System.out.println("t="+t);
        if(t<0.25)
            System.out.println("0<=t<1/4");
        else if(t<0.5)
            System.out.println("1/4<=t<1/2");
        else if(t<0.75)
            System.out.println("1/2<=t<3/4");
        else
            System.out.println("3/4<=t<1");
    }
}
student@student-HP-245-G6-Notebook-PC:~$ javac Testchoose.java
student@student-HP-245-G6-Notebook-PC:~$ java Testchoose
t=0.06985721594071936
0<=t<1/4
student@student-HP-245-G6-Notebook-PC:~$ javac Testchoose.java
student@student-HP-245-G6-Notebook-PC:~$ java Testchoose
t=0.7119579914650608
1/2<=t<3/4
```

EXAMPLE 4:
DETERMINING THE ORDER OF THREE NUMBERS



```

/*java program that uses pairwise comparisons to determine the increasing
order of three randomly generated real numbers using nested if*/
import java.util.Random;
class Nestedif
{
    public static void main(String[] args)
    {
        Random r=new Random();
        float a=r.nextFloat();
        System.out.println("a="+a);
        float b=r.nextFloat();
        System.out.println("b="+b);
        float c=r.nextFloat();
        System.out.println("c="+c);
        if(a<b)
            if(b<c)
                System.out.println("a<b<c");
            else if(a<c)
                System.out.println("a<c<b");
            else
                System.out.println("c<a<b");
        else
            if(a<c)
                System.out.println("b<a<c");
            else if(b<c)
                System.out.println("b<c<a");
            else
                System.out.println("c<b<a");
    }
}
student@student-HP-245-G6-Notebook-PC:~$ javac Order.java
student@student-HP-245-G6-Notebook-PC:~$ java Order
a=0.36980355
b=0.34350783
c=0.6177115
b<a<c
  
```

EXAMPLE 5 PARALLEL IF STATEMENTS:

```
/*java program that uses pairwise comparisons to determine the increasing
order of three randomly generated real numbers using parallel if*/
import java.util.Random;
class Parallelif
{
    public static void main(String[] args)
    {
        Random r=new Random();
        float a=r.nextFloat();
        System.out.println("a="+a);
        float b=r.nextFloat();
        System.out.println("b="+b);
        float c=r.nextFloat();
        System.out.println("c="+c);
        if(a<b && b<c)
            System.out.println("a<b<c");
        if(a<c && c<b)
            System.out.println("a<c<b");
        if(b<a && a<c)
            System.out.println("b<a<c");
        if(b<c && c<a)
            System.out.println("b<c<a");
        if(c<a && a<b)
            System.out.println("c<a<b");
        if(c<b && b<a)
            System.out.println("c<b<a");
    }
}
student@student-HP-245-G6-Notebook-PC:~$ javac Parallelif.java
student@student-HP-245-G6-Notebook-PC:~$ java Parallelif
a=0.65420145
b=0.7130023
c=0.8444757
a<b<c
```

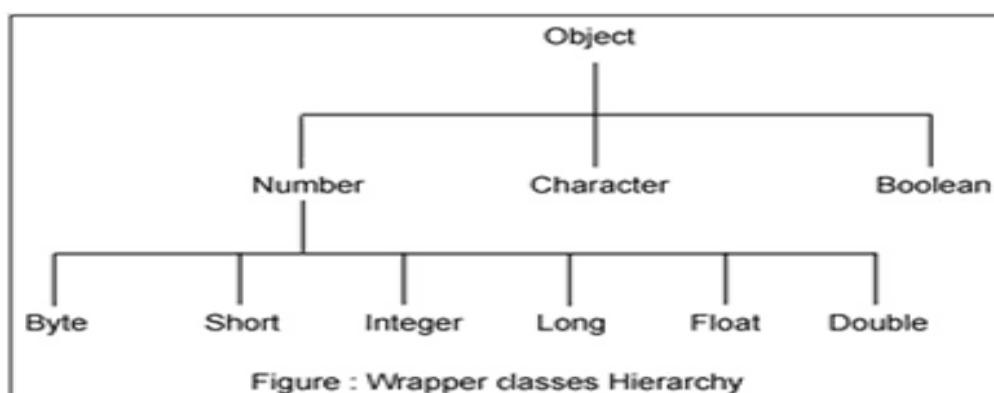
EXAMPLE 6:
USING the || OPERATOR:

```
//java program using || operator
import java.util.Random;
class Logicalor
{
    public static void main(String[] args)
    {
        Random r= new Random();
        float t=r.nextFloat();
        System.out.println("t="+t);
        if(t<0.25 || t>=0.75)
            System.out.println("Either t<0.25 or t>=0.75");
        else
            System.out.println("0.25<=t<0.75");
    }
}
student@student-HP-245-G6-Notebook-PC:~$ javac Logicalor.java
student@student-HP-245-G6-Notebook-PC:~$ java Logicalor
t=0.90939236
Either t<0.25 or t>=0.75
student@student-HP-245-G6-Notebook-PC:~$ java Logicalor
t=0.12464261
Either t<0.25 or t>=0.75
```

Wrapper classes

- Java is an Object oriented programming language so represent everything in the form of the object, but java supports 8 primitive data types these all are not part of object.
- To represent 8 primitive data types in the form of object form we required 8 java classes these classes are called wrapper classes.
- All wrapper classes present in the **java.lang** package and these all classes are **immutable** classes.

Wrapper classes hierarchy:-



Wrapper classes constructors:-

```
Integer i = new Integer(10);
Integer i1 = new Integer("100");
Float f1= new Float(10.5);
Float f1= new Float(10.5f);
Float f1= new Float("10.5");
Character ch = new Character('a');
```

datatypes	wrapper-class	Constructor argument
<i>byte</i>	<i>Byte</i>	<i>byte, String</i>
<i>short</i>	<i>Short</i>	<i>short, String</i>
<i>int</i>	<i>Integer</i>	<i>int, String</i>
<i>long</i>	<i>Long</i>	<i>long, String</i>
<i>float</i>	<i>Float</i>	<i>double, float, String</i>
<i>double</i>	<i>Double</i>	<i>double, String</i>
<i>char</i>	<i>Character</i>	<i>char</i>
<i>boolean</i>	<i>Boolean</i>	<i>boolean, String</i>

Note :- To create wrapper objects all most all wrapper classes contain two constructors but *Float* contains **three** constructors(*float, double, String*) & *char* contains **one** constructor(*char*).

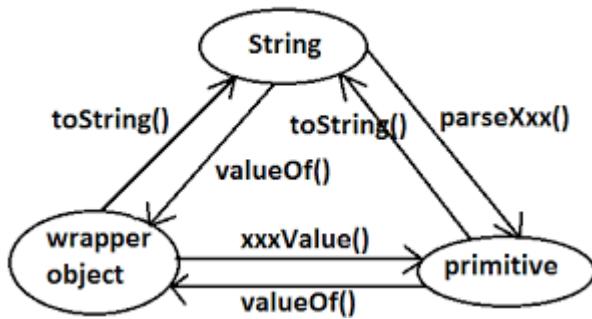
The most common methods of the wrapper class are

1. `valueOf()`
2. `xxxValue()-----intValue(), floatValue(), doubleValue() etc`
3. `toString()`
4. `parseXxx()-----parseInt(), parseFloat() etc`

1. `valueOf()`:-

in java we are able to create wrapper object in two ways.

- a) *By using constructor approach*
 - b) *By using `valueOf()` method*
- ✓ `valueOf()` method is used to create wrapper object just it is alternate to constructor approach and it a static method present in wrapper classes.



Example:-

```
class Test
{
    public static void main(String[] args)
    {
        //constructor approach to create wrapper object
        Integer i1 = new Integer(100);
        System.out.println(i1);

        Integer i2 = new Integer("100");
        System.out.println(i2);

        //valueOf() method to create Wrapper object
        Integer a1 = Integer.valueOf(10);
        System.out.println(a1);

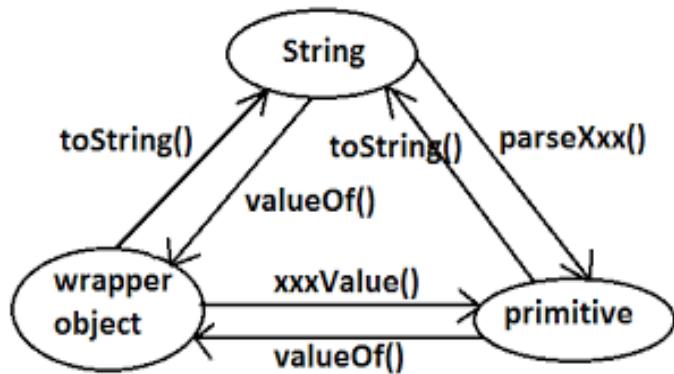
        Integer a2 = Integer.valueOf("1000");
        System.out.println(a2);
    }
}
```

Example :-conversion of primitive to String.

```
class Test
{
    public static void main(String[] args)
    {
        int a=100;
        int b=200;
        System.out.println(a+b);

        //primitive to String object
        String str1 = String.valueOf(a);
        String str2 = String.valueOf(b);
        System.out.println(str1+str2);
    }
}
```

2. xxxValue():- It is used to convert wrapper object into corresponding primitive value.



The most common methods of the Integer wrapper class related to `xxxValue()` are summarized in below table.

METHOD	PURPOSE
<code>byteValue()</code>	returns the value of this Integer as a byte
<code>doubleValue()</code>	returns the value of this Integer as a double
<code>floatValue()</code>	returns the value of this Integer as a float
<code>intValue()</code>	returns the value of this Integer as an int
<code>shortValue()</code>	returns the value of this Integer as a short
<code>longValue()</code>	returns the value of this Integer as a long

Example:-

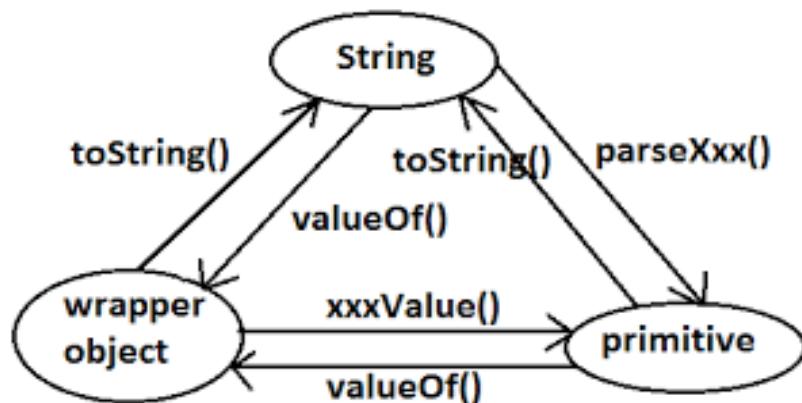
```
class Test
{
    public static void main(String[] args)
    {
        //valueOf() method to create Wrapper object
        Integer a1 = Integer.valueOf(10);
        System.out.println(a1);
        Integer a2 = Integer.valueOf("1000");
        System.out.println(a2);

        //xxxValue() [wrapper object into primitive value]
        int x1 = a1.intValue();
        byte x2 = a1.byteValue();
        double x3 = a1.doubleValue();
        System.out.println("int value=" +x1);
        System.out.println("byte value=" +x2);
        System.out.println("double value=" +x3);
    }
}
```

3. *toString()*:-

- *toString()* method present in Object class it returns class-name@hashcode.
- String, StringBuffer classes are overriding *toString()* method it returns content of the objects.
- All wrapper classes overriding *toString()* method to return content of the wrapper class objects.

METHOD	PURPOSE
<i>toString(i)</i>	<i>returns a new String object representing the integer i</i>



Example :-

```
class Test
{
    public static void main(String[] args)
    {
        Integer i1 = new Integer(100);
        System.out.println(i1);
        System.out.println(i1.toString());

        Integer i2 = new Integer("1000");
        System.out.println(i2);
        System.out.println(i2.toString());

        Integer i3 = new Integer("ten");//java.lang.NumberFormatException
        System.out.println(i3);
    }
}
```

- In above example for the integer constructor we are passing “1000” value in the form of String it is automatically converted into Integer format.
- In above example for the integer constructor we are passing “ten” in the form of String but this String is unable to convert into integer format it generate exception **java.lang.NumberFormatException**.

Example:-conversion of wrapper to String by using `toString()` method

```
class Test
{
    public static void main(String[] args)
    {
        Integer i1 = new Integer(100);
        Integer i2 = new Integer("1000");
        System.out.println(i1+i2); //1100
        //conversion [wrapper object - String]
        String str1 = i1.toString();
        String str2 = i2.toString();
        System.out.println(str1+str2); //1001000
    }
}
```

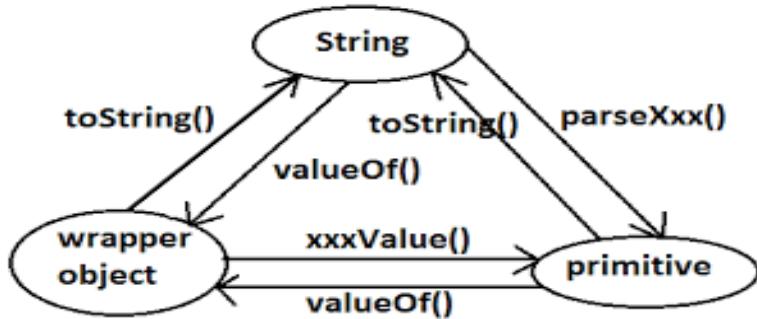
Example:-

- In java we are able to call `toString()` method only on reference type but not primitive type.
- If we are calling `toString()` method on primitive type then compiler generate error message.

```
class Test
{
    public static void main(String[ ] args)
    {
        Integer i1 = Integer.valueOf(100);
        System.out.println(i1);
        System.out.println(i1.toString());
        int a=100;
        System.out.println(a);
        //System.out.println(a.toString()); error:-int cannot be dereferenced
    }
}
```

4. parseXxx():- it is used to convert String into corresponding primitive value & it is a static method present in wrapper classes.

METHOD	PURPOSE
<code>parseInt(s)</code>	returns a signed decimal integer value equivalent to string s



Example :-

```

class Test
{
    public static void main(String[] args)
    {
        String str1="100";
        String str2="100";
        System.out.println(str1+str2);
        //parseXXX() conversion of String to primitive type
        int a1 = Integer.parseInt(str1);
        float a2 = Float.parseFloat(str2);
        System.out.println(a1+a2);
    }
}
  
```

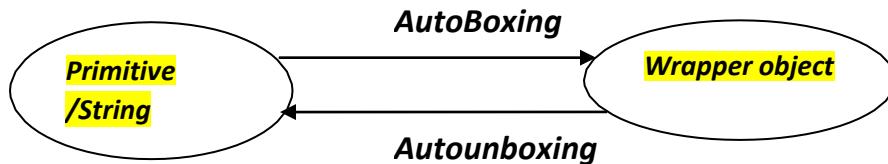
EXAMPLES:

- 1) **primitive ----->Wrapper Object**
`Integer i = Integer.valueOf(100);`
- 2) **wrapper object---- > primitive**
`byte b = i.byteValue();`
- 3) **String value ----> primitive**
`String str="100";
int a = Integer.parseInt(str);`
- 4) **primitive value -- > String Object**
`int a=100;
int b=200;
String s1 = String.valueOf(a);
String s2 = String.valueOf(b);
System.out.println(s1+s2); //100200`
- 5) **String value ---->Wrapper object**
`Integer i =Integer.valueOf("1000");`
- 6) **wrapper object--->String object**
`Integer i = new Integer(1000);
String s = i.toString();`

Autoboxing and Autounboxing:- (introduced in the 1.5 version)

- Up to 1.4 version to convert **primitive/String** into **Wrapper object** we are having two approaches
 - **Constructor approach**
 - **valueOf() method**
- Automatic conversion of primitive to wrapper object is called **autoboxing**.
- Automatic conversion of wrapper object to primitive is called **autounboxing**.

Automatic conversion of the primitive to wrapper and wrapper to the primitive:-



Example:-

```
class Test
{
    public static void main(String[] args)
    {
        //autoboxing [primitive - wrapper object]
        Integer i = 100;
        System.out.println(i);
        System.out.println(i.toString());

        //autounboxing [wrapper object - primitive]
        int a = new Integer(100);
        System.out.println(a);
    }
}
```

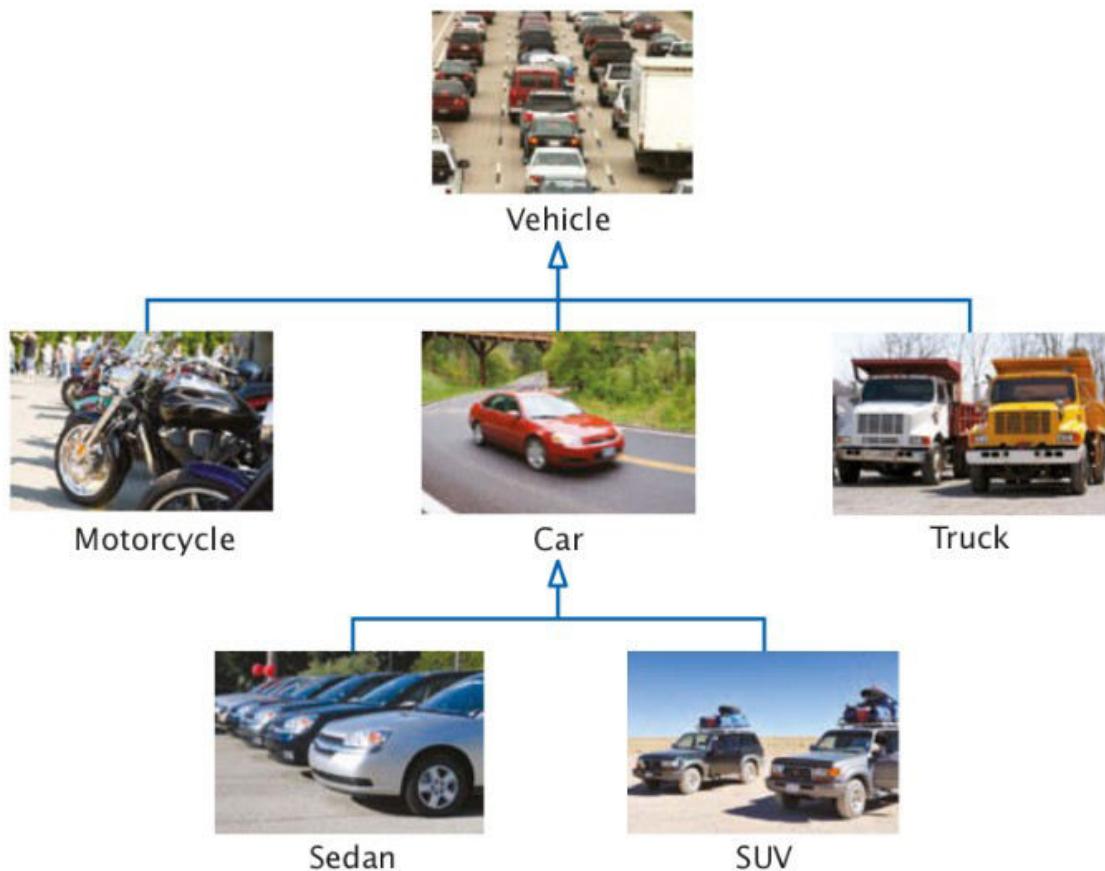
CHAPTER-3

OOPS CONCEPTS

POLYMORPHISM

ENCAPSULATION

INHERITENCE



Java OOPs Concepts

Object-Oriented Programming is a paradigm that provides many concepts, such as **inheritance**, **data binding**, **polymorphism**, etc.

Simula is considered the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.

Smalltalk is considered the first truly object-oriented programming language

The popular object-oriented languages are [Java](#), [C#](#), [PHP](#), [Python](#), [C++](#), etc.

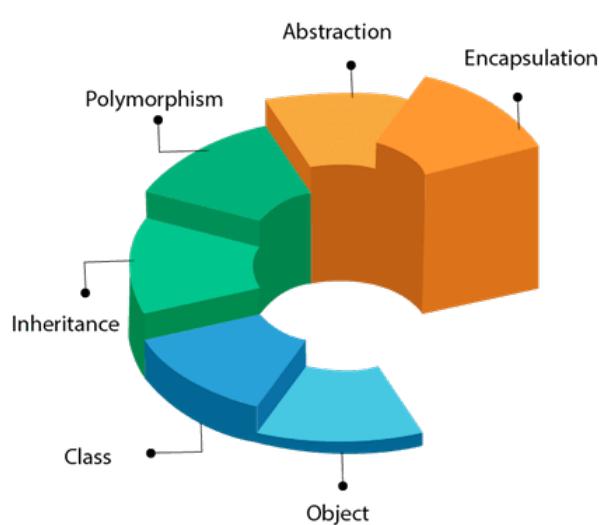
The main aim of object-oriented programming is to implement real-world entities, for example, object, classes, abstraction, inheritance, polymorphism, etc.

OOPs (Object-Oriented Programming System)

Object means a real-world entity such as a pen, chair, table, computer, watch, etc. **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

OOPs (Object-Oriented Programming System)



Object

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

Things an object knows about itself are called

- instance variables

instance
variables
(state)

Things an object can do are called

- methods

methods
(behavior)

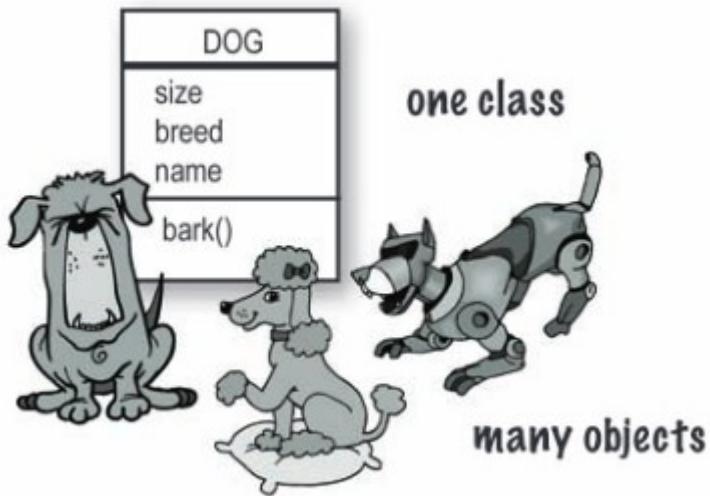
Song
title artist
setTitle() setArtist() play()

knows

does

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.



Class

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

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

Inheritance

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

Polymorphism

If *one task is performed in different ways*, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

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.



Abstraction

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.

Encapsulation

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

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



Capsule

Object Oriented Programming (OOPS)

Agenda:

1. Data Hiding
2. Abstraction
3. Encapsulation(Data Hiding+ Abstraction)
4. Tightly Encapsulated Class

1. Data Hiding:

- Our internal data should not go out directly i.e., outside person can't access our internal data directly.
- By using **private** modifier we can implement data hiding.

Example:

```
class Account
{
    private double balance;
    .....
    .....
}
```

- After providing proper username and password only, we can access our Account information.
- The main advantage of data hiding is **security**.

Note: recommended modifier for data members is **private**.

2. Abstraction:

- Hide internal implementation and just highlight the set of services, is called **abstraction**.
- By using **abstract classes** and **interfaces** we can implement abstraction.

Example:

By using ATM GUI screen bank, people are highlighting the set of services what they are offering without highlighting internal implementation.

The main advantages of Abstraction are:

1. We can achieve security as we are not highlighting our internal implementation.(i.e., outside person doesn't aware our internal implementation.)

2. Enhancement will become very easy because without effecting end-user , we can able to perform any type of changes in our internal system.
3. It provides more **flexibility** to the end user to use system very easily.
4. It improves **Maintainability** of the application.
5. It improves **modularity** of the application.
6. It improves **easyness** to use our system.

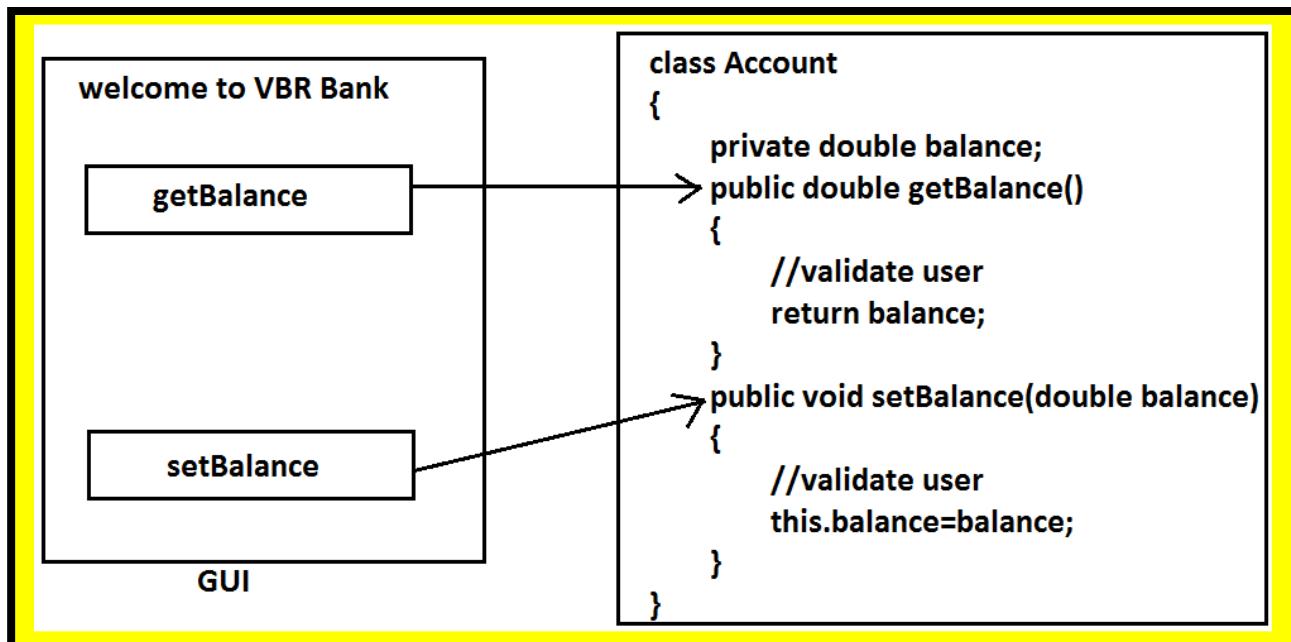
By using **interfaces (GUI screens)** we can implement abstraction.

3. Encapsulation:

- Binding of data and corresponding methods into a single unit is called Encapsulation .
- If any java class follows data hiding and abstraction such type of class is said to be encapsulated class.

Encapsulation=Datahiding+Abstraction

Example:



Every data member should be declared as private and for every member we have to maintain getter & Setter methods.

The main advantages of encapsulation are :

1. We can achieve security.
2. Enhancement will become very easy.
3. It improves maintainability and modularity of the application.
4. It provides flexibility to the user to use system very easily.

The main disadvantage of encapsulation is it increases length of the code and slows down execution.

4. Tightly encapsulated class:

A class is said to be tightly encapsulated if and only if every variable of that class declared as private whether the variable has getter and setter methods are not , and whether these methods declared as public or not, these checkings are not required to perform.

Example:

```
class Account
{
    private double balance;
    public double getBalance()
    {
        return balance;
    }
}
```

Which of the following classes are tightly encapsulated?

```
class A
{
    private int x=10; (valid)
}
class B extends A
{
    int y=20;(invalid)
}
class C extends A
{
    private int z=30; (valid)
}
```

Which of the following classes are tightly encapsulated?

```
class A
{
    int x=10;      //not
}
class B extends A
{
    private int y=20; //not
}
class C extends B
{
    private int z=30; //not
}
```

Note: if the parent class is not tightly encapsulated, then no child class is tightly encapsulated.

INHERITANCE

Agenda

1. Inheritance(IS-A relationship)
2. Types of inheritance:
 - Single inheritance
 - Multilevel inheritance
 - Hierarchical inheritance
 - Multiple inheritance
 - Hybrid inheritance
 - Cyclic inheritance

Inheritance(IS-A relationship):-

1. The process of acquiring fields (variables) and methods (behaviors) from one class to another class is called inheritance.
2. The main objective of inheritance is code extensibility, whenever we are extending class automatically the code is reused.
3. In inheritance, one class **giving** the **properties and behavior** & another class is **taking** the **properties and behavior**.
4. Inheritance is also known as **is-a** relationship. By using **extends** keyword we are achieving inheritance concept.
5. **extends** keyword used to achieve inheritance & it is providing **relationship** between two classes. when you make relationship then able to reuse the code.
6. In java, parent class is **giving** properties to child class and Child is **acquiring** properties from Parent.
7. To **reduce length of the code** and redundancy of the code sun people introduced inheritance concept.

Application code before inheritance	Application code after inheritance
<pre>class A { void m1(){} void m2(){} } class B { void m1(){} void m2(){} void m3(){} void m4(){} } class C { void m1(){} void m2(){} void m3(){} void m4(){} void m5(){} void m6(){} }</pre>	<pre>class A //parent class or super class or base { void m1(){} void m2(){} } class B extends A //child or sub or derived { void m3(){} void m4(){} } class C extends B { void m5(){} void m6(){} }</pre>

Note 1:-In java it is possible to create objects for both parent and child classes.

1. If we are creating object for parent class it is possible to call only parent specific methods.

```
A a=new A();
a.m1();
a.m2();
```

2. If we are creating object for child class it is possible to call parent specific and child specific.

```
B b=new B();
b.m1();
b.m2();
b.m3();
b.m4();
```

```
C c=new C();
c.m1();
c.m2();
c.m3();
c.m4();
c.m5();
c.m6();
```

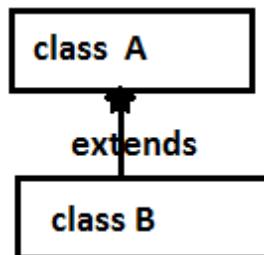
Types of inheritance :-

There are six types of inheritance.

- 1) Single inheritance
- 2) Multilevel inheritance
- 3) Hierarchical inheritance
- 4) Multiple inheritance
- 5) Hybrid inheritance
- 6) Cyclic inheritance

1) Single inheritance:-

- One class has one and only one direct super class is called single inheritance.
- In the absence of any other explicit super class, every class is implicitly a subclass of Object class.



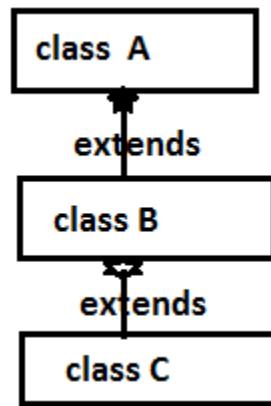
Class B extends A====>class B acquiring properties of A class.

Example:-

```
class Parent
{
    void property()
    {
        System.out.println("money");
    }
}
class Child extends Parent
{
    void m1()
    {
        System.out.println("m1 method");
    }
    public static void main(String[ ] args)
    {
        Child c = new Child();
        c.property(); //parent class method executed
        c.m1(); //child class method executed
    }
}
```

2) Multilevel inheritance:-

One Sub class is extending Parent class then that sub class will become Parent class of next extended class this flow is called multilevel inheritance.



Class B extends A ==> class B acquiring properties of A

class Class C extends B ==> class C acquiring properties of B class

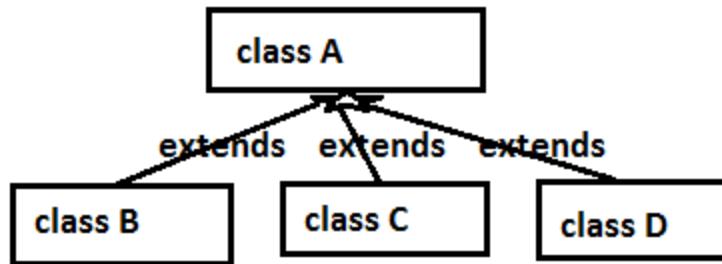
[indirectly class C using properties of A & B classes]

Example:-

```
class A
{
    void m1()
    {
        System.out.println("m1 method");
    }
}
class B extends A
{
    void m2()
    {
        System.out.println("m2 method");
    }
}
class C extends B
{
    void m3()
    {
        System.out.println("m3 method");
    }
}
public static void main(String[] args)
{
    A a = new A();          a.m1();
    B b = new B();          b.m1(); b.m2();
    C c = new C();          c.m1(); c.m2(); c.m3();
}
```

3) Hierarchical inheritance :-

More than one sub class is extending single Parent is called hierarchical inheritance.



Class B extends A ==> class B acquiring properties of A class

Class C extends A ==> class C acquiring properties of A class

Class D extends A ==> class D acquiring properties of A class

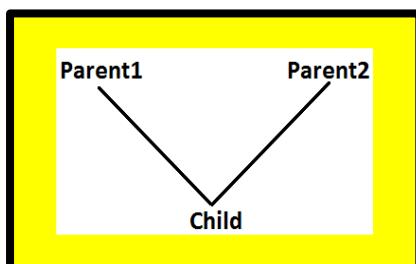
Example:-

```
class A
{
    void m1()
    {
        System.out.println("A class");
    }
}
class B extends A
{
    void m2()
    {
        System.out.println("B class");
    }
}
class C extends A
{
    void m2()
    {
        System.out.println("C class");
    }
}
class Test
{
    public static void main(String[] args)
    {
        B b= new B();
        b.m1();
        b.m2();
        C c = new C();
        c.m1();
        c.m2();
    }
}
```

4) Multiple inheritance:-

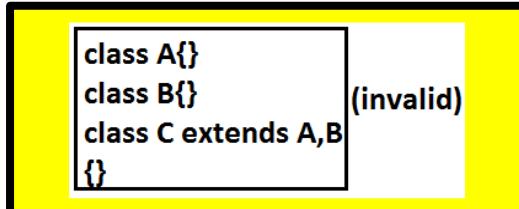
Having more than one Parent class at the same level is called multiple inheritance.

Example:



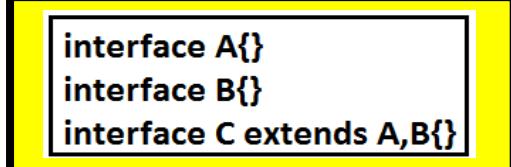
Any class can extends only one class at a time and can't extends more than one class Simultaneously.
Hence java won't provide support for multiple inheritance.

Example:



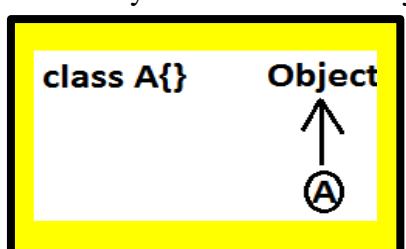
But an interface can extends any no. of interfaces at a time. Hence java provides support for multiple inheritance through interfaces.

Example:



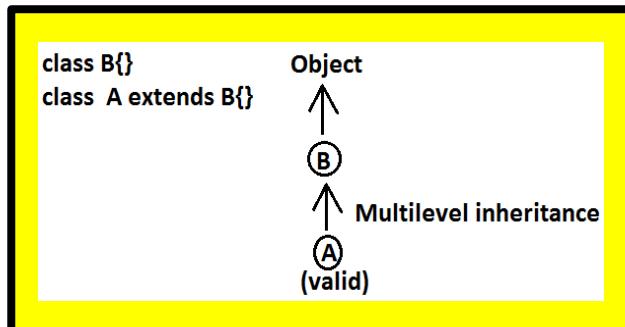
If our class doesn't extends any other class then only our class is the direct child class of object.

Example:

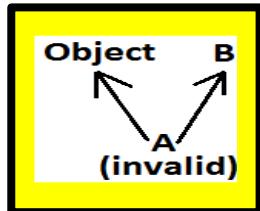


If our class extends any other class then our class is not direct child class of object, It is indirect child class of object , which forms multilevel inheritance.

Example 1:



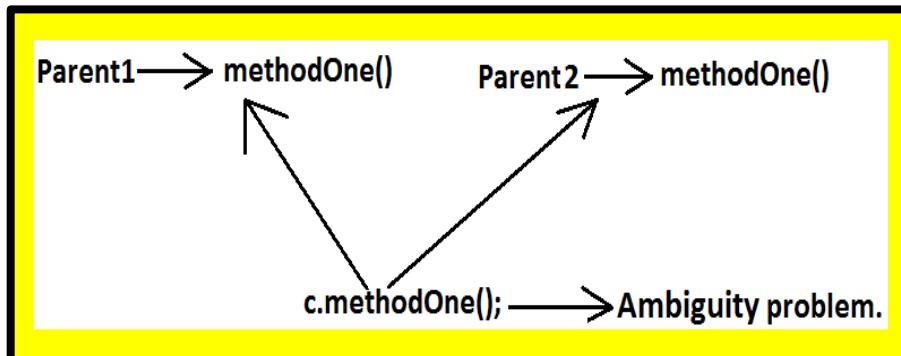
Example 2:



Why java won't provide support for multiple inheritance?

There may be a chance of raising ambiguity problems.

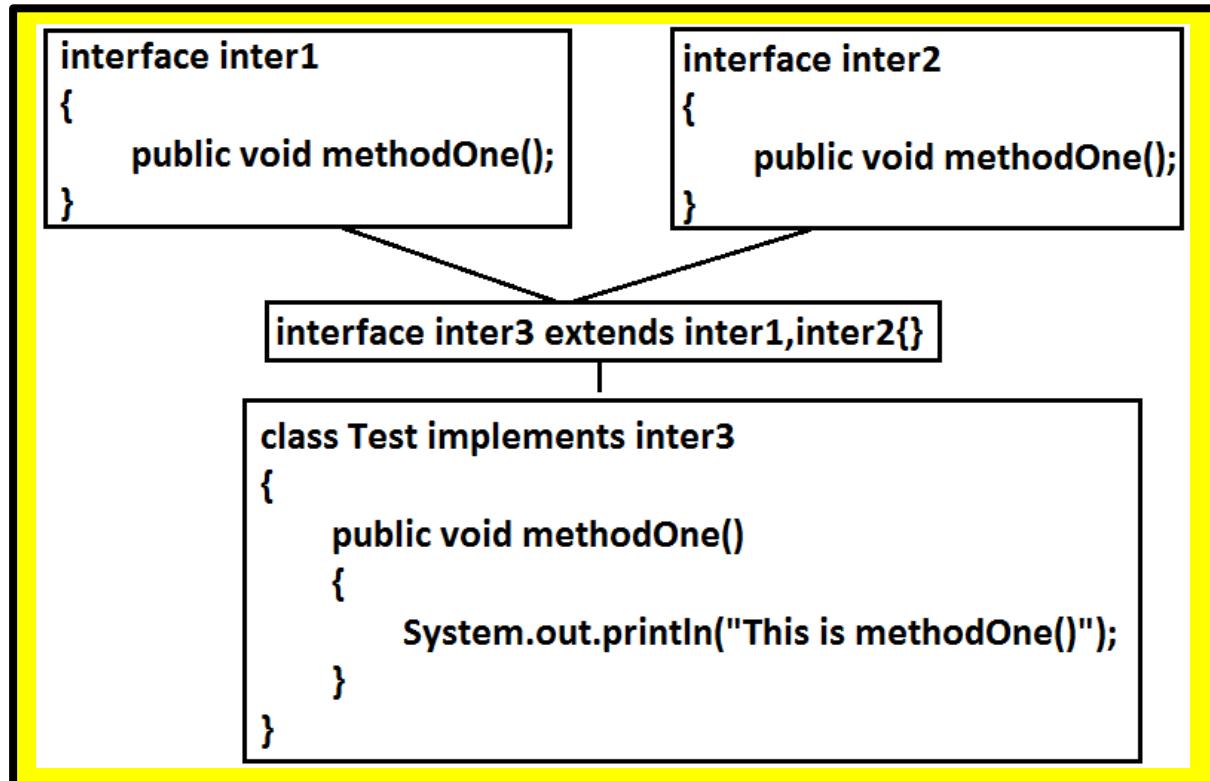
Example:



Why ambiguity problem won't be there in interfaces?

Interfaces having dummy declarations and they won't have implementations hence no ambiguity problem.

Example:

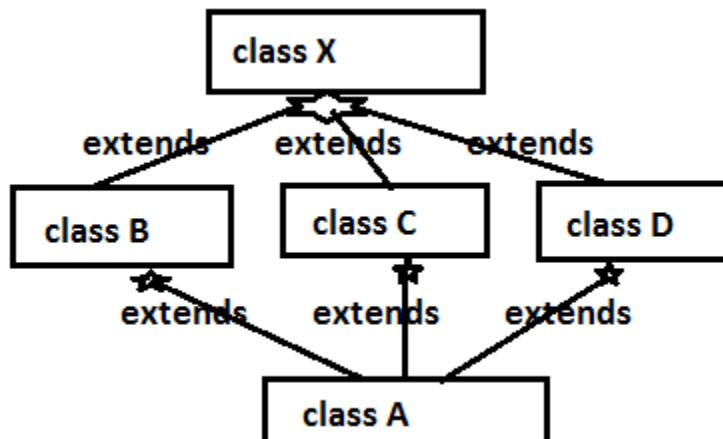


Example:-

```
class A
{
    void money()
    {
        System.out.println("A class money");
    }
}
class B
{
    void money()
    {
        System.out.println("B class money");
    }
}
class C extends A,B
{
    public static void main(String[ ] args)
    {
        C c = new C();
        c.money(); //which method executed A--->money() or B--->money
    }
}
```

5) Hybrid inheritance:-

- Hybrid is combination of hierarchical & multiple inheritance .
- Java is not supporting hybrid inheritance because multiple inheritance(not supported by java) is included in hybrid inheritance.

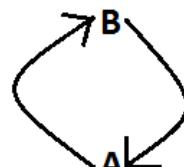


6) Cyclic inheritance:

Cyclic inheritance is not allowed in java.

Example 1:

```
class A extends B{} } (invalid)
class B extends A{} } C.E:cyclic inheritance involving A
```



Example 2:

class A extends A{}  cyclic inheritance involving A

Preventing inheritance:-

- You can prevent sub class creation by using final keyword in the parent class declaration.

final class Parent //for this class child class creation not possible because it is final.

```
{  
}  
class Child extends Parent  
{  
}
```

compilation error:- cannot inherit from final Parent

Note:-

- 1) Except for the Object class , a class has one direct super class.
- 2) A class inherit fields and methods from all its super classes whether directly or indirectly.
- 3) An abstract class can only be sub classed but cannot be instantiated.
- 4) In parent and child, it is recommended to create object of Child class.

Object Oriented Programming (OOPS)

Agenda:

- 1. HAS-A Relationship**
 - Composition
 - Aggregation
- 2. Method Signature**

HAS-A relationship:

1. HAS-A relationship is also known as **composition** (or) **aggregation**.
2. There is no specific keyword to implement HAS-A relationship but mostly we can use **new** operator.
3. The main advantage of HAS-A relationship is **reusability**.

Example:

```
class Engine
{
    //engine specific functionality
}
class Car
{
    Engine e=new Engine();
    //....;
    //....;
    //....;
}
```

- class Car HAS-A engine reference.
- The main dis-advantage of HAS-A relationship **increases dependency** between the components and creates maintains problems.

Composition vs Aggregation:

Composition:

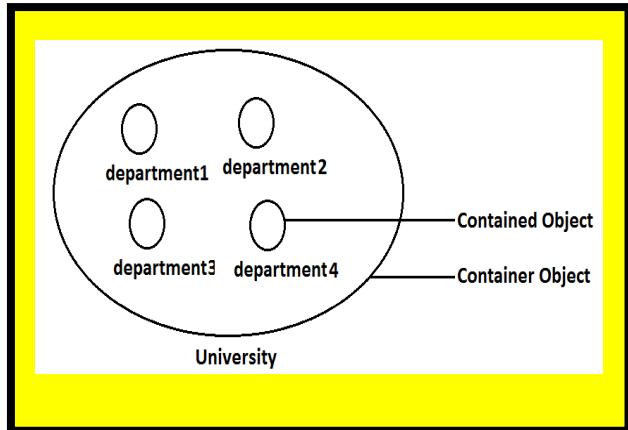
Without existing container object, if there **is no chance** of existing contained objects, then the relationship between container object and contained object is called composition, which is a **strong association**.

Example:

University consists of several departments. Whenever university object destroys, automatically all the department objects will be destroyed i.e., without **existing university object**, there is no chance

of existing dependent object. Hence these are strongly associated and this relationship is called composition.

Example:



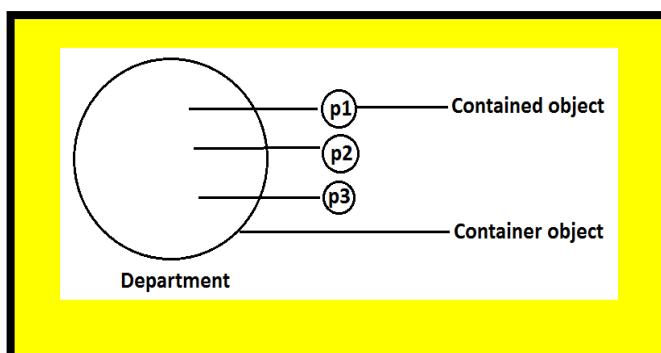
Aggregation :

Without existing container object, if there is a chance of existing contained objects, such type of relationship is called aggregation. In aggregation objects have weak association.

Example:

Within a department, there may be a chance of several professors will work whenever we are closing department still there may be a chance of existing professor object without existing department object the relationship between department and professor is called aggregation where the objects having weak association.

Example:



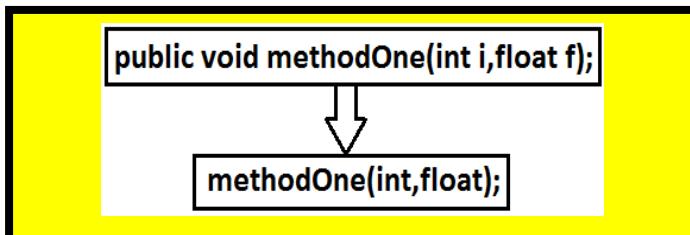
Note :

- In composition, container contained objects are strongly associated, and but container object holds contained objects directly.
- But in Aggregation, container and contained objects are weakly associated and container object just now holds the reference of contained objects.

Method signature:

In java, method signature consists of **name of the method** followed by **argument types**.

Example:



- In java return type is not part of the method signature.
- Compiler will use method signature while resolving method calls.

```
class Test
{
    public void m1(double d) { }
    public void m2(int i) { }
    public static void main(String args[])
    {
        Test t=new Test();
        t.m1(10.5);
        t.m2(10);
        t.m3(10.5); //CE
    }
}
```

```
CE : cannot find symbol
symbol : method m3(double)
location : class Test
```

Within the same class, we can't take 2 methods with the same signature, otherwise we will get compile time error.

Example:

```
public void methodOne() { }
public int methodOne()
{
    return 10;
}
```

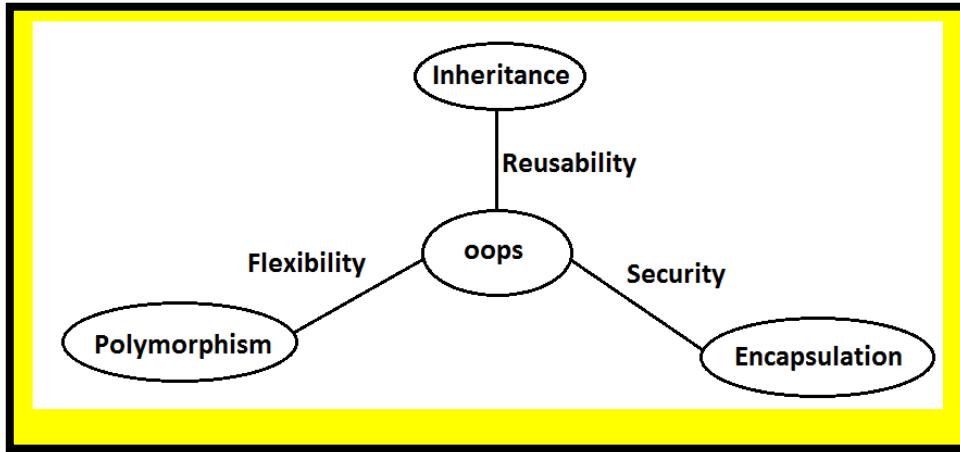
Output:
Compile time error
methodOne() is already defined in Test

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:

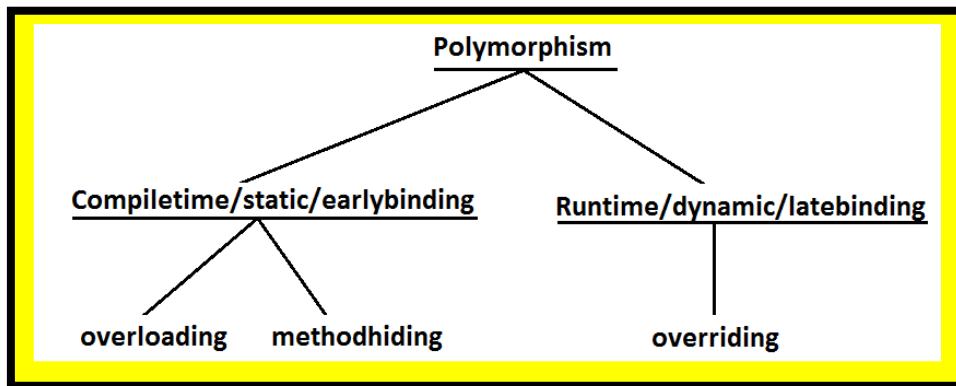
1. `abs(int)`
2. `abs(long)`
3. `abs(float)`

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:



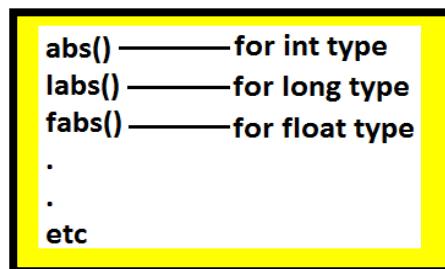
There are two types of polymorphism in java

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:-

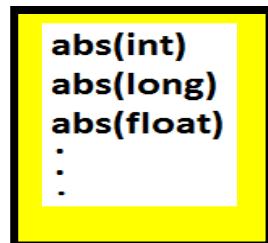
- 1) Two methods are said to be **overload**, if and only if both having the **same name** but **different argument types**.
- 2) In '**C**' language, we **can't take** 2 methods with the same name and different types. If there is a change in argument type, compulsory we should go for new method name.

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**.

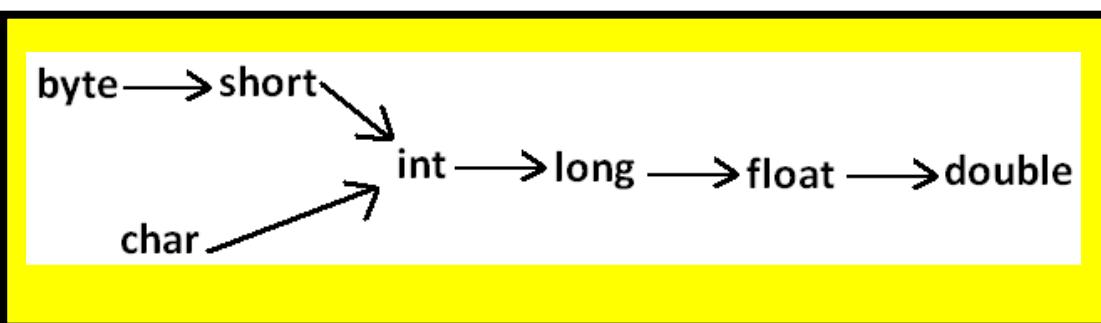
```
class Test
{
    public void methodOne()
    {
        System.out.println("no-arg method");
    }
    public void methodOne(int i)
    {
        System.out.println("int-arg method");
        //overloaded methods
    }
    public void methodOne(double d)
    {
        System.out.println("double-arg 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**.

Case 1: Automatic promotion in overloading.

- In overloading if compiler is unable to find the method with exact match we won't get any compile time error immediately.
- First compiler promotes the argument to the next level and checks whether the matched method is available or not. If it is available, then that method will be considered. If it is not available, then compiler promotes the argument once again to the next level. This process will be continued until all possible promotions still if the matched method is not available then we will get compile time error. This process is called automatic promotion in overloading.

The following are various possible automatic promotions in overloading.



Example:

```
class Test
{
    public void methodOne(int i)
    {
        System.out.println("int-arg method");
    }
    public void methodOne(float f) //overloaded methods
    {
        System.out.println("float-arg method");
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        //t.methodOne('a');//int-arg method
        //t.methodOne(10.1);//float-arg method
        t.methodOne(10.5); //C.E:cannot find symbol
    }
}
```

Overriding

Overriding

- **Rules for overriding**
- **Overriding with Checked Exceptions**
- **Overriding with respect to static methods.**

Method hiding

- **Differences between method hiding and overriding**

Overriding :

1. Whatever the Parent has by default available to the Child through inheritance, if the Child is not satisfied with Parent class method implementation then Child is allow to redefine that Parent class method in Child class in its own way this process is called overriding.
2. The Parent class method which is overridden is called **overridden method**.
3. The Child class method which is overriding is called **overriding method**.

Example 1:

```
class Parent
{
    public void property()
    {
        System.out.println("cash+land+gold");
    }
    public void marry()
    {
        System.out.println("subbalakshmi");           //overridden method
    }
}
class Child extends Parent
{
    public void marry()
    {
        System.out.println("3sha/4me/9tara/anushka"); //overriding method
    }
}
class Test
{
    public static void main(String[] args)
    {
        Parent p=new Parent();
        p.marry();                      //subbalakshmi(parent method)
        Child c=new Child();
        c.marry();                      //3sha/4me/9tara(child method)
        Parent p1=new Child();
    }
}
```

```

        p1.marry();           //3sha/4me/9tara(child method)
    }
}

```

4. In overriding method resolution is always takes care by JVM based on runtime object hence overriding is also considered as **runtime polymorphism** or **dynamic polymorphism** or **late binding**.
5. The process of overriding method resolution is also known as **dynamic method dispatch**.

Note: In overriding runtime object will play the role and reference type is dummy.

Rules for overriding :

1. In overriding **method names and arguments must be same**. That is method signature must be same.
2. **Private** methods are not visible in the Child classes, hence **overriding concept is not applicable for private methods**. Based on own requirement we can declare the same Parent class private method in child class also. It is valid but not overriding.

Example:

```

class Parent
{
    private void methodOne()
    {}

    class Child extends Parent
    {
        private void methodOne()
        {}
    }
}

```

it is valid but not overriding.

3. Parent class final methods we can't override in the Child class.

Example:

```

class Parent
{
    public final void methodOne()      {}
}

class Child extends Parent{
    public void methodOne(){}
}

```

Output:

Compile time error:

```

methodOne() in Child cannot override methodOne()
in Parent; overridden method is final

```

Parent class non final methods we can override as final in child class. We can override native methods in the child classes.

4. We should override Parent class abstract methods in Child classes to provide implementation.

Example:

```
abstract class Parent
{
    public abstract void methodOne();
}
class Child extends Parent
{
    public void methodOne() { }
```

Diagram:



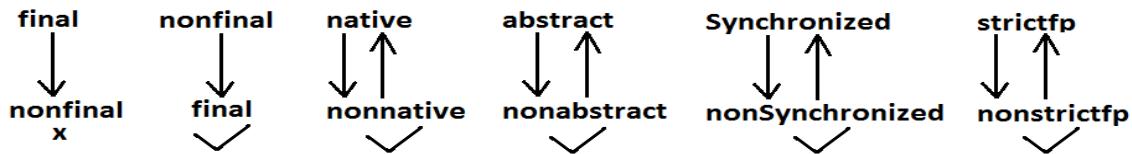
5. We can override a non-abstract method as abstract
this approach is helpful to stop availability of Parent method implementation to the next level child classes.

Example:

```
class Parent
{
    public void methodOne() { }
}
abstract class Child extends Parent
{
    public abstract void methodOne();
```

Synchronized, strictfp, modifiers won't keep any restrictions on overriding.

Diagram:



6. While overriding we can't reduce the scope of access modifier.

Example:

```

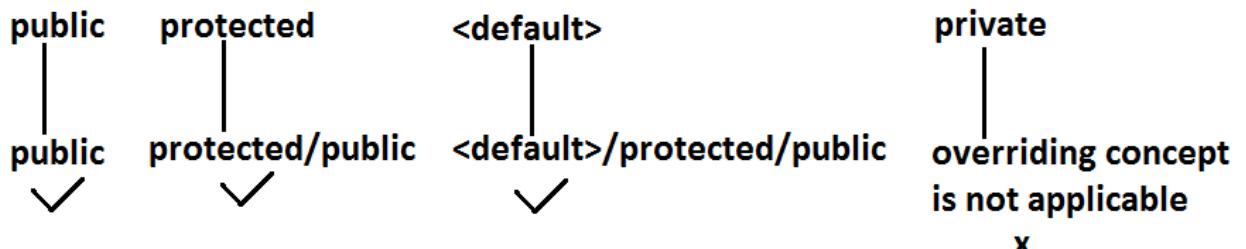
class Parent
{
    public void methodOne() { }
}
class Child extends Parent
{
    protected void methodOne( ) { }
}
  
```

Output:

Compile time error :

methodOne() in Child cannot override methodOne() in Parent;
attempting to assign weaker access privileges; was public

Diagram:

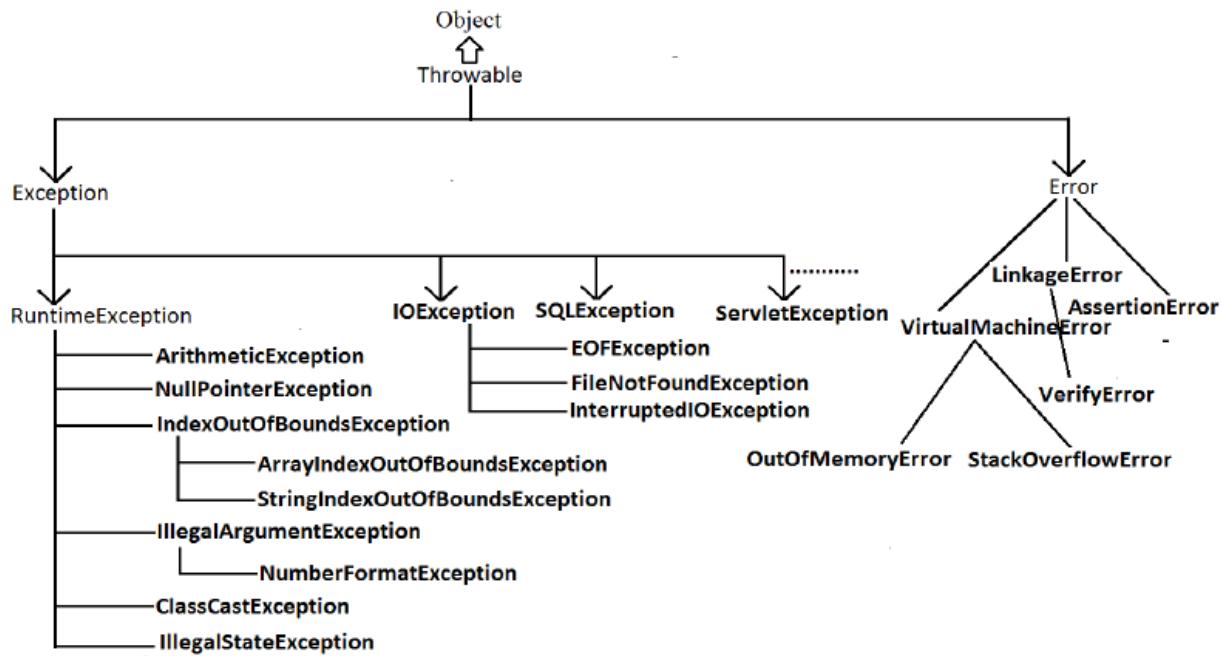


private < default < protected < public

Overriding with Checked Exceptions :

- The exceptions which are checked by the compiler for smooth execution of the program at runtime are called **checked exceptions**.
- The exceptions which are not checked by the compiler are called **un-checked exceptions**.
- RuntimeException and its child classes, Error and its child classes are **unchecked** except these the remaining are checked exceptions.

Diagram:



Rule: While overriding if the child class method throws any **checked exception**, compulsory the parent class method should throw the **same checked exception** or its parent, otherwise we will get compile time error.

But there are no restrictions for un-checked exceptions.

Example:

```
class Parent
{
    public void methodOne() {}
}
class Child extends Parent
{
    public void methodOne() throws Exception {}
}
```

Output:

Compile time error :

```
methodOne() in Child cannot override methodOne() in Parent;
overridden method does not throw java.lang.Exception
```

Examples :

- ① Parent: public void methodOne()throws Exception
Child: public void methodOne()
} valid
- ② Parent: public void methodOne()
Child : public void methodOne()throws Exception
} invalid
- ③ Parent: public void methodOne()throws Exception
Child: public void methodOne()throws Exception
} valid
- ④ Parent: public void methodOne()throws IOException
Child: public void methodOne()throws Exception
} invalid
- ⑤ Parent: public void methodOne()throws IOException
Child: public void methodOne()throws EOFException,FileNotFoundException
} valid
- ⑥ Parent: public void methodOne()throws IOException
Child : public void methodOne()throws EOFException,InterruptedException
} invalid
- ⑦ Parent: public void methodOne()throws IOException
Child: public void methodOne()throws EOFException,ArithmaticException
} valid
- ⑧ Parent: public void methodOne()
Child: public void methodOne()throws
ArithmaticException,NullPointerException,ClassCastException,RuntimeException
} valid

Overriding with respect to static methods:

Case 1:

We can't override a static method as non static.

Example:

```
class Parent
{
    public static void methodOne(){}
    //here static methodOne() method is a class level
}
class Child extends Parent
{
    public void methodOne(){}
    //here methodOne() method is a object level hence
    // we can't override methodOne() method
}
output :
```

CE: methodOne in Child can't override methodOne() in Parent ;
overriden method is static

Case 2:

Similarly we can't override a non static method as static.

Case 3:

```
class Parent
{
    public static void methodOne() {}
}
class Child extends Parent {
    public static void methodOne() {}
}
```

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

METHOD HIDING:

All rules of method hiding are exactly same as overriding except the following differences.

Overriding	Method hiding
1. Both Parent and Child class methods should be non static .	1. Both Parent and Child class methods should be static .
2. Method resolution is always takes care by JVM based on runtime object.	2. Method resolution is always takes care by compiler based on reference type.
3. Overriding is also considered as runtime polymorphism (or) dynamic polymorphism (or) late binding .	3. Method hiding is also considered as compile time polymorphism (or) static polymorphism (or) early binding .

Example:

```
class Parent
{
    public static void methodOne()
    {
        System.out.println("parent class");
    }
}
class Child extends Parent
{
    public static void methodOne()
    {
        System.out.println("child class");
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        Parent p=new Parent();
        p.methodOne();          //parent class
        Child c=new Child();
        c.methodOne();          //child class
        Parent p1=new Child();
        p1.methodOne();         //parent class
    }
}
```

Note: If both Parent and Child class methods are **non static** then it will become overriding and method resolution is based on runtime object. In this case the output is

```
Parent class
Child class
Child class
```

Dynamic method dispatch in java

Dynamic method dispatch is a mechanism to resolve overridden method call at run time instead of compile time. It is based on the concept of up-casting (A super class reference variable can refer subclass object.).

Example 1:

```
/**  
 * This program is used for simple method overriding example.  
 * with dynamic method dispatch.  
 */  
class Student {  
    /**  
     * This method is used to show details of a student.  
     */  
    public void show(){  
        System.out.println("Student details.");  
    }  
}  
  
public class CollegeStudent extends Student {  
    /**  
     * This method is used to show details of a college student.  
     */  
    public void show(){  
        System.out.println("College Student details.");  
    }  
  
    //main method  
    public static void main(String args[]){  
        //Super class can contain subclass object.  
        Student obj = new CollegeStudent();  
  
        //method call resolved at runtime  
        obj.show();  
    }  
}
```

Output:

College Student details.
School Student details.

Note: Only super class methods can be overridden in subclass, data members of super class cannot be overridden.

```
/**  
 * This program is used for simple method overriding example.  
 * with dynamic method dispatch.  
 */  
class Student {  
    int maxRollNo = 200;  
}  
  
class SchoolStudent extends Student{  
    int maxRollNo = 120;  
}  
  
class CollegeStudent extends SchoolStudent{  
    int maxRollNo = 100;  
}  
  
public class StudentTest {  
    public static void main(String args[]){  
        //Super class can contain subclass object.  
        Student obj1 = new CollegeStudent();  
        Student obj2 = new SchoolStudent();  
  
        //In both calls maxRollNo of super class will be printed.  
        System.out.println(obj1.maxRollNo);  
        System.out.println(obj2.maxRollNo);  
    }  
}
```

Output:

200
200

Abstract Modifier

Agenda:

Abstract Modifier

- Abstract Methods
- Abstract class

Abstract modifier:

Abstract is the modifier applicable only for methods and classes but not for variables.

a) Abstract methods:

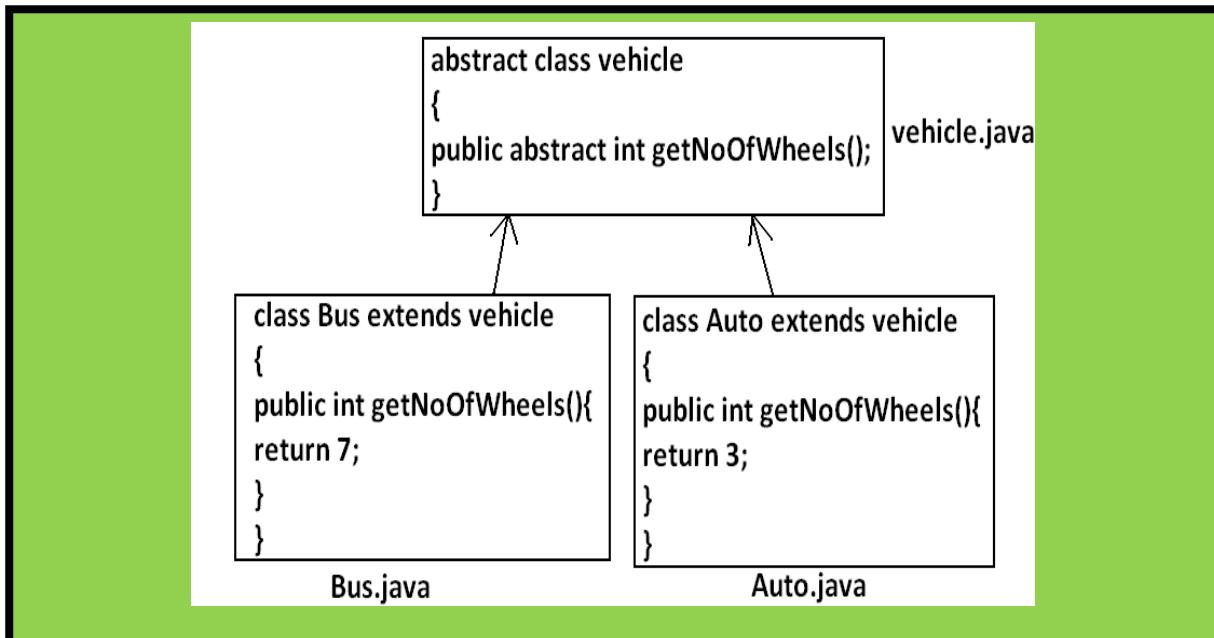
- Even though we don't have implementation still we can declare a method with abstract modifier. i.e., abstract methods have **only declaration** but **not implementation**.
- Hence abstract method declaration should compulsory ends with **semicolon**.

Example:

```
public abstract void methodOne(); → valid  
public abstract void methodOne(){} → invalid
```

Child classes are responsible to provide implementation for parent class abstract methods.

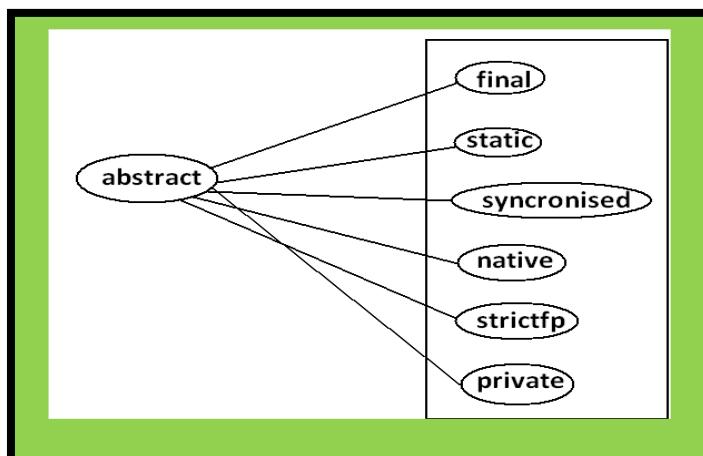
Program:



- The main advantage of **abstract methods** is , by declaring abstract method in parent class we can provide **guide lines** to the **child class** such that which methods they should **compulsory implement**.
- Abstract method **never talks about implementation**.
- **If** any modifier talks about implementation **then** the modifier will be **enemy** to abstract and that is always illegal combination for methods.

The following are the various illegal combinations for methods.

Diagram:



All the 6 combinations are illegal.

b) Abstract class:

For any java class, **if** we are **not allowed to create an object**, **such type of class** we have to declare with abstract modifier i.e., for abstract class, **instantiation is not possible**.

Example:

```

abstract class Test
{
    public static void main(String args[])
    {
        Test t=new Test();
    }
}
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:4: Test is abstract; cannot be instantiated
Test t=new Test();
```

What is the difference between abstract class and abstract method ?

- If a class contain at least one abstract method, then compulsory the corresponding class should be declared with abstract modifier. Because implementation is not complete and hence we can't create object of that class.
- Even though class doesn't contain any abstract methods, still we can declare the class as abstract i.e., an abstract class can contain zero no of abstract methods also.

Example1: HttpServlet class is abstract but it doesn't contain any abstract method.

Example2: Every adapter class is abstract but it doesn't contain any abstract method.

Example1:

```
class Parent
{
    public void methodOne();
}
Output:
Compile time error.
D:\Java>javac Parent.java
Parent.java:3: missing method body, or declare abstract
public void methodOne();
```

Example 2:

```
class Parent
{
    public abstract void methodOne() {}
}
Output:
Compile time error.
Parent.java:3: abstract methods cannot have a body
public abstract void methodOne() {}
```

Example3:

```
class Parent
{
    public abstract void methodOne();
}
Output:
Compile time error.
D:\Java>javac Parent.java
Parent.java:1: Parent is not abstract and does not override abstract
method methodOne() in Parent
class Parent
```

If a class extends any abstract class, then compulsory we should provide implementation for every abstract method of the parent class otherwise we have to declare child class as abstract.

Example:

```
abstract class Parent
{
    public abstract void methodOne();
```

```

        public abstract void methodTwo();
}
class child extends Parent
{
    public void methodOne() {}
}
Output:
Compile time error.
D:\Java>javac Parent.java
Parent.java:6: child is not abstract and does not override abstract
method methodTwo() in Parent
class child extends Parent

```

If we declare class child as abstract **then** the code compiles fine **but** child of child is responsible to provide implementation for methodTwo().

What is the difference between final and abstract ?

- For abstract methods, compulsory we should override in the child class to provide implementation. Whereas for final methods, we can't override. Hence abstract final combination is illegal for methods.
- For abstract classes, we should compulsory create child class to provide implementation whereas for final class we can't create child class. Hence final abstract combination is illegal for classes.
- Final class cannot contain abstract methods whereas abstract class can contain final method.

Example:

final class A { public abstract void methodOne(); }	abstract class A { public final void methodOne(); }
invalid	valid

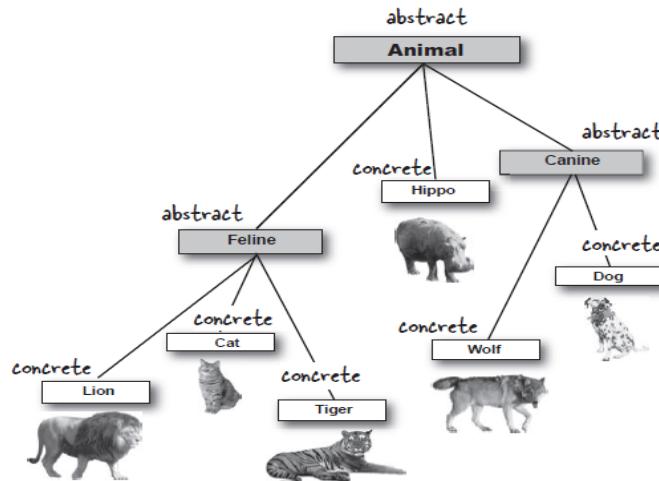
Note:

Usage of abstract methods, abstract classes and interfaces is always good Programming practice.

Abstract class vs Concrete class:

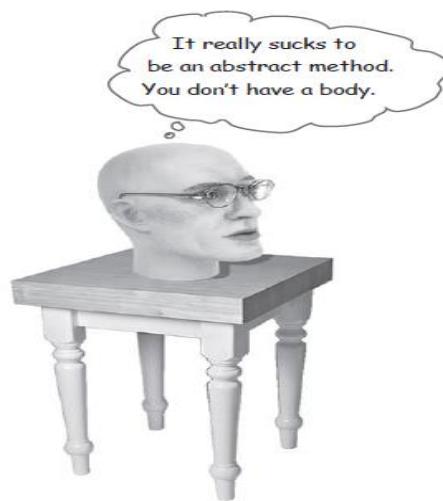
The only real difference is that a concrete class can be instantiated because it provides (or inherits) the implementation for all of its methods. An abstract class cannot be instantiated because at least one method has not been implemented.

A class that's not abstract is called a **concrete** class. In the Animal inheritance tree, if we make Animal, Canine, and Feline abstract, that leaves Hippo, Wolf, Dog, Tiger, Lion, and Cat as the concrete subclasses.



Important points:

1. An abstract method has no body! **public abstract void eat();**
2. If you declare an abstract method, you MUST mark the class abstract as well. You cannot have an abstract method in a non-abstract class.
3. You MUST implement all abstract methods.
4. Implementing an abstract method is just like overriding a method.



Final modifier

Agenda:

1. Final Modifier
 - a. Final Methods
 - b. Final Class
 - c. Final Variables
 - i. Final instance variables
 - At the time of declaration
 - Inside instance block
 - Inside constructor
 - ii. Final static variables
 - At the time of declaration
 - Inside static block
 - iii. Final local variables

Final modifier:

Final is the modifier applicable for **classes**, **methods** and **variables**.

a) Final methods:

- Whatever methods parent has, by default available to the child.
- If the **child** is not allowed to override any method, that method we have to declare with **final** in parent class i.e., final methods **cannot be overridden**.

Example:

Program 1:

```
class Parent
{
    public void property()
    {
        System.out.println("cash+gold+land");
    }
    public final void marriage()
    {
        System.out.println("Venkata laxmi");
    }
}
class child extends Parent
{
    public void marriage()
    {
        System.out.println("Samantha");
    }
}
```

OUTPUT:
Compile time error.

```
D:\Java>javac child.java  
child.java:3: marriage() in child cannot override marriage() in Parent;  
overridden method is final
```

b) Final class:

If a class declared as final, then we can't create the child class i.e., inheritance concept is not applicable for final classes.

Example:

Program 1:

```
final class Parent  
{  
}  
class child extends Parent  
{  
}
```

OUTPUT:

Compile time error.

```
D:\Java>javac child.java  
child.java:1: cannot inherit from final Parent  
class child extends Parent
```

Note: Every method present inside a final class is **always final** by default, whether we are declaring or not. But every **variable** present inside a final class need not be final.

Example:

```
final class parent  
{  
    static int x=10;  
    static  
    {  
        x=999;  
    }  
}
```

Advantages and disadvantages of final keyword:

Advantage:

we can achieve security.

Disadvantage:

we are missing the key benefits of OOPS:

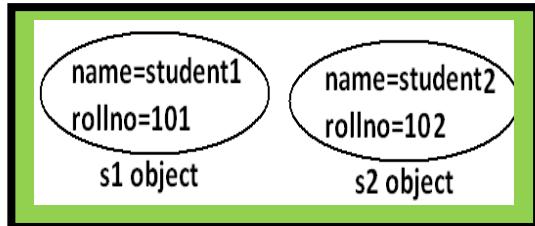
- i. polymorphism (because of final methods),
- ii. inheritance (because of final classes) hence if there is no specific requirement never recommended to use final keyword.

c) Final variables:

Final 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.

DIAGRAM:



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

Example:

```
class Test
{
    int i;
    public static void main(String args[])
    {
        Test t=new Test();
        System.out.println(t.i);
    }
}
Output:
D:\Java>javac Test.java
D:\Java>java Test
0
```

If the instance variable declared as `final`, compulsory we should perform initialization explicitly and JVM won't provide any default values. whether we are using or not otherwise we will get compile time error.

Example:

Program 1:

```
class Test
{
    int i;
}
Output:
D:\Java>javac Test.java
D:\Java>
```

Program 2:

```
class Test
{
    final int i;
```

```
Output:  
Compile time error.  
D:\Java>javac Test.java  
Test.java:1: variable i might not have been initialized  
class Test
```

Rule:

For the final instance variables we should perform **initialization** before constructor completion. That is the following are various possible places for this.

1) At the time of declaration:

Example:

```
class Test  
{  
    final int i=10;  
}  
Output:  
D:\Java>javac Test.java  
D:\Java>
```

2) Inside instance block:

Example:

```
class Test  
{  
    final int i;  
    {  
        i=10;  
    }  
}  
Output:  
D:\Java>javac Test.java  
D:\Java>
```

3) Inside constructor:

Example:

```
class Test  
{  
    final int i;  
    Test()  
    {  
        i=10;  
    }  
}  
Output:  
D:\Java>javac Test.java  
D:\Java>
```

If we are performing initialization anywhere else we will get compile time error.

Example:

```
class Test
{
    final int i;
    public void methodOne()
    {
        i=10;
    }
}
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:5: cannot assign a value to final variable i
i=10;
```

Final 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 the instance variables**. We have to declare those variables at class level by using **static** modifier.
- In the case of **instance variables**, for every object a **seperate copy** will be created. But in the case of **static variables**, a **single copy** will be created at class level and **shared by every object of that class**.
- For the static variables it is **not required to perform initialization**, explicitly jvm will always provide default values.

Example:

```
class Test
{
    static int i;
    public static void main(String args[])
    {
        System.out.println("value of i is :" + i);
    }
}
Output:
D:\Java>javac Test.java
D:\Java>java Test
Value of i is: 0
```

If the static variable declare as **final**, then compulsory we **should perform initialization** explicitly whether we are using or not otherwise we will get compile time error.(The JVM won't provide any default values)

Example:

```
class Test  
{  
    static int i;  
}
```

valid

```
class Test  
{  
    final static int i;  
}
```

invalid

Output:

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

```
Test.java:1: variable i might not have been initialized  
class Test
```

Rule:

For the final static variables we should perform initialization before class loading completion otherwise we will get compile time error. That is the following are possible places.

1) At the time of declaration:

Example:

```
class Test  
{  
    final static int i=10;  
}  
Output:  
D:\Java>javac Test.java  
D:\Java>
```

2) Inside static block:

Example:

```
class Test  
{  
    final static int i;  
    static  
    {  
        i=10;  
    }  
}  
Output:  
Compile successfully.
```

If we are performing initialization anywhere else we will get compile time error.

Example:

```
class Test
{
    final static int i;
    public static void main(String args[])
    {
        i=10;
    }
}
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:5: cannot assign a value to final variable i
i=10;
```

Final local variables:

- To meet temporary requirement of the Programmer, sometimes we can declare the variable inside a method or block or constructor such type of variables are called **local variables**.
- For the local variables, jvm won't provide any default value. Compulsory, we should perform initialization explicitly before using that variable.

Example:

```
class Test
{
    public static void main(String args[])
    {
        int i;
        System.out.println("hello");
    }
}
Output:
D:\Java>javac Test.java
D:\Java>java Test
Hello
```

Example:

```
class Test
{
    public static void main(String args[])
    {
        int i;
        System.out.println(i);
    }
}
Output:
Compile time error.
```

```
D:\Java>javac Test.java
Test.java:5: variable i might not have been initialized
System.out.println(i);
```

Even though local variable declared as the final, before using only, we should perform initialization.

Example:

```
class Test
{
    public static void main(String args[])
    {
        final int i;
        System.out.println("hello");
    }
}
Output:
D:\Java>javac Test.java
D:\Java>java Test
hello
```

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[])
    {
        private int x=10;----- (invalid)
        public int x=10; ----- (invalid)
        volatile int x=10; ----- (invalid)
        transient int x=10; ----- (invalid)
        final int x=10; ----- (valid)
    }
}
```

```
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:5: illegal start of expression
private int x=10;
```

Important points:

1. A final `variable` means you can't `change` its value.
2. A final `method` means you can't `override` the method.
3. A final `class` means you can't `extend` the class (i.e. you can't make a subclass).

Interfaces

Agenda

Interfaces

- Interface declarations and implementations
- Extends vs implements
- Interface methods
- Interface variables
- Interface vs abstract class vs concrete class
- Difference between interface and abstract class?

Interfaces:

- Interface is also one of the **type of class** it contains **only abstract methods** and Interfaces are not alternative for abstract class, it is **extension for abstract classes**.
- For the interfaces, the compiler will generates **.class** files.
- Interfaces giving the **information about the functionalities** and these are **not giving the information about internal implementation**.
- Inside the source file, it is possible to declare any number of interfaces. And we are declaring the interfaces by using **interface** keyword.

Syntax:-**interface interface-name { }**

Eg: **interface it1 { }**

BOTH EXAMPLES ARE SAME

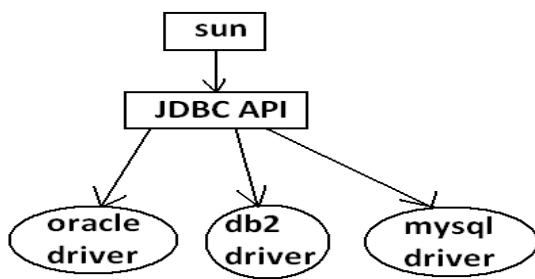
interface it1 { void m1(); void m2(); void m3(); }	abstract interface it1 { public abstract void m1(); public abstract void m2(); public abstract void m3(); }
--	---

Note: - If we are declaring or not each and every interface method by default **public abstract**. And the interfaces are by default **abstract** hence for the interfaces object creation is not possible.

Def 1 : Any Service Requirement Specification (SRS) is called an interface.

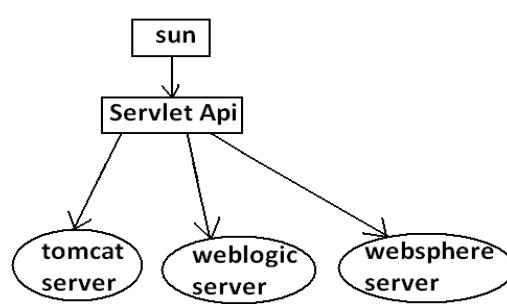
Example1: Sun people responsible to define JDBC API and database vendor will provide implementation for that.

Diagram:



Example2: Sun people define Servlet API to develop web applications web server vendor is responsible to provide implementation.

Diagram:



Def 2: From the client point of view an interface define the set of services what is expecting. From the service provider point of view an interface defines the set of services what is offering. Hence an interface is considered as a contract between client and service provider.

Example: ATM GUI screen describes the set of services what bank people offering, at the same time the same GUI screen the set of services what customer is expecting hence this GUI screen acts as a contract between bank and customer.

Def 3: Inside interface every method is always abstract whether we are declaring or not hence interface is considered as 100% pure abstract class.

Summary def: Any service requirement specification (SRS) or any contract between client and service provider or 100% pure abstract classes is considered as an interface.

Declaration and implementation of an interface:

Note1:

Whenever we are implementing an interface, compulsory for every method of that interface we should provide implementation otherwise we have to declare class as abstract in that case child class is responsible to provide implementation for remaining methods.

Note2:

Whenever we are implementing an interface method, compulsory it should be declared as public otherwise we will get compile time error.

Extends vs implements:

A class can extends **only one class** at a time.

Example:

```
class One
{
    public void methodOne()
    {
    }
}
class Two extends One
{
}
```

A class can implements **any no. of interfaces** at a time.

Example:

```
interface One
{
    public void methodOne();
}
interface Two
{
    public void methodTwo();
}
class Three implements One,Two
{
    public void methodOne()
    {
    }
    public void methodTwo()
    {
    }
}
```

A class can extend a class and can implement any no. of interfaces simultaneously.

```
interface One
{
    void methodOne();
}
class Two
{
    public void methodTwo()
    {
    }
}
class Three extends Two implements One
```

```
{  
    public void methodOne()  
    {  
    }  
}
```

An interface can extend any no. of interfaces at a time.

Example:

```
interface One  
{  
    void methodOne();  
}  
interface Two  
{  
    void methodTwo();  
}  
interface Three extends One,Two  
{  
}
```

Which of the following is true?

1. A class can extend any no.of classes at a time.(FALSE)
2. An interface can extend only one interface at a time.(FALSE)
3. A class can implement only one interface at a time.(FALSE)
4. A class can extend a class and can implement an interface but not both simultaneously.(FALSE)
5. An interface can implement any no.of interfaces at a time.(FALSE)
6. None of the above.

Ans: 6

Consider the expression X extends Y for which of the possibility of X and Y this expression is true?

1. Both x and y should be classes.
2. Both x and y should be interfaces.
3. **Both x and y can be classes or can be interfaces.**
4. No restriction.

Ans: 3

X extends Y, Z ?

X, Y, Z should be interfaces.

X extends Y implements Z ?

X, Y should be classes.

Z should be interface.

X implements Y, Z ?

X should be class.

Y, Z should be interfaces.

X implements Y extend Z ?

Example:

```
interface One
{
}
class Two
{
}
class Three implements One extends Two
{
}
Output:
Compile time error.
D:\Java>javac Three.java
Three.java:5: '{' expected
class Three implements One extends Two{
```

Interface methods:

- Every method present inside interface **is always public and abstract** whether we are declaring or not. Hence inside interface the following method declarations are equal.

```
void methodOne();
public Void methodOne();
abstract Void methodOne();           Equal
public abstract Void methodOne();
```

public: To make this method available for every implementation class.

abstract: Implementation class is responsible to provide implementation .

As every interface method **is always public and abstract** we can't use the following modifiers for interface methods.

Private, protected, final, static, synchronized, native, strictfp.

Inside interface which method declarations are valid?

1. public void methodOne(){ } (NOT VALID)
2. private void methodOne(); (NOT VALID)
3. public final void methodOne(); (NOT VALID)
4. public static void methodOne(); (NOT VALID)
5. public abstract void methodOne(); (VALID)

Ans: 5

Interface variables:

- An interface can contain variables.
- The main purpose of interface variables is to define **requirement level constants**.
- Every interface variable is always **public static and final** whether we are declaring or not.

Example:

```
interface interf
{
    int x=10;
}
```

public: To make it available for every implementation class.

static: Without existing object also we have to access this variable.

final: Implementation class can access this value but cannot modify.

Hence inside interface the following declarations are equal.

```
int x=10;
public int x=10;
static int x=10;
final int x=10;                                Equal
public static int x=10;
public final int x=10;
static final int x=10;
public static final int x=10;
```

- As every interface variable by default **public static final** we can't declare with the following modifiers.
 - Private
 - Protected
 - Transient
 - Volatile

- For the interface variables, compulsory we should perform initialization **at the time of declaration only** otherwise we will get compile time error.

Example:

```
interface Interf
{
int x;
}
Output:
Compile time error.
D:\Java>javac Interf.java
Interf.java:3: = expected
int x;
```

Which of the following declarations are valid inside interface ?

1. int x; (NOT VALID)
2. private int x=10; (NOT VALID)
3. public volatile int x=10; (NOT VALID)
4. public transient int x=10; (NOT VALID)
5. public static final int x=10; (VALID)

Ans: 5

Interface variables can be accessed from implementation class but cannot be modified.

Example:

```
interface Interf
{
    int x=10;
}
```

Example 1:

```
class Test implements Interf
{
    public static void main(String args[]){
        x=20;
        System.out.println("value of x"+x);
    }
}
output:
compile time error.
D:\Java>javac Test.java
Test.java:4: cannot assign a value to final variable x
x=20;
```

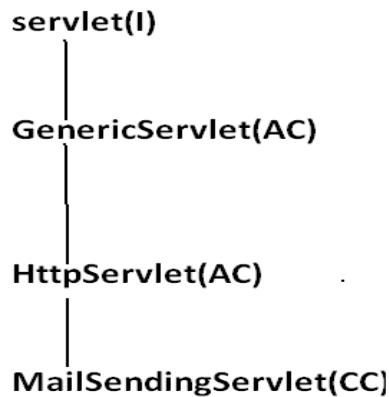
Example 2:

```
class Test implements Interf
{
    public static void main(String args[])
    {
        int x=20;
        //here we declaring the variable x.
        System.out.println(x);
    }
}
Output:
D:\Java>javac Test.java
D:\Java>java Test
20
```

**What is the difference between interface, abstract class and concrete class?
When we should go for interface, abstract class and concrete class?**

- If we don't know anything about implementation just we have requirement specification then we should go for **interface(I)**.
- If we are talking about implementation but not completely (**partial implementation**) then we should go for **abstract class(AC)**.
- If we are talking about **implementation completely** and ready to provide service then we should go for **concrete class(CC)**.

Example:



What is the Difference between interface and abstract class ?

interface	Abstract class
If we don't know anything about implementation just we have requirement specification then we should go for interface.	If we are talking about implementation but not completely (partial implementation) then we should go for abstract class.
Every method present inside interface is always public and abstract whether we are declaring or not.	Every method present inside abstract class need not be public and abstract .
We can't declare interface methods with the modifiers private, protected, final, static, synchronized, native, strictfp .	There are no restrictions on abstract class method modifiers .
Every interface variable is always public static final whether we are declaring or not.	Every abstract class variable need not be public static final .
Every interface variable is always public static final we can't declare with the following modifiers. Private, protected, transient, volatile.	There are no restrictions on abstract class variable modifiers.
For the interface variables compulsory we should perform initialization at the time of declaration otherwise we will get compile time error.	It is not require to perform initialization for abstract class variables at the time of declaration.
Inside interface we can't take static and instance blocks.	Inside abstract class we can take both static and instance blocks.
Inside interface we can't take constructor.	Inside abstract class we can take constructor.

We can't create object for abstract class but abstract class can contain constructor what is the need ?

Abstract class constructor will be executed when ever we are creating child class object to perform initialization of child object.

Example:

```
class Parent
{
    Parent()
    {
        System.out.println(this.hashCode());
    }
}
class child extends Parent
{
    child()
    {
        System.out.println(this.hashCode());
    }
}
```

```

class Test
{
    public static void main(String args[])
    {
        child c=new child();
        System.out.println(c.hashCode());
    }
}

```

Note : We can't create object for abstract class either directly or indirectly.

Every method present inside interface is abstract but in abstract class also we can take only abstract methods then what is the need of interface concept ?

We can replace interface concept with abstract class. But it is not a good programming practice.
We are misusing the roll of abstract class. It may create performance problems also.
(this is just like recruiting IAS officer for sweeping purpose)

Example :

<pre> interface X { ----- } class Test implements X { ----- } Test t=new Test(); //takes 2 sec 1) performance is high 2) while implementing X we can extend some other classes </pre>	<pre> abstract class X { ----- } class Test extends X { ----- } Test t=new Test(); //takes 20sec 1) performance is low 2) while extending X we can't extend any other classes </pre>
---	---

If every thing is abstract, then it is recommended to go for interface.

Why abstract class can contain constructor where as interface doesn't contain constructor ?

- The main purpose of constructor is to perform initialization of an object i.e., provide values for the instance variables, Inside interface **every variable is always static** and **there is no chance of existing instance variables**. Hence constructor is not required for interface.
- But **abstract class can contains instance variable** which are required for the child object to perform initialization for those instance variables constructor is required in the case of abstract class.

LinkedList in Java

Linked List is a part of the [Collection framework](#) present in [java.util package](#). This class is an implementation of the [LinkedList data structure](#) which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node. Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays. It also has few disadvantages like the nodes cannot be accessed directly instead we need to start from the head and follow through the link to reach to a node we wish to access.

Example: The following implementation demonstrates how to create and use a linked list.

//Program using LinkedList class

```
import java.util.*;
```

```
public class Test444 {
```

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

```
    {
```

```
        // Creating object of the
```

```
        // class linked list
```

```
        LinkedList<String> ll = new LinkedList<String>();
```

```
        // Adding elements to the linked list
```

```
        ll.add("A");
```

```
        ll.add("B");
```

```
        ll.addLast("C");
```

```
        ll.addFirst("D");
```

```
        ll.add(2, "E");
```

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

```
        ll.remove("B");
```

```
        ll.remove(3);
```

```
        ll.removeFirst();
```

```
        ll.removeLast();
```

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

```
    }
```

```
}
```

output:

```
student@student-HP-245-G6-Notebook-PC:~$ javac Test444.java
```

```
student@student-HP-245-G6-Notebook-PC:~$ java Test444
```

```
[D, A, E, B, C]
```

```
[A]
```

Java List

List in Java provides the facility to maintain the *ordered collection*. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list.

The List interface is found in the `java.util` package and inherits the Collection interface. It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions. The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector. The ArrayList and LinkedList are widely used in Java programming. The Vector class is deprecated since Java 5.

List Interface declaration

```
public interface List<E> extends Collection<E>
```

Java List Methods

Method	Description
<code>void add(int index, E element)</code>	It is used to insert the specified element at the specified position in a list.
<code>boolean add(E e)</code>	It is used to append the specified element at the end of a list.
<code>boolean addAll(Collection<? extends E> c)</code>	It is used to append all of the elements in the specified collection to the end of a list.
<code>boolean addAll(int index, Collection<? extends E> c)</code>	It is used to append all the elements in the specified collection, starting at the specified position of the list.
<code>void clear()</code>	It is used to remove all of the elements from this list.
<code>boolean equals(Object o)</code>	It is used to compare the specified object with the elements of a list.
<code>int hashCode()</code>	It is used to return the hash code value for a list.
<code>E get(int index)</code>	It is used to fetch the element from the particular position of the list.
<code>boolean isEmpty()</code>	It returns true if the list is empty, otherwise false.
<code>int lastIndexOf(Object o)</code>	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
<code>Object[] toArray()</code>	It is used to return an array containing all of the elements in this list in the correct order.
<code><T> T[] toArray(T[] a)</code>	It is used to return an array containing all of the elements in this list in the correct order.
<code>boolean contains(Object o)</code>	It returns true if the list contains the specified element
<code>boolean containsAll(Collection<?> c)</code>	It returns true if the list contains all the specified element
<code>int indexOf(Object o)</code>	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this

	element.
E remove(int index)	It is used to remove the element present at the specified position in the list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element.
boolean removeAll(Collection<?> c)	It is used to remove all the elements from the list.
void replaceAll(UnaryOperator<E> operator)	It is used to replace all the elements from the list with the specified element.
void retainAll(Collection<?> c)	It is used to retain all the elements in the list that are present in the specified collection.
E set(int index, E element)	It is used to replace the specified element in the list, present at the specified position.
void sort(Comparator<? super E> c)	It is used to sort the elements of the list on the basis of specified comparator.
Spliterator<E> spliterator()	It is used to create spliterator over the elements in a list.
List<E> subList(int fromIndex, int toIndex)	It is used to fetch all the elements lies within the given range.
int size()	It is used to return the number of elements present in the list.

Java List vs ArrayList

List is an interface whereas ArrayList is the implementation class of List.

How to create List

The ArrayList and LinkedList classes provide the implementation of List interface. Let's see the examples to create the List:

//Creating a List of type String using ArrayList

```
List<String> list=new ArrayList<String>();
```

//Creating a List of type Integer using ArrayList

```
List<Integer> list=new ArrayList<Integer>();
```

//Creating a List of type Book using ArrayList

```
List<Book> list=new ArrayList<Book>();
```

//Creating a List of type String using LinkedList

```
List<String> list=new LinkedList<String>();
```

In short, you can create the List of any type. The ArrayList<T> and LinkedList<T> classes are used to specify the type. Here, T denotes the type

Java List Example

Let's see a simple example of List where we are using the ArrayList class as the implementation.

```
import java.util.*;
public class ListExample1
{
    public static void main(String args[])
    {
        //Creating a List
        List<String> list=new ArrayList<String>();
        //Adding elements in the List
        list.add("Mango");
        list.add("Apple");
        list.add("Banana");
        list.add("Grapes");
        //Iterating the List element using for-each loop
        for(String fruit:list)
            System.out.println(fruit);

    }
}
```

Output:

```
student@student-HP-245-G6-Notebook-PC:~$ javac ListExample1.java
student@student-HP-245-G6-Notebook-PC:~$ java ListExample1
```

Mango

Apple

Banana

Grapes

How to convert Array to List

We can convert the Array to List by traversing the array and adding the element in list one by one using list.add() method. Let's see a simple example to convert array elements into List.

```
import java.util.*;
public class ArrayToListExample
{
    public static void main(String args[])
    {
        //Creating Array
        String[] array={"Java","Python","PHP","C++"};
        System.out.println("Printing Array: "+Arrays.toString(array));
        //Converting Array to List
        List<String> list=new ArrayList<String>();
        for(String lang:array)
        {
            list.add(lang);
        }
        System.out.println("Printing List: "+list);
    }
}
```

Output:

```
student@student-HP-245-G6-Notebook-PC:~$ javac ArrayToListExample.java
```

```
student@student-HP-245-G6-Notebook-PC:~$ java ArrayToListExample
```

```
Printing Array: [Java, Python, PHP, C++]
```

```
Printing List: [Java, Python, PHP, C++]
```

How to convert List to Array

We can convert the List to Array by calling the list.toArray() method. Let's see a simple example to convert list elements into array.

```
import java.util.*;
public class ListToArrayExample
{
    public static void main(String args[])
    {
        List<String> fruitList = new ArrayList<>();
        fruitList.add("Mango");
        fruitList.add("Banana");
        fruitList.add("Apple");
        fruitList.add("Strawberry");
        //Converting ArrayList to Array
        String[] array = fruitList.toArray(new String[fruitList.size()]);
        System.out.println("Printing Array: "+Arrays.toString(array));
        System.out.println("Printing List: "+fruitList);
    }
}
```

Output:

```
student@student-HP-245-G6-Notebook-PC:~$ javac ListToArrayExample.java
```

```
student@student-HP-245-G6-Notebook-PC:~$ java ListToArrayExample
```

```
Printing Array: [Mango, Banana, Apple, Strawberry]
```

```
Printing List: [Mango, Banana, Apple, Strawberry]
```

CHAPTER-4

INPUT/OUTPUT and EXCEPTION HANDLING



FILE HANDLING

The data stored in computer memory in two ways.

1) Temporary storage.

RAM is temporary storage. Whenever we are executing java program that memory is created, when program completes memory destroyed. This type of memory is called volatile memory.

2) Permanent storage.

When we write a program and save it in hard disk, that type of memory is called permanent storage. It is also known as non-volatile memory.

When we work with stored files, we need to follow following task.

- 1) Determine whether the file is there or not.
- 2) Open a file.
- 3) Read the data from the file.
- 4) Writing information to file.
- 5) Closing file.

Stream :-

Stream is a channel it supports continuous flow of data from one place to another place.

Java.io is a package that contains number of classes. By using those classes, we are able to send the data from one place to another place.

```
java
|---->io
    |-->InputStream(class)
    |-->OutputStream(class)
    |-->FileReader(class)
    |-->FileWriter(class)
    |-->Serializable(interface)
```

In java language we are transferring the data in the form of two ways:-

1. Byte format
2. Character format

Stream/channel:-

It is acting as medium by using stream or channel we are able to send particular data from one place to the another place.

Streams are two types:-

1. **Byte oriented stream.**(supports byte formatted data to transfer)
2. **Character oriented stream.**(supports character formatted data to transfer)

Byte oriented streams:-

Java.io.InputStream

byte channel:-

1)InputStream

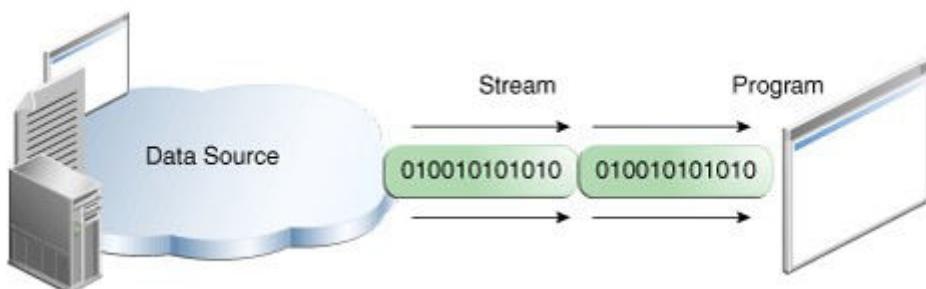
```
public native int read() throws java.io.IOException;
public void close() throws java.io.IOException;
```

2) FileOutputStream

```
public native void write(int) throws java.io.IOException;  
public void close() throws java.io.IOException;
```

To read the data from the destination file to the java application, we have to use **FileInputStream class**.

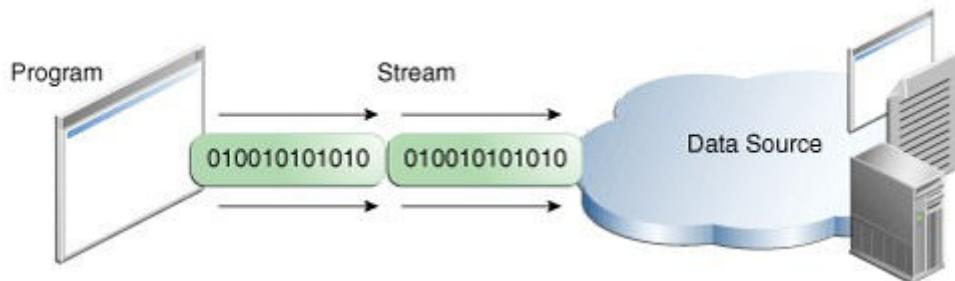
To read the data from the .txt file, we have to use **read()** method.



Java.io.FileOutputStream:-

To write the data to the destination file, we have to use the **FileOutputStream class**.

To write the data to the destination file, we have to use **write()** method.



Ex:- It will supports one **character** at a time.

```
import java.io.*;  
class Test  
{  
    public static void main(String[] args)throws Exception  
    {  
        //Byte oriented channel  
        FileInputStream fis = new FileInputStream("abc.txt");//read data from source file  
        FileOutputStream fos = new FileOutputStream("xyz.txt");//write data to target file  
        int c;  
        while((c=fis.read())!=-1)/  
        {  
            System.out.print((char)c);  
            fos.write(c);  
        }  
        System.out.println("read() & write operatoins are completed");  
        fis.close();  
        fos.close();  
    }  
}
```

File:

```
File f=new File("GameFile.txt");
```

- This line first checks whether GameFile.txt file is already available (or) not.
- If it is already available then "f" simply refers that file.
- If it is not already available then it won't create any physical file, just creates a java File object represents name of the file.

Example:

```
import java.io.*;  
class FileDemo  
{  
    public static void main(String[] args) throws IOException  
    {  
        File f=new File("cricket.txt");           //no file yet  
        System.out.println(f.exists());           //false  
        f.createNewFile();                      //make a file, "cricket.txt" which is assigned to f  
        System.out.println(f.exists());           //true  
    }  
}  
output :  
1st run :  
false  
true  
  
2nd run :  
true  
true
```

A java File object can represent a directory also.

Example:

```
import java.io.*;  
class FileDemo  
{  
    public static void main(String[ ] args) throws IOException  
    {  
        File f=new File("cricket123");           //no directory yet  
        System.out.println(f.exists());           //false  
        f.mkdir();                            //create an actual directory  
        System.out.println(f.exists());           //true  
    }  
}
```

Note: in UNIX everything is a file, java "file IO" is based on UNIX operating system hence in java also we can represent both files and directories by File object only.

File class constructors:

1. `File f=new File(String name);`
Creates a java File object to represent name of the file or directory **in current working directory**.
2. `File f=new File(String subdirname , String name);`
Creates a java File object to represent name of the file or directory present **in specified sub directory.(some other location)**
3. `File f=new File(File subdir, String name);`

Requirement: Write code to create a file named with demo.txt in current working directory.

Program:

```
import java.io.*;  
class FileDemo  
{  
    public static void main(String[] args) throws IOException  
    {  
        File f=new File("demo.txt");  
        f.createNewFile();  
    }  
}
```

Requirement: Write code to create a directory named with Ramu123 in current working directory and create a file named with abc.txt in that directory.

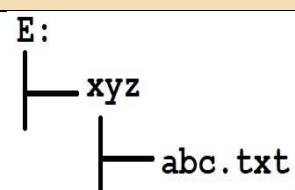
Program:

```
import java.io.*;  
class FileDemo  
{  
    public static void main(String[] args) throws IOException  
    {  
        File f1=new File("Ramu123"); //create an object  
        f1.mkdir(); //create an actual directory  
        File f2=new File("Ramu123","abc.txt");  
        f2.createNewFile(); //make a file, "Ramu123" which is assigned to f2  
    }  
}
```

Requirement: Write code to create a file named with abc.txt present in E:\xyz folder.

Program:

```
import java.io.*;  
class FileDemo  
{  
    public static void main(String[] args) throws IOException  
    {  
        File f=new File("E:\\xyz","abc.txt");  
        f.createNewFile();  
    }  
}
```



Import methods of File class:

Methods	Description
boolean exists();	Returns true if the specified file or directory available.
boolean createNewFile();	First This method will check whether the specified file is already available or not. If it is already available then this method simply returns false without creating any physical file. If this file is not already available then it will create a new file and returns true
boolean mkdir();	First This method will check whether the directory is already available or not. If it is already available then this method simply returns false without creating any directory. If this directory is not already available then it will create a new directory and returns true
boolean isFile();	Returns true if the specified File object represents a physical file.
String[] list();	It returns the names of all files and subdirectories present in the specified directory.
long length();	Returns the no of characters present in the file.
boolean isDirectory();	Returns true if the File object represents a directory.
boolean delete();	To delete a file or directory.

Requirement: Write a program to display the names of all files and directories present in c:\\ramu_classes.

```
import java.io.File;
import java.io.IOException;

public class FileDemo
{
    public static void main(String[] args) throws IOException
    {
        int count=0;
        File f=new File("c:\\ramu_classes");
        String[ ] s=f.list();           //list all files and directories in this directory

        for(String s1:s)               //for each string s1 in s
        {
            count++;
            System.out.println(s1);
        }

        System.out.println("total number : "+count);
    }
}
```

```
}
```

Requirement: Write a program to display only file names

```
import java.io.*;
class FileDemo
{
    public static void main(String[] args) throws IOException
    {
        int count=0;
        File f=new File("c:\\ramu_classes");
        String[ ] s=f.list( );

        for(String s1:s)
        {
            File f1=new File(f,s1);
            if(f1.isFile( ))
            {
                count++;
                System.out.println(s1);
            }
        }
        System.out.println("total number : "+count);
    }
}
```

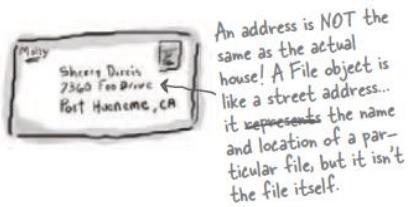
Requirement: Write a program to display only directory names

```
import java.io.*;
class FileDemo
{
    public static void main(String[ ] args) throws IOException
    {
        int count=0;
        File f=new File("c:\\ramu_classes");
        String[ ] s=f.list( );

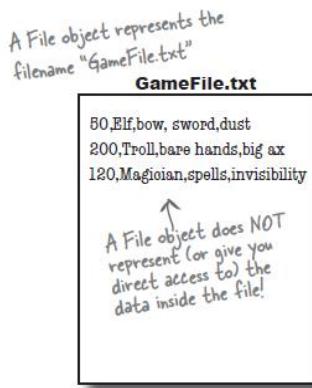
        for(String s1:s)
        {
            File f1=new File(f,s1);
            if(f1.isDirectory())
            {
                count++;
                System.out.println(s1);
            }
        }
        System.out.println("total number : "+count);
    }
}
```

Important points:

- ➔ A file object represents the name and path of a file or directory on disk.
- ➔ For example: /Users/Krishna/Data/GameFile.txt
- ➔ But it does NOT represent, or give us access to, the data in the file!



An address is NOT the same as the actual house! A File object is like a street address... it represents the name and location of a particular file, but it isn't the file itself.



A File object represents the filename "GameFile.txt"
GameFile.txt

50,Elf,bow,sword,dust

200,Troll,bare hands,big ax

120,Magician,spells,invisibility

A File object does NOT represent (or give you direct access to) the data inside the file!

FileReader:

We can use FileReader object to **read character data from the file**.

Constructors:

1. **FileReader fr=new FileReader(String file);**
2. **FileReader fr=new FileReader (File f);**

Methods:

1) int read();

It attempts to read **next character** from the file and return its ASCII **value**. If the next character is not available, then we will get **-1**.

```
int i=fr.read();
System.out.println((char)i);
```

As this method returns ASCII value, while printing we have to perform **type casting**.

2) int read(char[] ch);

It attempts to read **enough characters from the file into char[] array** and returns the **no of characters copied from the file into char[] array**.

```
File f=new File("abc.txt");
Char[ ] ch=new Char[(int)f.length()];
```

3) void close();

It is used to close the FileReader class.

Approach 1:

```
import java.io.*;
class FileReaderDemo
{
    public static void main(String[] args)throws IOException
    {
        FileReader fr=new FileReader("cricket.txt");           //just an object
        int i=fr.read();           //more amount of data
        while(i != -1)
        {
            System.out.print((char)i);           //typecasting mandatory
            i=fr.read();
        }
    }
}
```

Output:

Charan
Software solutions
ABC

Approach 2:

```
import java.io.*;
class FileReaderDemo
{
    public static void main(String[] args) throws IOException
    {
        File f=new File("cricket.txt");           //just an object
        FileReader fr=new FileReader(f);          //create a FileReader object
        char[ ] ch=new char[(int)f.length( )];    //small amount of data
        fr.read(ch);                            //read the whole file!
        for(char ch1:ch)                      //print the array
        {
            System.out.print(ch1);
        }
    }
}
```

Output:

XYZ

Software solutions.

Usage of *FileWriter* and *FileReader* is not recommended because :

1. While writing data by *FileWriter* compulsory we should insert **line separator(\n)** manually which is a bigger headache to the programmer.
2. While reading data by *FileReader* we have to read **character by character** instead of **line by line** which is not convenient to the programmer.
3. To overcome these limitations we should go for *BufferedWriter* and *BufferedReader* concepts.

FileWriter:

By using FileWriter object, we can write **character** data to the file.

Constructors:

```
FileWriter fw=new FileWriter(String fname);
```

→ It creates a new file. It gets file name in **string**.

```
FileWriter fw=new FileWriter(File f);
```

→ It creates a new file. It gets file name in File **object**.

The above 2 constructors meant for overriding.

Instead of **overriding**, if we want **append** operation then we should go for the following 2 constructors.

```
FileWriter fw=new FileWriter(String file, boolean append);
FileWriter fw=new FileWriter(File file, boolean append);
```

If the specified physical file is not already available, then these constructors will create that file.

Methods:

Method	Description
write(int ch);	To write a single character to the file.
write(char[] ch);	To write an array of characters to the file.
write(String text);	To write a String to the file.
flush();	To give the guarantee the total data include last character also written to the file.
close();	To close the stream.

Example:

```
import java.io.*;
class FileWriterDemo
{
    public static void main(String[ ] args) throws IOException
    {
        FileWriter fw=new FileWriter("cricket.txt",true); //create a FileWriter object
```

```
        fw.write(99);           //adding a single character
        fw.write("charan\nsoftware solutions"); //write characters to the file
        fw.write("\n");
        char[ ] ch={'a','b','c'};
        fw.write(ch);
        fw.write("\n");
        fw.flush();           //flush before closing
        fw.close();           //close file when done always.
    }
}
Output:
charan
software solutions
abc
```

Note :

- The main problem with FileWriter is “we have to insert line separator manually”, which is difficult to the programmer. ('\n')
- And even line separator varying from system to system.

BufferedWriter:

By using BufferedWriter object we can write character data to the file.

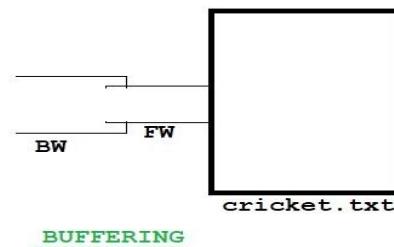
Constructors:

```
BufferedWriter bw=new BufferedWriter(writer w);  
BufferedWriter bw=new BufferedWriter(writer w, int buffersize);
```

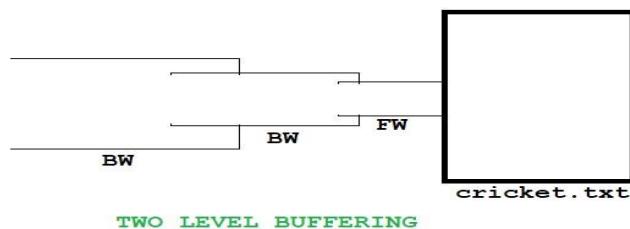
Note: BufferedWriter can't communicate directly with the file it can communicate via some writer object.

Which of the following declarations are valid?

1. BufferedWriter bw=new BufferedWriter("cricket.txt"); **(invalid)**
2. BufferedWriter bw=new BufferedWriter (new File("cricket.txt")); **(invalid)**
3. BufferedWriter bw=new BufferedWriter (new FileWriter("cricket.txt")); **(valid)**



4) **BufferedWriter bw=new BufferedWriter(new BufferedWriter(new FileWriter("cricket.txt"))); (valid)**



Methods:

1. **write(int ch);** //To write a single character to the file
2. **write(char[] ch);** //To write n array of characters
3. **write(String s);** //To write string to the file
4. **flush();**
5. **close();**
6. **newline();** //To insert a line separator

When compared with FileWriter which of the following capability is available extra as method form in BufferedWriter.

1. Writing data to the file.
2. Closing the file.
3. Flushing the file.
4. Inserting new line character.

Ans : 4

Example:

```
import java.io.*;
class BufferedWriterDemo
{
    public static void main(String[] args) throws IOException
    {
        FileWriter fw=new FileWriter("cricket.txt");
        BufferedWriter bw=new BufferedWriter(fw);
        bw.write(100);
        bw.newLine();
        char[ ] ch={'a','b','c','d'};
        bw.write(ch);
        bw.newLine();
        bw.write("Ramu");
        bw.newLine();
        bw.write("software solutions");
        bw.flush();
        bw.close();
    }
}
```

Output:



Note : Whenever we are closing BufferedWriter, automatically internal FileWriter will be closed and we are not required to close explicitly.

bw.close()	fw.close()	bw.close() fw.close()
✓	✗	✗

BufferedReader:

We can use BufferedReader to read character data from the file.

The main advantage of BufferedReader when compared with FileReader is we can read data line by line in addition to character by character.

Constructors:

```
BufferedReader br=new BufferedReader(Reader r);  
BufferedReader br=new BufferedReader(Reader r, int buffersize);
```

Note: BufferedReader **can't communicate directly with the File** and it **can communicate via some Reader object.**

Methods:

1. int read(); //to read a single character
2. int read(char[] ch); //to read n array of characters
3. void close();
4. **String readLine();**

It attempts to read next line and return it , from the File. if the next line is not available then this method returns **null**.

Example:

```
import java.io.*;  
class BufferedReaderDemo  
{  
    public static void main(String[] args) throws IOException  
    {  
        FileReader fr=new FileReader("cricket.txt");  
        BufferedReader br=new BufferedReader(fr);  
        String line=br.readLine();  
        while(line!=null)  
        {  
            System.out.println(line);  
            line=br.readLine();  
        }  
        br.close();  
    }  
}
```

Note: Whenever we are closing BufferedReader, automatically underlying FileReader will be closed, it is not required to close explicitly. Even this rule is applicable for BufferedWriter also.

fr.close();	br.close();	fr.close();
x	✓	x

The most enhanced Reader to read character data from the file is BufferedReader.

PrintWriter:

- This is the most enhanced Writer to write character data to the file.
- The main advantage of PrintWriter over FileWriter and BufferedWriter is
 - ➔ By using FileWriter and BufferedWriter we can write **only character data** to the File but
 - ➔ By using PrintWriter we can write **any type of data** to the File.

Constructors:

- 1) `PrintWriter pw=new PrintWriter(String fname);`
- 2) `PrintWriter pw=new PrintWriter(File f);`
- 3) `PrintWriter pw=new PrintWriter(Writer w);`

Note: PrintWriter can communicate directly with the File and can communicate via some Writer object also.

Methods:

1. `write(int ch);`
2. `write (char[] ch);`
3. `write(String s);`
4. `flush();`
5. `close();`
6. `print(char ch);`
7. `print (int i);`
8. `print (double d);`
9. `print (boolean b);`
10. `print (String s);`
11. `println(char ch);`
12. `println (int i);`
13. `println(double d);`
14. `println(boolean b);`
15. `println(String s);`

Example:

```
import java.io.*;
class PrintWriterDemo {
    public static void main(String[] args) throws IOException
    {
        FileWriter fw=new FileWriter("cricket.txt");
        PrintWriter out=new PrintWriter(fw);
```

```

        out.write(100);
        out.println(100);
        out.println(true);
        out.println('c');
        out.println("SaiCharan");
        out.flush();
        out.close();
    }
}
Output:
d100
true
c
SaiCharan

```

What is the difference between write(100) and print(100)?

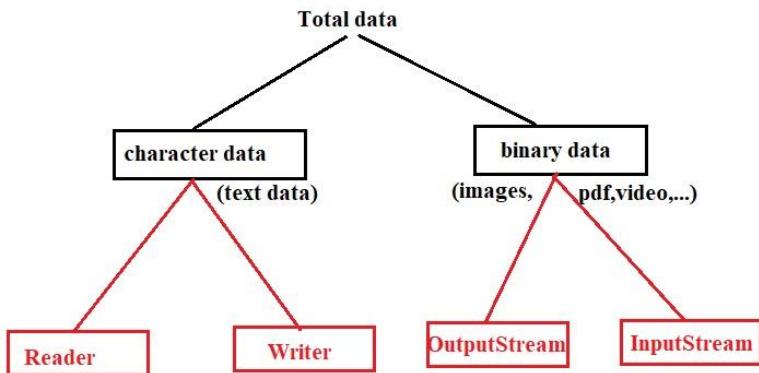
In the case of write(100) the corresponding character "d" will be added to the File but in the case of print(100) "100" value will be added directly to the File.

Note 1:

1. The most enhanced Reader to read character data from the File is BufferedReader.
2. The most enhanced Writer to write character data to the File is PrintWriter.

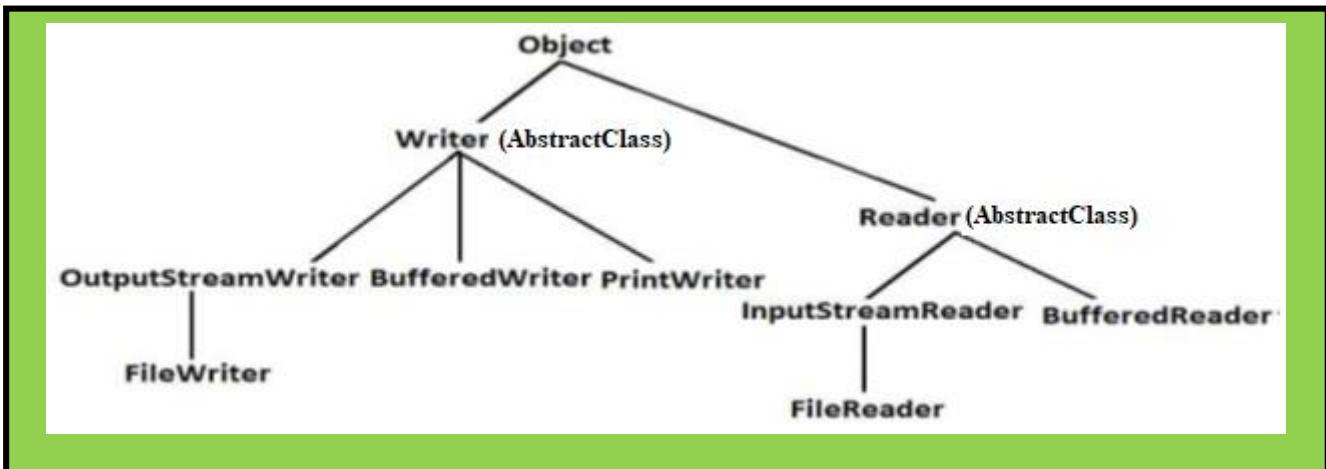
Note 2:

1. we can use Readers and Writers to handle character data.
we can use InputStreams and OutputStreams to handle binary data (like images, audio files, video files etc).



2. We can use OutputStream to write binary data to the File and we can use InputStream to read binary data from the File.

Diagram:



EXCEPTION HANDLING

Agenda

- 1. Introduction to Exception Handling.
- 2. Runtime stack mechanism
- 3. Default exception handling in java

Introduction

Exception: An unwanted unexpected event that disturbs normal flow of the program is called exception.

Example:

SleepingException
 TyrePunchuredException
 FileNotFoundException ...etc

- It is highly recommended to handle exceptions. The main objective of exception handling is graceful (normal) termination of the program.

What is the meaning of exception handling?

Exception handling doesn't mean repairing an exception. We have to define alternative way to continue rest of the program normally this way of "**defining alternative is nothing but exception handling**".

Example: Suppose our programming requirement is to read data from remote file locating at London at runtime if London file is not available our program should not be terminated abnormally.

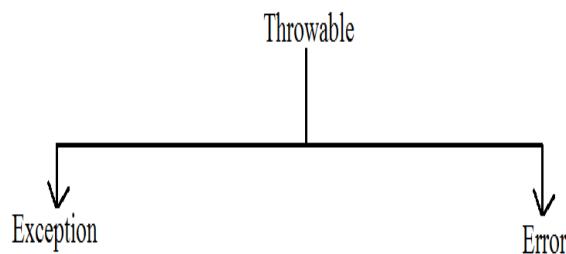
We have to provide a local file to continue rest of the program normally. This way of defining alternative is nothing but exception handling.

Example:

```
try
{
    read data from london file
}
catch(FileNotFoundException e)
{
    use local file and continue rest of the program normally
}
...
```

Exception hierarchy:

- **Throwable** acts as a root for exception hierarchy.
- **Throwable** class contains the following two child classes.



1) Exception: Most of the cases, exceptions are caused by our program and these are recoverable.

Ex :

If FileNotFoundException occurs, we can use local file and we can continue rest of the program execution normally.

2) Error: Most of the cases, errors are not caused by our program.

These are due to lack of system resources and these are non recoverable.

Ex :

If OutOfMemoryError occurs being a programmer, we can't do anything the program will be terminated abnormally.

System Admin or Server Admin is responsible to raise/increase heap memory.

Checked Vs Unchecked Exceptions:

→ The exceptions which are checked by the compiler for smooth execution of the program at runtime are called checked exceptions.

1. HallTicketMissingException
2. PenNotWorkingException
3. FileNotFoundException

→ The exceptions which are not checked by the compiler are called unchecked exceptions.

1. BombBlaustException
2. ArithmeticException
3. NullPointerException

Note: RuntimeException and its child classes, Error and its child classes are unchecked and all the remaining are considered as checked exceptions.

Note: Whether exception is checked or unchecked compulsory it should occur at runtime only there is no chance of occurring any exception at compile time.

Partially checked vs fully checked :

→ A checked exception is said to be fully checked, if and only if all its child classes are also checked.

Example:

- 1) IOException
- 2) InterruptedException

→ A checked exception is said to be partially checked, if and only if some of its child classes are unchecked.

Example:

Exception

Note: The only partially checked exceptions available in java are:

1. Throwable.
2. Exception.

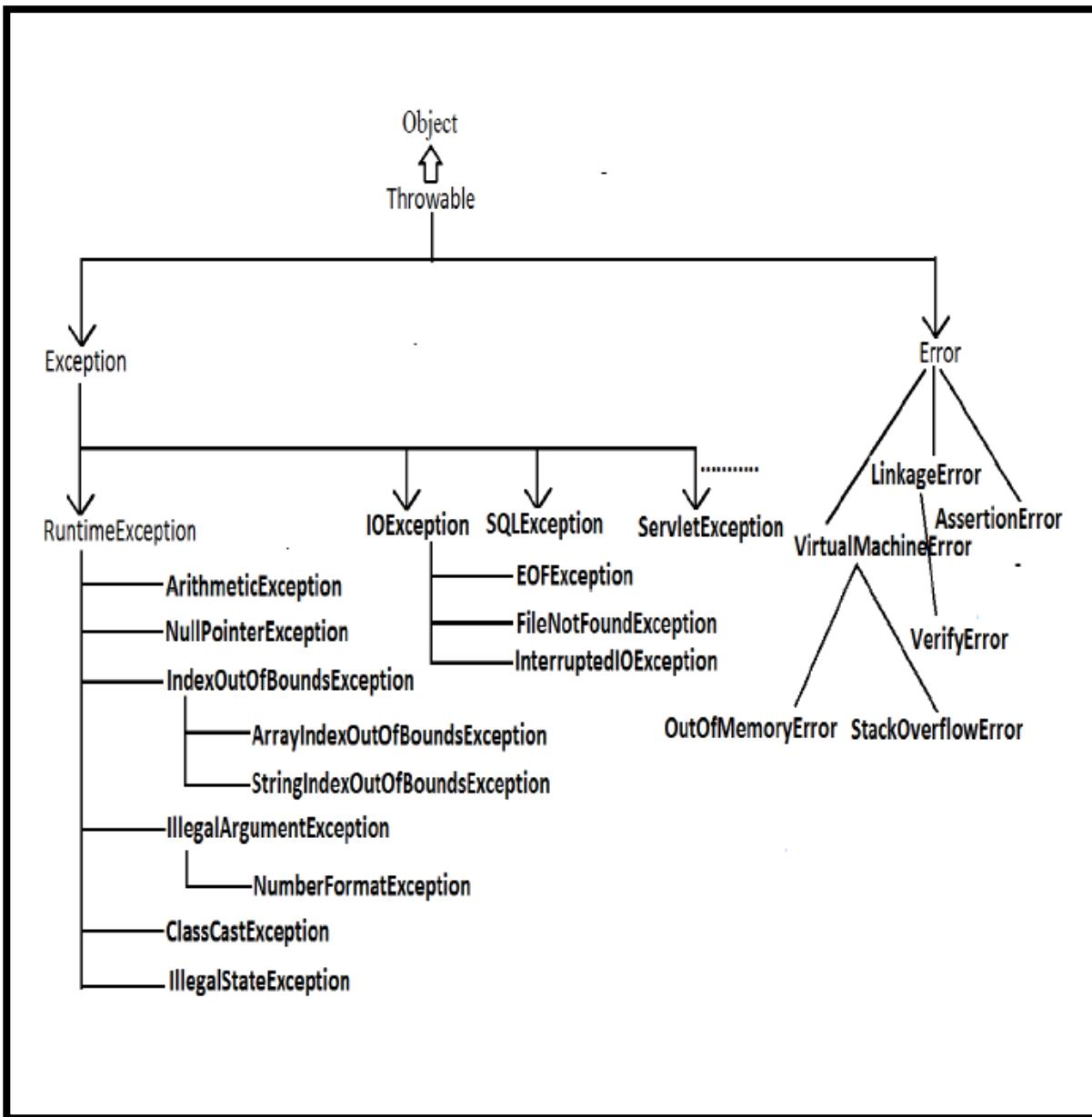


Figure: Exception Hierarchy

Q: Describe behavior of following exceptions ?

1. RuntimeException-----unchecked
2. Error-----unchecked
3. IOException-----fully checked
4. Exception-----partially checked
5. InterruptedException-----fully checked
6. Throwable-----partially checked
7. ArithmeticException -----unchecked
8. NullPointerException ----- unchecked
9. FileNotFoundException -----fully checked

Exception Handling

Agenda

1. Customized exception handling by try catch
2. Control flow in try catch
3. Try with multiple catch blocks
4. Finally
5. Difference between final, finally, finalize
6. Control flow in try catch finally
7. Control flow in nested try catch finally
8. Various possible combinations of try catch finally

1. Customized exception handling by try catch:

- It is highly recommended to handle exceptions.
- In our program the code which may cause an exception is called risky code, we have to place risky code inside try block and the corresponding handling code inside catch block.

Example:

```
try
{
    risky code
}
catch(Exception e)
{
    handling code
}
```

Without try catch	With try catch
<pre>class Test { psvm(String[] args) { S.o.p("statement1"); S.o.p(10/0); S.o.p("statement3"); } } output: statement1 RE:AE:/by zero at Test.main()</pre>	<pre>class Test { psvm(String[] args) { S.o.p("statement1"); try { S.o.p(10/0); } catch(ArithmeticException e) { S.o.p(10/2); } } }</pre>

Abnormal termination.

```
s.o.p("statement3");
}
}
Output:
statement1
5
statement3
```

Normal termination.

2. Control flow in try catch:

```
try
{
    statement1;
    statement2;
    statement3;
}
catch(X e)
{
    statement4;
}
statement5;
```

- **Case 1:** There is no exception.
1, 2, 3, 5 normal termination.
- **Case 2:** if an exception raised at statement 2 and corresponding catch block matched 1, 4, 5 normal termination.
- **Case 3:** if an exception raised at statement 2 but the corresponding catch block not matched , 1 followed by abnormal termination.
- **Case 4:** if an exception raised at statement 4 or statement 5 then it's always abnormal termination of the program.

Note:

1. Within the try block if anywhere an exception raised then rest of the try block won't be executed even though we handled that exception. Hence we have to place/take only risk code inside try and length of the try block should be as less as possible.
2. If any statement which raises an exception and it is not part of any try block then it is always abnormal termination of the program.
3. There may be a chance of raising an exception inside catch and finally blocks also in addition to try block.

3.Try with multiple catch blocks:

The way of handling an exception is varied from exception to exception. Hence for every exception type it is recommended to take a separate catch block. That is try with multiple catch blocks is possible and recommended to use.

Example:

<pre> try { . . } catch(Exception e) { default handler } </pre>	<pre> try { . . } catch(FileNotFoundException e) { use local file } catch(ArithmetricException e) { perform these Arithmetric operations } catch(SQLEception e) { don't use oracle db, use mysqlDb } catch(Exception e) { default handler } </pre>
---	---

This approach is not recommended because for any type of Exception we are using the same catch block.

This approach is highly recommended because for any exception raise we are defining a separate catch block.

If try with multiple catch blocks present then order of catch blocks is very important. It should be from child to parent by mistake if we are taking from parent to child then we will get Compile time error saying, "exception xxx has already been caught".

Example:

```

class Test
{
    psvm(String[] args)
    {
        try
        {
            System.out.println(10/0);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
        catch(ArithmeticException e)
        {
            e.printStackTrace();
        }
    }
}

CE:exception
java.lang.ArithmetiсException has already been caught

```

```

class Test
{
    psvm(String[] args)
    {
        try
        {
            System.out.println(10/0);
        }
        catch(ArithmeticException e)
        {
            e.printStackTrace();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}

Output:
Compile successfully.

```

4.Finally block:

- It is never recommended to take clean up code inside try block because there is no guarantee for the execution of every statement inside a try.
- It is never recommended to place clean up code inside catch block because if there is no exception then catch block won't be executed.
- We require some place to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether handled or not handled such type of place is nothing but finally block.
- Hence the main objective of finally block is to maintain cleanup code.

Example:

```
try
{
    risky code
}
catch(x e)
{
    handling code
}
finally
{
    cleanup code
}
```

The speciality of finally block is it will be executed always irrespective of whether the exception raised or not raised and whether handled or not handled.

Example 1:

```
class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("try block executed");
        }
        catch(ArithmetricException e)
        {
            System.out.println("catch block executed");
        }
        finally
        {
            System.out.println("finally block executed");
        }
    }
}
```

Output:

```
Try block executed
Finally block executed
```

Example 2:

```
class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("try block executed");
            System.out.println(10/0);
        }
        catch(ArithmetricException e)
```

```

        {
            System.out.println("catch block executed");
        }
    finally
    {
        System.out.println("finally block executed");
    }
}
}
Output:
Try block executed
Catch block executed
Finally block executed
Example 3:
class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("try block executed");
            System.out.println(10/0);
        }
        catch(NullPointerException e)
        {
            System.out.println("catch block executed");
        }
    finally
    {
        System.out.println("finally block executed");
    }
}
}
Output:
Try block executed
Finally block executed
Exception in thread "main" java.lang.ArithmetricException: / by zero
at Test.main(Test.java:8)

```

5. Difference between final, finally, and finalize:

Final:

- Final is the modifier applicable for class, methods and variables.
- If a class declared as the final then child class creation is not possible.
- If a method declared as the final then overriding of that method is not possible.

- If a variable declared as the final then reassignment is not possible.

Finally:

- It is the block always associated with try catch to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether handled or not handled.
- There is only one situation where the finally block won't be executed is whenever we are using **System.exit(0)** method.

Finalize:

It is a method which should be called by garbage collector always just before destroying an object to perform cleanup activities.

Note: To maintain clean up code finally block is recommended over finalize() method because we can't expect exact behavior of GC.

6. Control flow in try catch finally:

Example:

```
class Test
{
    public static void main(String[] args)
    {
        Try
        {
            System.out.println("statement1");
            System.out.println("statement2");
            System.out.println("statement3");
        }
        catch(Exception e)
        {
            System.out.println("statement4");
        }
        finally
        {
            System.out.println("statement5");
        }
        System.out.println("statement6");
    }
}
```

- **Case 1:** If there is no exception. 1, 2, 3, 5, 6 normal termination.
- **Case 2:** if an exception raised at statement 2 and corresponding catch block matched. 1,4,5,6 normal terminations.
- **Case 3:** if an exception raised at statement 2 and corresponding catch block is not matched. 1,5 abnormal termination.
- **Case 4:** if an exception raised at statement 4 then it's always abnormal termination but before the finally block will be executed.

- **Case 5:** if an exception raised at statement 5 or statement 6 its always abnormal termination.

7. Control flow in nested try catch finally:

Example:

```
class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("statement1");
            System.out.println("statement2");
            System.out.println("statement3");
            try
            {
                System.out.println("statement4");
                System.out.println("statement5");
                System.out.println("statement6");
            }
            catch (ArithmaticException e)
            {
                System.out.println("statement7");
            }
            finally
            {
                System.out.println("statement8");
            }
            System.out.println("statement9");
        }
        catch (Exception e)
        {
            System.out.println("statement10");
        }
        finally
        {
            System.out.println("statement11");
        }
        System.out.println("statement12");
    }
}
```

- **Case 1:** if there is no exception. 1, 2, 3, 4, 5, 6, 8, 9, 11, 12 normal termination.
- **Case 2:** if an exception raised at statement 2 and corresponding catch block matched 1,10,11,12 normal terminations.
- **Case 3:** if an exception raised at statement 2 and corresponding catch block is not matched 1, 11 abnormal termination.
- **Case 4:** if an exception raised at statement 5 and corresponding inner catch has matched 1, 2, 3, 4, 7, 8, 9, 11, 12 normal termination.

- **Case 5:** if an exception raised at statement 5 and inner catch has not matched but outer catch block has matched. 1, 2, 3, 4, 8, 10, 11, 12 normal termination.
- **Case 6:** if an exception raised at statement 5 and both inner and outer catch blocks are not matched. 1, 2, 3, 4, 8, 11 abnormal termination.
- **Case 7:** if an exception raised at statement 7 and the corresponding catch block matched 1, 2, 3, 4, 5, 6, 8, 10, 11, 12 normal termination.
- **Case 8:** if an exception raised at statement 7 and the corresponding catch block not matched 1, 2, 3, 4, 5, 6, 8, 11 abnormal terminations.
- **Case 9:** if an exception raised at statement 8 and the corresponding catch block has matched 1, 2, 3, 4, 5, 6, 7, 10, 11, 12 normal termination.
- **Case 10:** if an exception raised at statement 8 and the corresponding catch block not matched 1, 2, 3, 4, 5, 6, 7, 11 abnormal terminations.
- **Case 11:** if an exception raised at statement 9 and corresponding catch block matched 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12 normal termination.
- **Case 12:** if an exception raised at statement 9 and corresponding catch block not matched 1, 2, 3, 4, 5, 6, 7, 8, 11 abnormal termination.
- **Case 13:** if an exception raised at statement 10 is always abnormal termination but before that finally block 11 will be executed.
- **Case 14:** if an exception raised at statement 11 or 12 is always abnormal termination.

Note: if we are not entering into the try block then the finally block won't be executed. Once we entered into the try block without executing finally block we can't come out.

We can take try-catch inside try i.e., nested try-catch is possible

The most specific exceptions can be handled by using inner try-catch and generalized exceptions can be handle by using outer try-catch.

```
Example:
class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println(10/0);
        }
        catch(ArithmaticException e)
        {
            System.out.println(10/0);
        }
        finally{
            String s=null;
            System.out.println(s.length());
        }
    }
}
output :
RE:NullPointerException
```

Note: Default exception handler can handle only one exception at a time and that is the most recently raised exception.

8. Various possible combinations of try catch finally:

1. Whenever we are writing try block compulsory we should write either catch or finally.
i.e., try without catch or finally is invalid.
2. Whenever we are writing catch block compulsory we should write try.
i.e., catch without try is invalid.
3. Whenever we are writing finally block compulsory we should write try.
i.e., finally without try is invalid.
4. In try-catch-finally order is important.
5. With in the try-catch -finally blocks we can take try-catch-finally.
i.e., nesting of try-catch-finally is possible.
6. For try-catch-finally blocks curly braces are mandatory.

```
try {}  
catch (X e) {}
```



```
try {}  
catch (X e) {}  
catch (Y e) {}
```



```
try {}  
catch (X e) {}  
catch (X e) {} //CE:exception ArithmeticException has already been caught
```



```
try {}  
catch (X e) {}  
finally {}
```



```
try {}  
finally {}
```



```
try {} //CE: 'try' without 'catch', 'finally' or resource declarations
```



```
catch (X e) {} //CE: 'catch' without 'try'
```



```
finally {} //CE: 'finally' without 'try'
```



```
try {} //CE: 'try' without 'catch', 'finally' or resource declarations  
System.out.println("Hello");  
catch {} //CE: 'catch' without 'try'
```

X

```
try {}  
catch (X e) {}  
System.out.println("Hello");  
catch (Y e) {} //CE: 'catch' without 'try'
```

X

```
try {}  
catch (X e) {}  
System.out.println("Hello");  
finally {} //CE: 'finally' without 'try'
```

X

```
try {}  
finally {}  
catch (X e) {} //CE: 'catch' without 'try'
```

X

```
try {}  
catch (X e) {}  
try {}  
finally {}
```

✓

```
try {}  
catch (X e) {}  
finally {}  
finally {} //CE: 'finally' without 'try'
```

X

```
try {}  
catch (X e) {  
try {}  
catch (Y e1) {}  
}
```

✓

```
try {  
try {} //CE: 'try' without 'catch', 'finally' or resource declarations  
}  
catch (X e) {}
```

X

```
try //CE: '{' expected  
System.out.println("Hello");  
catch (X e1) {} //CE: 'catch' without 'try'
```

X

```
try {}  
catch (X e) //CE:{' expected  
System.out.println("Hello");
```

X

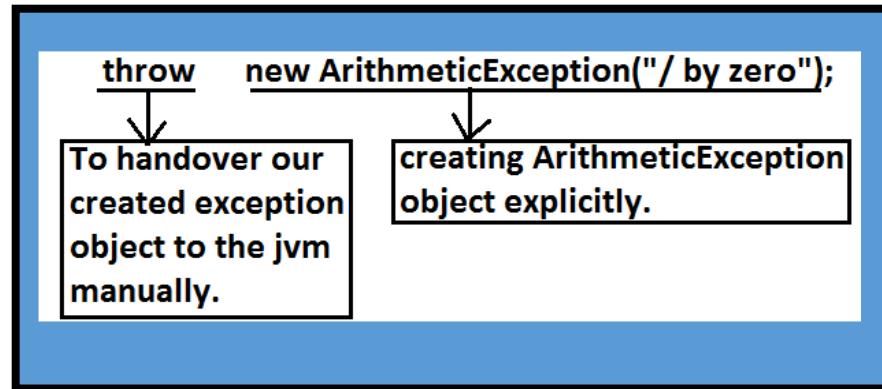
```
try {}  
catch (NullPointerException e1) {}  
finally //CE: '{' expected  
System.out.println("Hello");
```

X

Throw statement:

Sometimes we can create Exception object explicitly and we can hand over to the JVM manually by using throw keyword.

Example:



The result of following 2 programs is exactly same.

```
class Test
{
    p s v main(String[] args)
    {
        S.o.p(10/0);
    }
}
```

In this case creation of arithmeticException object and handover to the jvm will be performed automatically by the main() method.

```
class Test
{
    public static void main(String[] args)
    {
        throw new ArithmeticException("/ by zero");
    }
}
```

In this case we are creating exception object explicitly and handover to the JVM manually.

Note: In general we can use throw keyword for customized exceptions but not for predefined exceptions.

Case 1:**throw e;**

If e refers null then we will get NullPointerException.

Example:

```
class Test3
{
    static ArithmeticException e=new ArithmeticException();
    public static void main(String[] args)
    {
        throw e;
    }
}
```

Output:

Runtime exception: Exception in thread "main"
java.lang.ArithmaticException

```
class Test3
{
    static ArithmeticException e;
    public static void main(String[] args)
    {
        throw e;
    }
}
```

Output:

Exception in thread "main"
java.lang.NullPointerException
at Test3.main(Test3.java:5)

Case 2:

After throw statement we can't take any statement directly otherwise we will get compile time error saying unreachable statement.

Example:

```
class Test3
{
    public static void main(String[] args){
        System.out.println(10/0);
        System.out.println("hello");
    }
}
```

Output:

Runtime error: Exception in thread "main"
java.lang.ArithmaticException: / by zero
at Test3.main(Test3.java:4)

```
class Test3
{
    public static void main(String[] args){
        throw new ArithmeticException("/ by zero");
        System.out.println("hello");
    }
}
```

Output:

Compile time error.
Test3.java:5: unreachable statement
System.out.println("hello");

Case 3:

We can use throw keyword only for Throwable types otherwise we will get compile time error saying incomputable types.

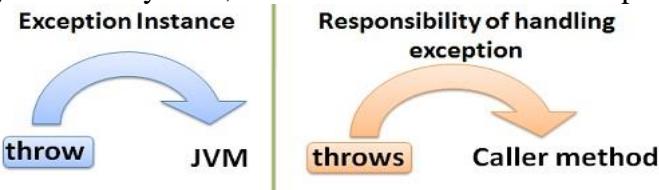
Example:

```
class Test3
{
    public static void main(String[] args)
    {
        throw new Test3();
    }
}
Output:
Compile time error.
Test3.java:4: incompatible types
found   : Test3
required: java.lang.Throwable
throw new Test3();
```

```
class Test3 extends RuntimeException
{
    public static void main(String[] args)
    {
        throw new Test3();
    }
}
Output:
Runtime error: Exception in thread "main"
Test3
    at Test3.main(Test3.java:4)
```

Throws statement:

In our program if there is any chance of raising checked exception, compulsory we should handle either by try catch or by throws keyword, otherwise the code won't compile.



Example:

```
import java.io.*;
class Test3
{
    public static void main(String[] args)
    {
        PrintWriter out=new PrintWriter("abc.txt");
        out.println("hello");
    }
}
```

CE :

Unreported exception java.io.FileNotFoundException;
must be caught or declared to be thrown.

Example:

```
class Test3
{
    public static void main(String[] args)
    {
        Thread.sleep(5000);
    }
}
```

Unreported exception java.lang.Interruptedexception;
must be caught or declared to be thrown.

We can handle this compile time error by using the following 2 ways.

Example:

By using try catch	By using throws keyword
<pre>class Test3 { psvm(String[] args) { try { Thread.sleep(5000); } catch(InterruptedException e){ } } }</pre> <p>Output: Compile and running successfully</p>	<p>We can use throws keyword to delineate the responsibility of exception handling to the caller method. Then caller method is responsible to handle that exception.</p> <pre>class Test3 { psvm(String[] args) throws InterruptedException { Thread.sleep(5000); } }</pre> <p>Output: Compile and running successfully</p>

Note :

- Hence the main objective of "throws" keyword is to delineate the responsibility of exception handling to the caller method.
- "throws" keyword required only checked exceptions. Usage of throws for unchecked exception there is no use.
- "throws" keyword required only to convene complier. Usage of throws keyword doesn't prevent abnormal termination of the program.
Hence recommended to use try-catch over throws keyword.

Example:

```
class Test
{
    public static void main(String[] args) throws InterruptedException
    {
        doStuff();
    }
    public static void doStuff() throws InterruptedException
    {
        doMoreStuff();
    }
    public static void doMoreStuff() throws InterruptedException
    {
        Thread.sleep(5000);
    }
}
```

Output:**Compile and running successfully.**

In the above program, if we are removing at least **one throws** keyword, then the program won't compile.

Case 1:

we can use throws keyword **only for Throwable types** otherwise we will get compile time error saying "**incompatible types**".

Example:

```
class Test3
{
    psvm(String[] args) throws Test3
    {
    }
}
Output:
Compile time error
Test3.java:2: incompatible types
found   : Test3
```

```
class Test3 extends RuntimeException
{
    psvm(String[] args) throws Test3
    {
    }
}
Output:
Compile and running successfully.
```

```
required: java.lang.Throwable
public static void main(String[] args)
    throws Test3
```

Case 2: Example:

```
class Test3
{
    public static void main(String[] args)
    {
        throw new Exception();
    }
}
Output:
Compile time error.
Test3.java:3: unreported exception
    java.lang.Exception;
must be caught or declared to be thrown
```

```
class Test3
{
    public static void main(String[] args)
    {
        throw new Error();
    }
}
Output:
Runtime error
Exception in thread "main" java.lang.Error
at Test3.main(Test3.java:3)
```

Case 3:

In our program with in the try block, if there is no chance of rising an exception then we can't write catch block for that exception. otherwise we will get compile time error saying “**exception XXX is never thrown in body of corresponding try statement**”. But this rule is applicable only for fully checked exception.

Example:

```
class Test
{
    public static void main(String[] args){
        try{
            System.out.println("hello");
        }
        catch(Exception e)
        {} output:
        } hello
    } partial checked
```

```
class Test
{
    public static void main(String[] args){
        try{
            System.out.println("hello");
        }
        catch(ArithmeticException e)
        {} output:
        } hello
    } unchecked
```

```
class Test
{
    public static void main(String[] args){
        try{
            System.out.println("hello");
        }
        catch(java.io.IOException e)
        {} output:
        } compile time error
    } fully checked
```

```
class Test
{
    public static void main(String[] args){
        try{
            System.out.println("hello");
        }
        catch(InterruptedException e)
        {} output:
        } compile time error
    } Fully checked
```

```
class Test
{
    public static void main(String[] args){
        try{
            System.out.println("hello");
        }
        catch(Error e)
        {} output:
        } compile successfully
    } unchecked
```

Case 4:

We can use throws keyword only for constructors and methods but not for classes.

Example:

```
class Test throws Exception    //invalid
{
    Test() throws Exception          //valid
    {}
    methodOne() throws Exception   //valid
    {
    }
}
```

Exception handling keywords summary:

1. **try:** To maintain risky code.
2. **catch:** To maintain handling code.
3. **finally:** To maintain cleanup code.
4. **throw:** To handover our created exception object to the JVM manually.
5. **throws:** To delegate responsibility of exception handling to the caller method.

Various possible compile time errors in exception handling:

1. Exception XXX has already been caught.
2. Unreported exception XXX must be caught or declared to be thrown.
3. Exception XXX is never thrown in body of corresponding try statement.
4. Try without catch or finally.
5. Catch without try.
6. Finally without try.
7. Incompatible types.
Found:test
Required:java.lang.Throwable;
8. Unreachable statement.

DIFFERENCES BETWEEN THROW AND THROWS:

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Throw is followed by an instance.	Throws is followed by class.
3)	Throw is used within the method.	Throws is used with the method signature.
4)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method() throws IOException,SQLException.

Customized Exceptions (User defined Exceptions):

Sometimes we can create our own exception to meet our programming requirements. Such type of exceptions are called customized exceptions (user defined exceptions).

Example:

1. InSufficientFundsException
2. TooYoungException
3. TooOldException

Program:

```
class TooYoungException extends RuntimeException
{
    TooYoungException(String s)
    {
        super(s);
    }
}
class TooOldException extends RuntimeException
{
    TooOldException(String s)
    {
        super(s);
    }
}
class CustomizedExceptionDemo
{
public static void main(String[ ] args)
{
    int age=Integer.parseInt(args[0]);
    if(age>60)
    {
        throw new TooYoungException("please wait some more time.... u will get best match");
    }
    else if(age<18)
    {
        throw new TooOldException("u r age already crossed....no chance of getting married");
    }
    else
    {
        System.out.println("you will get match details soon by e-mail");
    }
}
```

Output:

1)E:\Sai>java CustomizedExceptionDemo 61
Exception in thread "main" TooYoungException:
please wait some more time.... u will get best match
at CustomizedExceptionDemo.main(CustomizedExceptionDemo.java:21)

2)E:\Sai>java CustomizedExceptionDemo 27
You will get match details soon by e-mail

3)E:\Sai>java CustomizedExceptionDemo 9
Exception in thread "main" TooOldException:
u r age already crossed....no chance of getting married
at CustomizedExceptionDemo.main(CustomizedExceptionDemo.java:25)

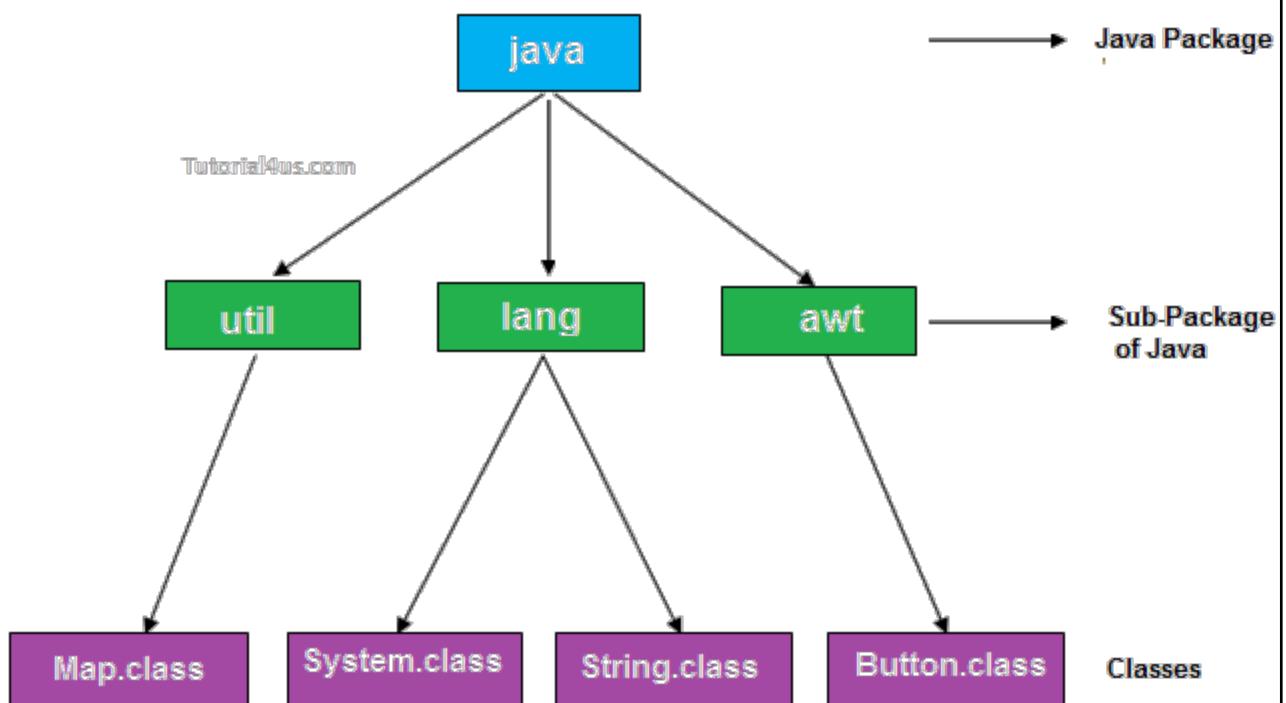
Note: It is highly recommended to maintain our customized exceptions as unchecked by extending RuntimeException.
We can catch any Throwable type including Errors also.

Example:

```
try
{
}
catch(Error e)    valid
{}
```

CHAPTER-5

PACKAGES AND MULTITHREADING



Packages

Introduction to packages:

- 1) The package contains group of related **classes and interfaces**.
- 2) The package is an encapsulation mechanism it is binding the related classes and interfaces.
- 3) We can declare a package with the help of package keyword.
- 4) Package is nothing but physical directory structure and it is providing clear-cut separation between the project modules.
- 5) Whenever we are dividing the project into the packages(modules) the sharability of the project will be increased.

Syntax:-

```
Package package_name;  
Ex:-    package com.dss;
```

The packages are divided into two types

- 1) Predefined packages
- 2) User defined packages

Predefined packages:-

The java predefined packages are introduced by sun peoples these packages contains predefined classes and interfaces.

Ex:- java.lang
 Java.io
 Java.awt
 Java.util
 Java.netetc

Java.lang:-

The most commonly required classes and interfaces to write a sample program is encapsulated into a separate package is called java.lang package.

Ex:- String(class)
 StringBuffer(class)
 Object(class)
 Runnable(interface)
 Cloneable(interface)

Note:-

the default package in the java programming is java.lang if we are importing or not importing by default this package is available for our programs.

Java.io package:-

The classes which are used to perform the input output operations that are present in the java.io packages.

Ex:- FileInputStream(class)

FileOutputStream(class)
FileWriter(class)
FileReader(class)

Java.net package:-

The classes which are required for connection establishment in the network those classes are present in the java.net package.

Ex:- Socket

ServerSocket
InetAddress
URL

Java.awt package:-

The classes which are used to prepare graphical user interface those classes are present in the java.awt package.

Ex: Button(class)
Checkbox(class)
Choice(Class)
List(class)

User defined packages:-

- 1) The packages which are declared by the user are called user defined packages.
- 2) In the single source file it is possible to take the only one package. If we are trying to take two packages at that situation the compiler raise a compilation error.
- 3) In the source file it is possible to take single package.
- 4) While taking package name we have to follow some coding standards.

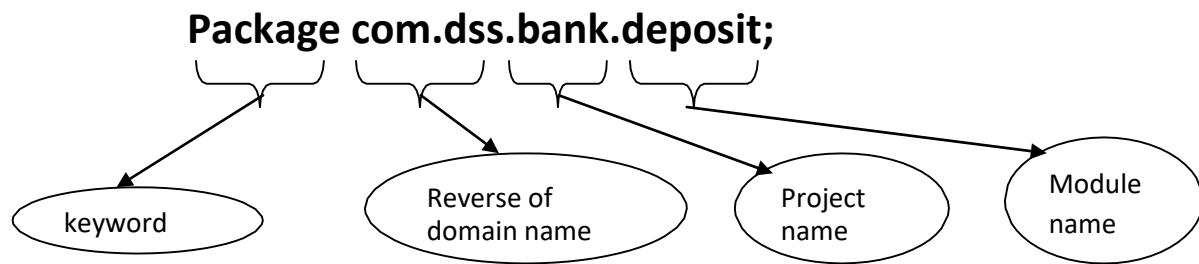
Whenever we taking package name don't take the names like pack1, pack2, sandhya, sri..... these are not a proper coding formats.

Rules to follow while taking package name:-(not mandatory but we have to follow)

- 1) The package name is must reflect with your organization name. The name is reverse of the organization domain name.
Domain name:- **www.dss.com**
Package name:- **Package com.dss;**
- 2) Whenever we are working in particular project(Bank) at that moment we have to take the package name is as fallows.
Project name :- **Bank**
package :- **Package com.dss.Bank;**
- 3) The project contains the module (deposit) at that situation our package name should reflect with the module name also.

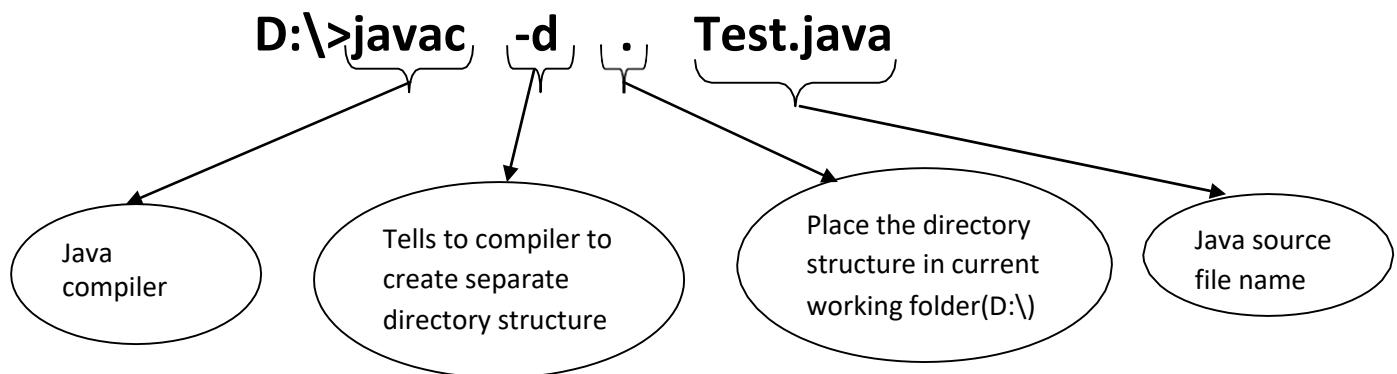
Domain name:- **www.dss.com**
Project name:- **Bank**
Module name:- **deposit**
package name:- **Package com.dss.bank.deposit;**

For example the source file contains the package structure is like this:-

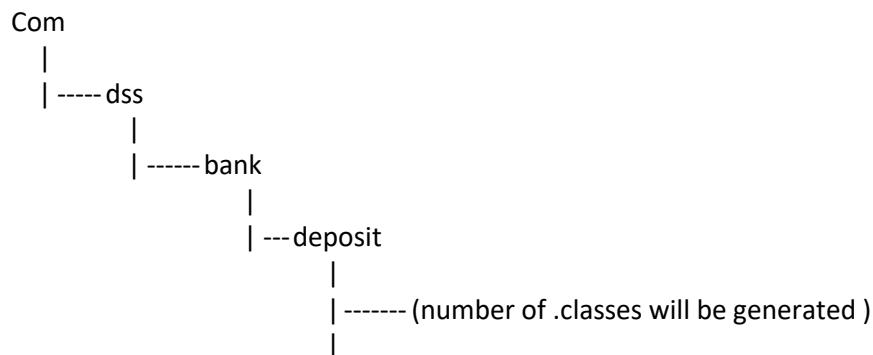


Note:-

If the source file contains the package statement then we have to compile that program with the help of following statements.



After compilation of the code the folder structure is as shown below.



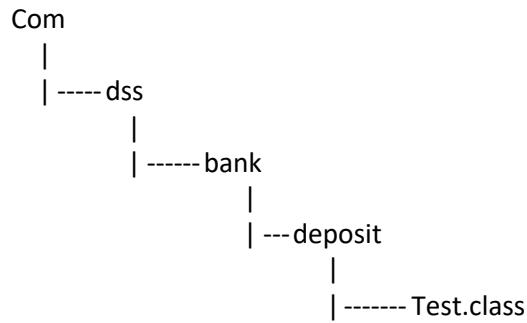
Note :-

If it a predefined package or user defined package the packages contains number of classes.

Ex 1:-

```
package com.dss.bank.deposit;
class Test
{
    public static void main(String[] args)
    {
        System.out.println("package example program");
    }
}
```

Compilation : javac -d . Test.java



Execution :java com.dss.bank.deposit.Test

Ex:- (compilation error)

```
package com.dss.bank.deposit;
package com.dss.online.corejava;
class Test
{
    public static void main(String[] args)
    {
        System.out.println("package example program");
    }
}
```

Reason:-

Inside the source file it is possible to take the single package not possible to take the multiple packages.

Ex 2:-

```
package com.tcs.OnlineExam.corejava;
class Test
{
    public static void main(String[] args)
    {
        System.out.println("package example program");
    }
}
```

```
}
```

```
class A
```

```
{
```

```
}
```

```
class B
```

```
{
```

```
}
```

```
class C
```

```
{
```

```
}
```

Compilation :- javac -d . Test.java

```
Com
 |
 | ----tcs
 |
 | -----OnlineExam
 |
 | ---corejava
 |
 | -----Test.class
 | -----A.class
 | -----B.class
 | -----C.class
```

Execution :- java com.tcs.onlineexam.Test

Note:-

The package contains any number of .classes the .class files generation totally depends upon the number of classes present on the source file.

Import session:-

The main purpose of the import session is to make available the java predefined support into our program.

Predefined packages support:-

Ex1:-

```
Import java.lang.String;
```

String is a predefined class to make available predefined string class to the our program we have to use import session.

Ex 2:-

```
Import java.awt.*;
```

To make available all predefined class present in the awt package into our program. That * represent all the classes present in the awt package.

User defined packages support:-

I am taking two user defined packages are

1) Package pack1;

 Class A

 {

 }

 Class B

 {

 }

2) Package pack2

 Class D

 {

 }

Ex 1:-

 Import pack1.A;

A is a class present in the pack1 to make available that class to the our program we have to use import session.

Ex 2:-

 Import pack1.*;

By using above statement we are importing all the classes present in the pack1 into our program. Here * represent the all the classes.

Note:-

If it is a predefined package or user defined package whenever we are using that package classes into our program we must make available that package into our program with the help of import statement.

Public:-

- This is the modifier applicable for classes, methods and variables (only for instance and static variables but not for local variables).
- If a class is declared with public modifier then we can access that class from anywhere (within the package and outside of the package).
- If we declare a member(variable) as a public then we can access that member from anywhere but Corresponding class should be visible i.e., before checking member visibility we have to check class visibility.

Ex:-

```
public class Test          // public class can access anywhere
{
    public int a=10; //public variable can access any where
    public void m1()      //public method can access any where
    {
        System.out.println("public method access in any package");
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1();
        System.out.println(t.a);
```

```
    }
}
```

Default:-

- This is the modifier applicable for classes, methods and variables (only for instance and static variables but not for local variables).
- If a class is declared with <default> modifier then we can access that class only within that current package but not from outside of the package.
- Default access also known as a package level access.
- The default modifier in the java language is **default**.

Ex:-

```
class Test
{
    void m1()
    {
        System.out.println("m1-method");
    }
    void m2()
    {
        System.out.println("m2-method");
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1();
        t.m2();
    }
}
```

Note :-

in the above program we are not providing any modifier for the methods and classes at that situation the default modifier is available for methods and classes that is default modifier. Hence we can access that methods and class with in the package.

Private:-

- private is a modifier applicable for methods and variables.
- If a member declared as private then we can access that member only from within the current class.
- If a method declare as a private we can access that method only within the class. it is not possible to call even in the child classes also.

```
class Test
{
    private void m1()
    {
        System.out.println("we can access this method only with in this class");
    }
    public static void main(String[] args)
    {
        Test t=new Test();
    }
}
```

```
        t.m1();
    }
}
```

Protected :-

- If a member declared as protected then we can access that member with in the current package anywhere but outside package only in child classes.
- But from outside package we can access protected members only by using child reference. If we try to use parent reference we will get compile time error.
- Members can be accessed only from instance area directly i.e., from static area we can't access instance members directly otherwise we will get compile time error.

Ex:-demonstrate the user defined packages and user defined imports.

krishna project source file:-

```
package com.dss;
public class StatesDemo
{
    public void ap()
    {
        System.out.println("ANDHRA PRADESH");
    }
    public void tl()
    {
        System.out.println("TELENGANA");
    }
    public void tn()
    {
        System.out.println("TAMILNADU");
    }
}
```

Tcs project source file:-

```
package com.tcs;
import com.dss.StatesDemo;//or import com.dss.*;
public class StatesInfo
{
    public static void main(String[] args)
    {
        StatesDemo sd=new StatesDemo();
        sd.ap();
        sd.tl();
        sd.tn();
    }
}
```

Step 1 :- javac -d . StatesDemo.java

Step 2 :- javac -d . StatesInfo.java

Step 3 :- java com.tcs.StatesInfo

Static import:-

- 1) this concept is introduced in 1.5 version.
- 2) if we are using the static import it is possible to call static variables and static methods directly to the java programming.

Ex:-without static import

```
import java.lang.*;  
class Test  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Hello  
World!");  
    }  
}
```

```
Ex:-package com.dss;  
public class Test  
{  
    public static int a=100;  
    public static void m1()  
    {  
        System.out.println("m1 method");  
    }  
};
```

Ex :- with static import

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

```
Ex:-  
package com.tcs;  
import static com.dss.Test.*;  
class Test1  
{  
    public static void main(String[] args)  
    {  
        System.out.println(a);  
        m1();  
    }  
}
```

Source file Declaration rules:-

The source file contains the following elements

- 1) Package declaration---→optional ---- →at most one package(0 or 1)---→1st statement
 - 2) Import declaration-----→optional-----→any number of imports ----- →2nd statement
 - 3) Class declaration-----→optional----→any number of classes----- →3rd statement
 - 4) Interface declaration---→optional----→any number of interfaces ---→3rd statement
 - 5) Comments declaration→optional----→any number of comments --- →3rd statement
- a. The package must be the first statement of the source file and it is possible to declare at most one package within the source file .
 - b. The import session must be in between the package and class statement. And it is possible to declare any number of import statements within the source file.
 - c. The class session is must be after package and import statement and it is possible to declare any number of class within the source file.
 - i. It is possible to declare at most one public class.
 - ii. It is possible to declare any number of non-public classes.
 - d. The package and import statements are applicable for all the classes present in the source file.
 - e. It is possible to declare comments at beginning and ending of any line of declaration it is possible to declare any number of comments within the source file.

Preparation of userdefined API (application programming interface document):-

1. API document nothing but user guide.
2. Whenever we are buying any product the manufacturing people provides one document called user guide. By using userguide we are able to use the product properly.
3. James gosling is developed java product whenever james gosling is deliverd the project that person is providing one user document called API(application programming interface) document it contains the information about hoe to use the product.
4. To prepare userdefined api document for the userdefined projects we must provide the description by using documentation comments that information is visible in API document.
5. If we want to create api document for your source file at that situation your source file must contains all members(classes,methods,variables....) with modifiers.

```
package com.dss;  
/*  
 *parent class used to get parent values  
 */  
public class Test extends Object  
{  
    /** by using this method we are able get some boy*/  
    public void marry(String ch)  
    {  
        System.out.println("xxx boy");  
    }  
}
```

The screenshot shows a Java API documentation page for the class `Test`. The browser title is "Generated Documentation (Untitled) - Windows Internet Explorer". The URL in the address bar is "D:\index.html". The page has a standard header with links for Package, Class, Tree, Deprecated, Index, Help, FRAMES, NO FRAMES, and All Classes.

Class Test

`java.lang.Object`
└ `com.dss.Test`

`public class Test
extends java.lang.Object`

Constructor Summary

`Test()`

Method Summary

`void marry(java.lang.String ch)`
by using this method we are able get some boy

Methods inherited from class `java.lang.Object`

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Test

`public Test()`

Multi Threading

Introduction

- 1) In the earlier days, the computer's memory is occupied by only one program. After completion of one program, it is possible to execute another program. It is called **uniprogramming**.
- 2) Whenever one program execution is completed then only second program execution will be started. Such type of execution is called cooperative execution, this execution we are having lot of disadvantages.
 - a) Most of the times **memory will be wasted**.
 - b) **CPU utilization** will be reduced because only program allow executing at a time.
 - c) The **program queue** is developed on the basis cooperative execution

To overcome above problem, a new programming style was introduced and is called as multiprogramming.

Multiprogramming means executing the more than one program at a time.

- 1) All these programs are controlled by the CPU scheduler.
- 2) **CPU scheduler** will allocate a particular time period for each and every program.
- 3) Executing several programs simultaneously is called multiprogramming.
- 4) Multiprogramming mainly focuses on the number of programs.
- 5) In multiprogramming a program can be entered in different states.
 - a. Ready state.
 - b. Running state.
 - c. Waiting state.

Advantages of multiprogramming:-

1. The main advantage of multithreading is to provide simultaneous execution of two or more parts of a application to improve the CPU utilization.
2. **CPU utilization** will be increased.
3. Execution speed will be increased and response time will be decreased.
4. **CPU resources** are not wasted.

Thread:-

1. Thread is nothing but separate path of sequential execution.
2. The **independent** execution technical name is called thread.
3. Whenever different parts of the program executed simultaneously that each and every part is called thread.
4. The thread is **light weight process** because whenever we are creating thread it is not occupying the separate memory it uses the **same memory**. Whenever the memory is shared means it is not consuming more memory.

Executing more than one thread a time is called **multithreading**.

Multitasking: Executing several tasks simultaneously is the concept of multitasking. There are two types of multitasking.

1. **Process based multitasking.**
2. **Thread based multitasking.**



Process based multitasking:

Executing several tasks simultaneously where each task is a **separate independent process** such type of multitasking is called process based multitasking.

Example:

- While **typing a java program** in the editor we can able to **listen mp3 audio songs** at the same time we can **download a file from the net** all these tasks are independent of each other and executing simultaneously and hence it is Process based multitasking.
- This type of multitasking is best suitable at "os level".

Thread based multitasking:

Executing several tasks simultaneously where each task is a separate **independent part of the same program**, is called Thread based multitasking.

And each independent part is called a "Thread".

1. This type of multitasking is best suitable for "programmatic level".
2. When compared with "C++", developing multithreading examples is very easy in java because java provides in built support for multithreading through a rich API (**Thread, Runnable, ThreadGroup**, ..etc).
3. In multithreading on 10% of the work the programmer is required to do and 90% of the work will be down by java API.
4. Whether it is process based or Thread based the main objective of multitasking is to improve performance of the system by reducing **response time**.

The main important application areas of multithreading are:

1. To implement multimedia graphics.
2. To develop animations.
3. To develop video games etc.
4. To develop web and application servers.

The ways to define instantiate and start a new Thread:

We can define a Thread in the following 2 ways.

1. By extending Thread class.
2. By implementing Runnable interface.

Defining a Thread by extending "Thread class":

Example:

```
class MyThread extends Thread
{
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            System.out.println("child Thread");
        }
    }
}

defining a Thread.
Job of a Thread.
```

```
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread(); //Instantiation of a Thread
        t.start(); //starting of a Thread

        for(int i=0;i<5;i++)
        {
            System.out.println("main thread");
        }
    }
}
```

Case 1: Thread Scheduler:

- If multiple Threads are waiting to execute then which Thread will execute 1st is decided by "Thread Scheduler" which is part of JVM.

- Which algorithm or behavior followed by Thread Scheduler we can't expect exactly it is the JVM vendor dependent hence in multithreading examples we can't expect exact execution order and exact output.
 - The following are various possible outputs for the above program.

Case 2: Difference between `t.start()` and `t.run()` methods.

- In the case of `t.start()` a new Thread will be created which is responsible for the execution of `run()` method.
 - But in the case of `t.run()` no new Thread will be created and `run()` method will be executed just like a normal method by the main Thread.
 - In the above program if we are replacing `t.start()` with `t.run()` the following is the output.

```
Output:  
child thread  
main thread  
main thread  
main thread  
main thread  
main thread
```

Entire output produced by only main Thread.

Case 3: importance of Thread class start() method.

For every Thread the required mandatory activities like registering the Thread with Thread Scheduler will takes care by Thread class start() method and programmer is responsible just to define the job of the Thread inside run() method.

That is start() method acts as best assistant to the programmer.

Example:

```
start()
{
    1. Register Thread with Thread Scheduler
    2. All other mandatory low level activities.
    3. Invoke or calling run() method.
}
```

We can conclude that without executing Thread class start() method there is no chance of starting a new Thread in java. Due to this start() is considered as **heart of multithreading**.

Case 4: If we are not overriding run() method:

If we are not overriding run() method then Thread class run() method will be executed which has empty implementation and hence we won't get any output.

Example:

```
class MyThread extends Thread
{}
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
    }
}
```

It is highly recommended to override run() method. Otherwise don't go for multithreading concept.

Case 5: Overloading of run() method.

We can overload run() method but Thread class start() method always invokes no argument run() method the other overload run() methods we have to call explicitly then only it will be executed just like normal method.

Example:

```
class MyThread extends Thread
{
    public void run()
    {
        System.out.println("no arg method");
    }
    public void run(int i)
    {
        System.out.println("int arg method");
    }
}
```

```

}
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
    }
}
Output:
No arg method

```

Case 6: overriding of start() method:

If we override start() method then our start() method will be executed just like a normal method call and no new Thread will be started.

Example:

```

class MyThread extends Thread
{
    public void start()
    {
        System.out.println("start method");
    }
    public void run()
    {
        System.out.println("run method");
    }
}
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
        System.out.println("main method");
    }
}
Output:
start method
main method

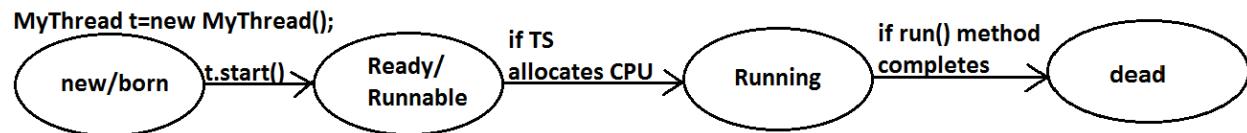
```

Entire output produced by only main Thread.

Note : It is never recommended to override start() method.

Case 7: life cycle of the Thread:

Diagram:

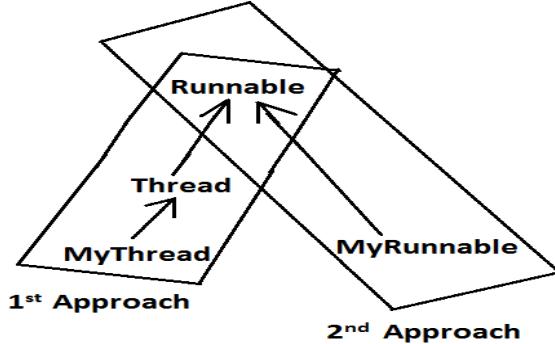


- Once we created a Thread object then the Thread is said to be in new state or born state.
- Once we call start() method then the Thread will be entered into Ready or Runnable state.
- If Thread Scheduler allocates CPU then the Thread will be entered into running state.
- Once run() method completes then the Thread will enter into dead state.

Defining a Thread by implementing Runnable interface:

We can define a Thread even by implementing Runnable interface also.
 Runnable interface present in java.lang.pkg and contains only one method run().

Diagram:



Example:

defining a Thread

```

class MyRunnable implements Runnable
{
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            System.out.println("child Thread");
        }
    }
}
  
```

job of a Thread

```

class ThreadDemo
{
    public static void main(String[] args)
    {
        MyRunnable r=new MyRunnable();
        Thread t=new Thread(r); //here r is a Target Runnable
        t.start();

        for(int i=0;i<10;i++)
        {
            System.out.println("main thread");
        }
    }
}
  
```

```

Output:
main thread
child Thread

```

We can't expect exact output but there are several possible outputs.

Case study:

```

MyRunnable r=new MyRunnable();
Thread t1=new Thread();
Thread t2=new Thread(r);

```

Case 1: t1.start():

A new Thread will be created which is responsible for the execution of Thread class run() method.

Output:

```

main thread
main thread
main thread
main thread
main thread

```

Case 2: t1.run():

No new Thread will be created but Thread class run() method will be executed just like a normal method call.

Output:

```

main thread
main thread
main thread

```

main thread
main thread

Case 3: t2.start():

New Thread will be created which is responsible for the execution of MyRunnable run() method.

Output:

main thread
main thread
main thread
main thread
main thread
child Thread
child Thread
child Thread
child Thread
child Thread

Case 4: t2.run():

No new Thread will be created and MyRunnable run() method will be executed just like a normal method call.

Output:

child Thread
child Thread
child Thread
child Thread
child Thread
main thread
main thread
main thread
main thread
main thread

Case 5: r.start():

We will get compile time error saying start()method is not available in MyRunnable class.

Output:

Compile time error
E:\SCJP>javac ThreadDemo.java
ThreadDemo.java:18: cannot find symbol
Symbol: method start()
Location: class MyRunnable

Case 6: r.run():

No new Thread will be created and MyRunnable class run() method will be executed just like a normal method call.

Output:

```
child Thread
child Thread
child Thread
child Thread
child Thread
main thread
main thread
main thread
main thread
main thread
```

In which of the above cases a new Thread will be created which is responsible for the execution of MyRunnable run() method ?

t2.start();

In which of the above cases a new Thread will be created ?

t1.start();
t2.start();

In which of the above cases MyRunnable class run() will be executed ?

t2.start();
t2.run();
r.run();

Best approach to define a Thread:

- Among the 2 ways of defining a Thread, implements Runnable approach is always recommended.
- In the 1st approach our class should always extends Thread class there is no chance of extending any other class hence we are missing the benefits of inheritance.
- But in the 2nd approach while implementing Runnable interface we can extend some other class also. Hence implements Runnable mechanism is recommended to define a Thread.

Thread class constructors:

1. Thread t=new Thread();
2. Thread t=new Thread(Runnable r);
3. Thread t=new Thread(String name);
4. Thread t=new Thread(Runnable r,String name);
5. Thread t=new Thread(ThreadGroup g,String name);

6. Thread t=new Thread(ThreadGroup g,Runnable r);
7. Thread t=new Thread(ThreadGroup g,Runnable r,String name);
8. Thread t=new Thread(ThreadGroup g,Runnable r,String name,long stackSize);

Thread Priorities

- Every Thread in java has some priority it may be default priority generated by JVM (or) explicitly provided by the programmer.
- The valid range of Thread priorities is 1 to 10[but not 0 to 10] where 1 is the least priority and 10 is highest priority.
- Thread class defines the following constants to represent some standard priorities.
 1. Thread.MIN_PRIORITY-----1
 2. Thread.MAX_PRIORITY-----10
 3. Thread.NORM_PRIORITY-----5
- There are no constants like Thread.LOW_PRIORITY, Thread.HIGH_PRIORITY
- Thread scheduler uses these priorities while allocating CPU.
- The Thread which is having highest priority will get chance for 1st execution.
- If 2 Threads having the same priority then we can't expect exact execution order it depends on Thread scheduler whose behavior is vendor dependent.
- We can get and set the priority of a Thread by using the following methods.
 1. public final int getPriority()
 2. public final void setPriority(int newPriority); //the allowed values are 1 to 10
- The allowed values are 1 to 10 otherwise we will get runtime exception saying "IllegalArgumentException".

Default priority:

The default priority only for the main Thread is 5. But for all the remaining Threads the default priority will be inheriting from parent to child. That is whatever the priority parent has by default the same priority will be for the child also.

Example 1:

```
class MyThread extends Thread
{}
class ThreadPriorityDemo
{
    public static void main(String[] args)
    {
        System.out.println(Thread.currentThread().getPriority()); //5
        Thread.currentThread().setPriority(9);
        MyThread t=new MyThread();
        System.out.println(t.getPriority()); //9
    }
}
```

Example 2:

```
class MyThread extends Thread
{
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            System.out.println("child thread");
        }
    }
}
class ThreadPriorityDemo
```

```
{  
    public static void main(String[] args)  
    {  
        MyThread t=new MyThread();  
        //t.setPriority(10);      //----> 1  
        t.start();  
        for(int i=0;i<10;i++)  
        {  
            System.out.println("main thread");  
        }  
    }  
}
```

- If we are commenting line 1 then both main and child Threads will have the same priority and hence we can't expect exact execution order.
- If we are not commenting line 1 then child Thread has the priority 10 and main Thread has the priority 5 hence child Thread will get chance for execution and after completing child Thread main Thread will get the chance in this the output is:

Output:

```
child thread  
main thread
```

Some operating systems (like windowsXP) may not provide proper support for Thread priorities. We have to install separate bats provided by vendor to provide support for priorities.

The Methods to Prevent a Thread from Execution:

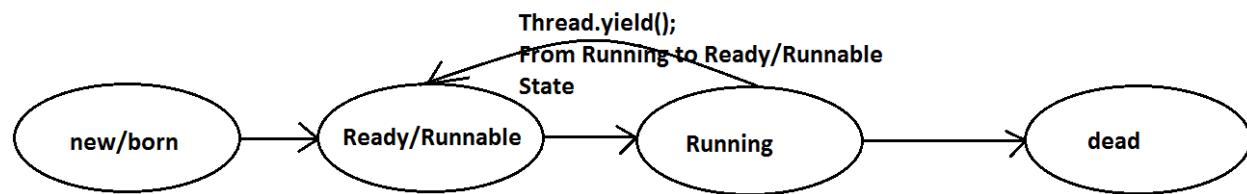
We can prevent(stop) a Thread execution by using the following methods.

1. yield();
2. join();
3. sleep();

yield():

1. yield() method causes "to pause current executing Thread for giving the chance of remaining waiting Threads of same priority".
2. If all waiting Threads have the low priority or if there is no waiting Threads then the same Thread will be continued its execution.
3. If several waiting Threads with same priority available then we can't expect exact which Thread will get chance for execution.
4. The Thread which is yielded when it get chance once again for execution is depends on mercy of the Thread scheduler.
5. public static native void yield();

Diagram:



Example:

```

class MyThread extends Thread
{
    public void run()
    {
        for(int i=0;i<5;i++)
        {
            Thread.yield();
            System.out.println("child thread");
        }
    }
}
class ThreadYieldDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
    }
}
  
```

```

        t.start();
        for(int i=0;i<5;i++)
        {
            System.out.println("main thread");
        }
    }
Output:
main thread
main thread
main thread
main thread
main thread
child thread
child thread
child thread
child thread
child thread

```

In the above program child Thread always calling yield() method and hence main Thread will get the chance more number of times for execution.

Hence the chance of completing the main Thread first is high.

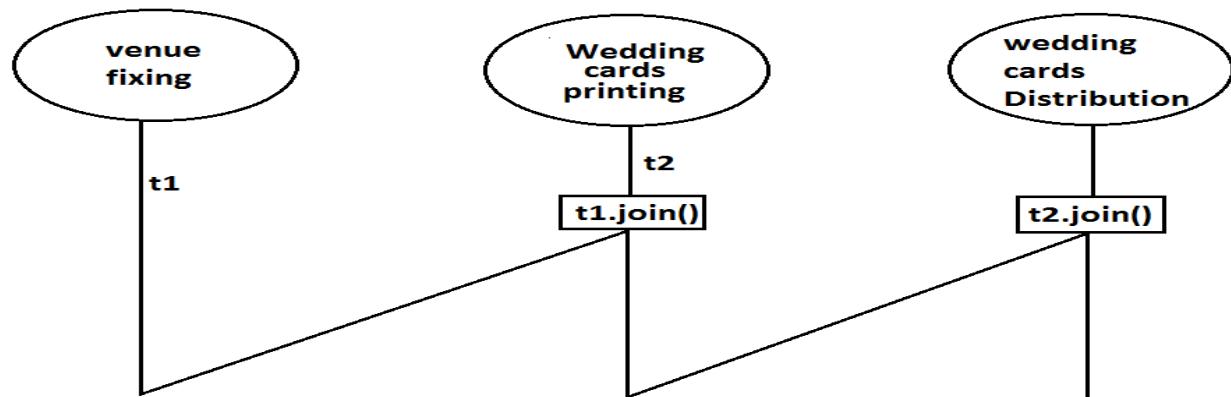
Note : Some operating systems may not provide proper support for yield() method.

Join():

If a Thread wants to wait until completing some other Thread then we should go for join() method.

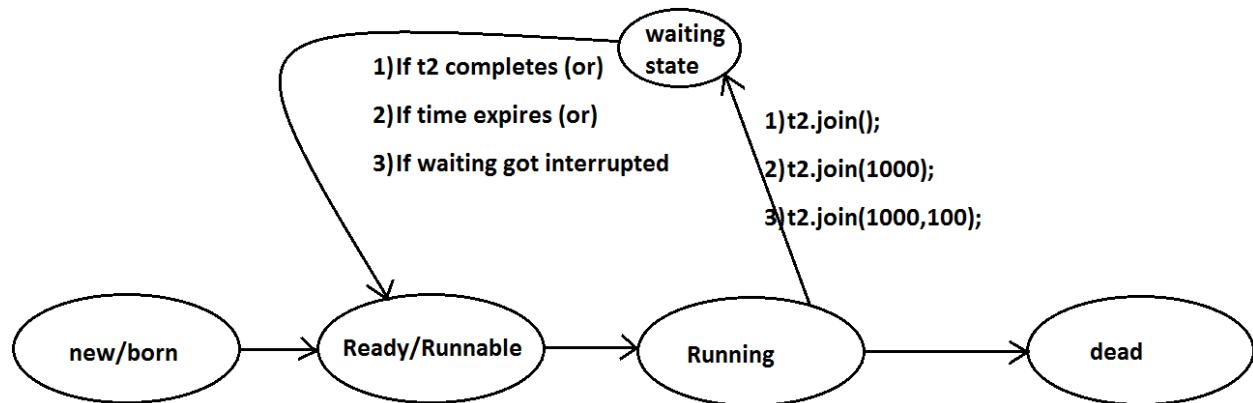
Example: If a Thread t1 executes t2.join() then t1 should go for waiting state until completing t2.

Diagram:



1. public final void join() throws InterruptedException
2. public final void join(long ms) throws InterruptedException
3. public final void join(long ms,int ns) throws InterruptedException

Diagram:



Every join() method throws InterruptedException, which is checked exception hence compulsory we should handle either by **try catch** or by **throws** keyword.

Otherwise we will get compiletime error.

Example:

```

class MyThread extends Thread
{
    public void run()
    {
        for(int i=0;i<5;i++)
        {
            System.out.println("Sita Thread");
            try
            {
                Thread.sleep(2000);
            }
            catch (InterruptedException e){}
        }
    }
}
class ThreadJoinDemo
{
    public static void main(String[] args) throws
InterruptedException
    {
        MyThread t=new MyThread();
        t.start();
        //t.join();    //--->1
        for(int i=0;i<5;i++)
        {
    
```

```
        System.out.println("Rama Thread");  
    }  
}  
}
```

- If we are commenting line 1 then both Threads will be executed simultaneously and we can't expect exact execution order.
 - If we are not commenting line 1 then main Thread will wait until completing child Thread in this the output is sita Thread 5 times followed by Rama Thread 5 times.

Waiting of child Thread until completing main Thread :

```
Example:  
class MyThread extends Thread  
{  
    static Thread mt;  
    public void run()  
    {  
        try  
        {  
            mt.join();  
        }  
        catch (InterruptedException e){ }  
  
        for(int i=0;i<5;i++)  
        {  
            System.out.println("Child Thread");  
        }  
    }  
}  
class ThreadJoinDemo  
{  
    public static void main(String[] args) throws  
InterruptedException  
    {  
        MyThread mt=Thread.currentThread();  
        MyThread t=new MyThread();  
        t.start();  
  
        for(int i=0;i<5;i++)  
        {  
            Thread.sleep(2000);  
            System.out.println("Main Thread");  
        }  
    }  
}
```

```
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Child Thread
Child Thread
Child Thread
Child Thread
Child Thread
```

Note :

If main thread calls join() on child thread object and child thread called join() on main thread object then both threads will wait for each other forever and the program will be hanged(like deadlock if a Thread class join() method on the same thread itself then the program will be hanged).

Example :

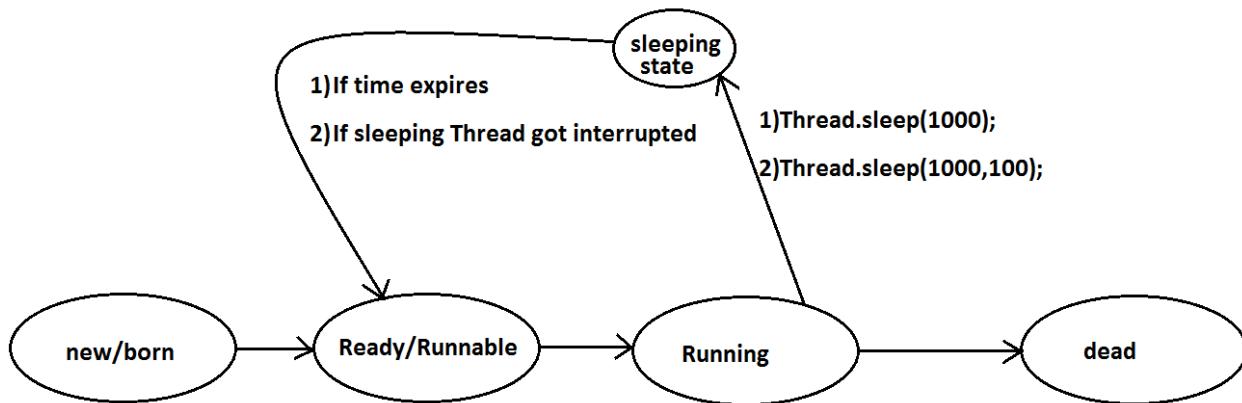
```
class ThreadDemo
{
    public static void main() throws InterruptedException
    {
        Thread.currentThread().join();
        -----
        main           main
    }
}
```

Sleep() method:

If a Thread don't want to perform any operation for a particular amount of time then we should go for sleep() method.

1. **public static native void sleep(long ms) throws InterruptedException**
2. **public static void sleep(long ms,int ns) throws InterruptedException**

Diagram:



Example:

```

class ThreadJoinDemo
{
    public static void main(String[] args) throws
InterruptedException
    {
        System.out.println("M");
        Thread.sleep(3000);
        System.out.println("E");
        Thread.sleep(3000);
        System.out.println("G");
        Thread.sleep(3000);
        System.out.println("A");
    }
}
  
```

Output:

M
E
G
A

Interrupting a Thread:

How a Thread can interrupt another thread ?

If a Thread can interrupt a sleeping or waiting Thread by using `interrupt()`(break off) method of `Thread` class.

`public void interrupt();`

Example:

```

class MyThread extends Thread
{
    public void run()
    {
        try
        {
  
```

```

        for(int i=0;i<5;i++)
        {
            System.out.println("i am lazy Thread :"+i);
            Thread.sleep(2000);
        }
    } catch (InterruptedException e)
    {
        System.out.println("i got interrupted");
    }
}
class ThreadInterruptDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
        //t.interrupt();      //--->1
        System.out.println("end of main thread");
    }
}

```

- If we are commenting line 1 then main Thread won't interrupt child Thread and hence child Thread will be continued until its completion.
- If we are not commenting line 1 then main Thread interrupts child Thread and hence child Thread won't continued until its completion in this case the output is:

End of main thread
I am lazy Thread: 0
I got interrupted

Note:

- Whenever we are calling interrupt() method we may not see the effect immediately, if the target Thread is in sleeping or waiting state it will be interrupted immediately.
- If the target Thread is not in sleeping or waiting state then interrupt call will wait until target Thread will enter into sleeping or waiting state. Once target Thread entered into sleeping or waiting state it will effect immediately.
- In its lifetime if the target Thread never entered into sleeping or waiting state then there is no impact of interrupt call simply interrupt call will be wasted.

Example:

```

class MyThread extends Thread
{
    public void run()
    {
        for(int i=0;i<5;i++)
        {
            System.out.println("iam lazy thread");
        }
    }
}

```

```

        System.out.println("I'm entered into sleeping stage");
    try
    {
        Thread.sleep(3000);
    }
    catch (InterruptedException e)
    {
        System.out.println("i got interrupted");
    }
}
class ThreadInterruptDemo1
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
        t.interrupt();
        System.out.println("end of main thread");
    }
}

```

- In the above program interrupt() method call invoked by main Thread will wait until child Thread entered into sleeping state.
- Once child Thread entered into sleeping state then it will be interrupted immediately.

Comparison of yield, join and sleep() method?

property	Yield()	Join()	Sleep()
1) Purpose?	To pause current executing Thread for giving the chance of remaining waiting Threads of same priority.	If a Thread wants to wait until completing some other Thread then we should go for join.	If a Thread don't want to perform any operation for a particular amount of time then we should go for sleep() method.
2) Is it static?	yes	no	yes
3) Is it final?	no	yes	no
4) Is it overloaded?	No	yes	yes
5) Is it throws InterruptedException?	no	yes	yes
6) Is it native method?	yes	no	sleep(long ms) -->native sleep(long ms,int ns) -->non-native

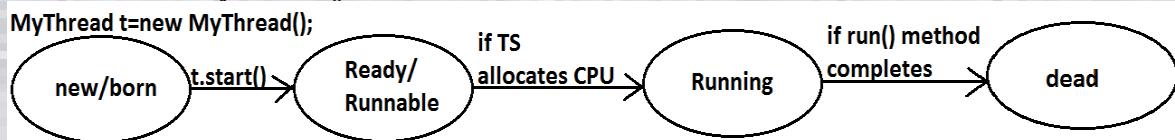
LIFECYCLE OF THREAD:

Life cycle stages are:-

- 1) New
- 2) Ready
- 3) Running state
- 4) Blocked / waiting / non-running mode
- 5) Dead state

New :-

Once we created a Thread object then the Thread is said to be in new state or born state.
MyThread t=new MyThread();



Ready :-

Once we call start() method then the Thread will be entered into Ready or Runnable state.
t.start()

Running state:-

If thread scheduler allocates CPU for particular thread. Thread goes to running state. The Thread is running state means the run() is executed.

Blocked State:-

If the running thread got interrupted or goes to sleeping state at that moment it goes to the blocked state.

Dead State:-

Once run() method completes then the Thread will entered into dead state. If the business logic of the project is completed means run() over thread goes dead state.

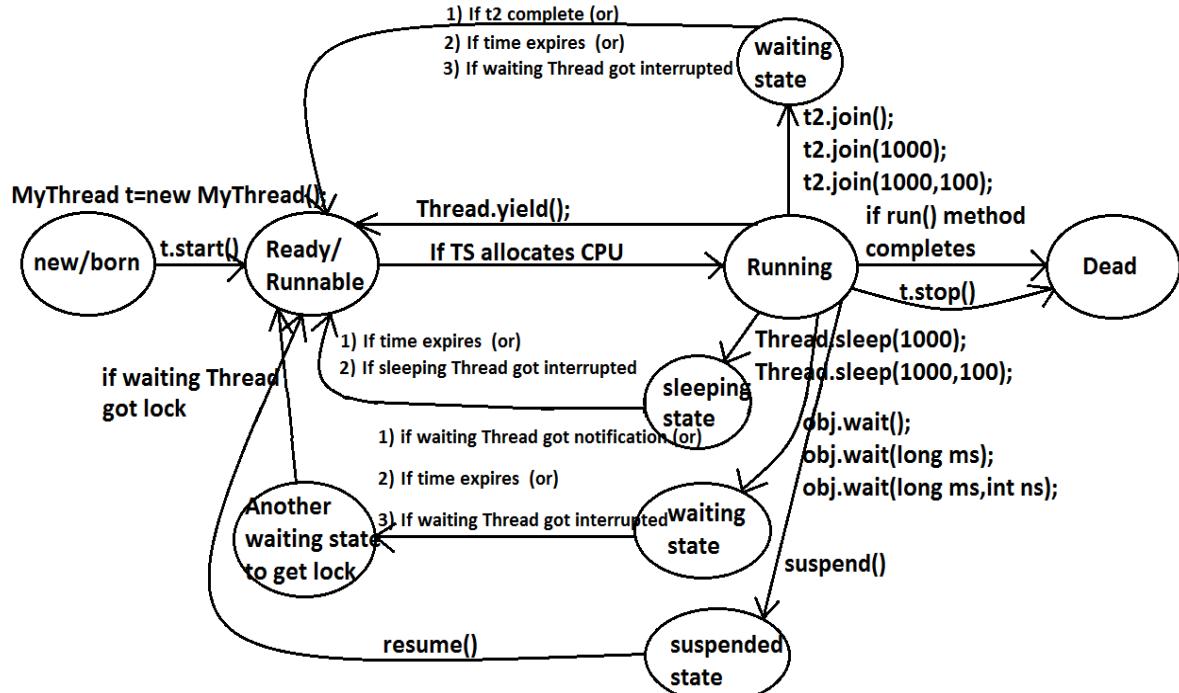


Figure: LIFE CYCLE OF A THREAD

Synchronization

1. **Synchronized** is the keyword applicable for **methods** and **blocks** but not for classes and variables.
2. If a method or block declared as the synchronized then at a time **only one Thread is allowed** to execute that method or block on the given object.
3. The main advantage of synchronized keyword is we can resolve **data inconsistency** problems.
4. But the main disadvantage of synchronized keyword is it **increases waiting time** of the Thread and effects performance of the system.
5. Hence if there is no specific requirement then never recommended to use synchronized keyword.
6. Internally synchronization concept is implemented by using **lock concept**.
7. Every java object has a lock. A lock has only one key. Most of the time lock is unlocked and nobody cares.
8. If a Thread wants to execute any synchronized method on the given object, **First it has to get the lock of that object**. Once a Thread got the lock of that object then it is allowed to execute any synchronized method on that object. If the synchronized method execution completes then automatically Thread releases lock.
9. While a Thread executing any synchronized method the remaining Threads are not allowed execute any synchronized method on that object simultaneously. But remaining Threads are allowed to execute any non-synchronized method simultaneously. [lock concept is implemented **based on object** but not based on method].

Example:

```
class Display
{
    public synchronized void wish(String name)
    {
        for(int i=0;i<5;i++)
        {
            System.out.print("good morning:");
            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {}
            System.out.println(name);
        }
    }
}
class MyThread extends Thread
{
    Display d;
    String name;
    MyThread(Display d, String name)
    {
```

```

        this.d=d;
        this.name=name;
    }
    public void run()
    {
        d.wish(name);
    }
}
class SynchronizedDemo
{
    public static void main(String[] args)
    {
        Display d1=new Display();
        MyThread t1=new MyThread(d1,"dhoni");
        MyThread t2=new MyThread(d1,"yuvaraj");
        t1.start();
        t2.start();
    }
}

```

If **we are not declaring wish() method as synchronized** then both Threads will be executed simultaneously and we will get irregular output.

Output:

```

good morning:good morning:yuvaraj
good morning:dhoni
good morning:yuvaraj
dhoni

```

If **we declare wish() method as synchronized** then the Threads will be executed one by one that is until completing the 1st Thread the 2nd Thread will wait in this case we will get regular output which is nothing but

Output:

```

good morning:dhoni
good morning:dhoni
good morning:dhoni
good morning:dhoni
good morning:dhoni
good morning:dhoni
good morning:yuvaraj
good morning:yuvaraj
good morning:yuvaraj
good morning:yuvaraj
good morning:yuvaraj

```

Example :-

```

class Test
{
    public static synchronized void x(String msg)//only one thread is able to access
    {
        try
        {
            System.out.println(msg);
            Thread.sleep(4000);
            System.out.println(msg);
            Thread.sleep(4000);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}

class MyThread1 extends Thread
{
    public void run(){Test.x("ratan");}
}

class MyThread2 extends Thread
{
    public void run(){Test.x("anu");}
}

class MyThread3 extends Thread
{
    public void run(){Test.x("banu");}
}

class TestDemo
{
    public static void main(String[] args)//main thread -1
    {
        MyThread1 t1 = new MyThread1();
        MyThread2 t2 = new MyThread2();
        MyThread3 t3 = new MyThread3();
        t1.start();//2-Threads
        t2.start();//3-Threads
        t3.start();//4-Threads
    }
}

```

If method is synchronized:

D:\DP>java ThreadDemo
 anu
 anu
 banu
 banu
 ratan
 ratan

if method is non-synchronized:-

D:\DP>java ThreadDemo
 banu

```
ratan
anu
banu
anu
ratan
```

synchronized blocks:-

If the application method contains 100 lines but if we want to synchronized only 10 lines of code use synchronized blocks.

The synchronized block contains less scope compare to method.

If we are writing all the method code inside the synchronized blocks it will work same as the synchronized method.

Syntax:-

```
synchronized(object)
{
    //code
}

class Heroin
{
    public void message(String msg)
    {
        synchronized(this){
            System.out.println("hi "+msg+" "+Thread.currentThread().getName());
            try{Thread.sleep(5000);}
            catch(InterruptedException e){e.printStackTrace();}
        }
        System.out.println("hi krishnasoft");
    }
}

class MyThread1 extends Thread
{
    Heroin h;
    MyThread1(Heroin h)
    {this.h=h;}
    public void run()
    {
        h.message("Samantha");
    }
}

class MyThread2 extends Thread
{
    Heroin h;
    MyThread2(Heroin h)
    {this.h=h;}
    public void run()
    {
        h.message("Kavi");
    }
}

class ThreadDemo
```

```
public static void main(String[] args)
{
    Heroin h = new Heroin();
    MyThread1 t1 = new MyThread1(h);
    MyThread2 t2 = new MyThread2(h);
    t1.start();
    t2.start();
}
```

Deadlock:

- If two Threads are waiting for each other forever (without end) such type of situation (**infinite waiting**) is called deadlock.
- There are no resolution techniques for dead lock but several prevention (avoidance) techniques are possible.
- **Synchronized keyword is the cause for deadlock** hence whenever we are using synchronized keyword we have to take special care.

Example:

```

class A
{
    public synchronized void tata(B b)
    {
        System.out.println("Thread1 starts execution of tata() method");
        try
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException e)
        {}
        System.out.println("Thread1 trying to call b.last()");
        b.last();
    }
    public synchronized void last()
    {
        System.out.println("inside A, this is last()method");
    }
}
class B
{
    public synchronized void toto(A a)
    {
        System.out.println("Thread2 starts execution of toto() method");
        try
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException e)
        {}
        System.out.println("Thread2 trying to call a.last()");
        a.last();
    }
    public synchronized void last()
    {
        System.out.println("inside B, this is last() method");
    }
}
class DeadLock implements Runnable
{
    A a=new A();
    B b=new B();
    DeadLock()
    {
        Thread t=new Thread(this);
        t.start();
        a.tata(b); //main thread
    }
    public void run()
    {

```

```
        b.toto(a); //child thread
    }
    public static void main(String[] args)
    {
        new DeadLock(); //main thread
    }
}
Output:
Thread1 starts execution of tata() method
Thread2 starts execution of toto() method
Thread2 trying to call a.last()
Thread1 trying to call b.last()
//here cursor always waiting.
```

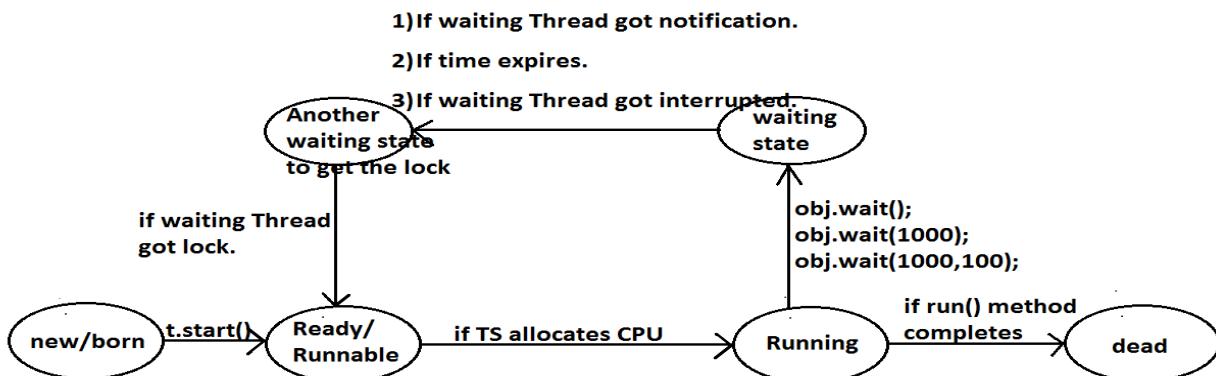
Note : If we remove atleast one synchronized keyword then we won't get DeadLock. Hence synchronized keyword is the only reason for DeadLock due to this while using synchronized keyword we have to handle carefully.

Inter Thread communication (wait(),notify(), notifyAll()):

- Two Threads can communicate with each other by using **wait()**, **notify()** and **notifyAll()** methods.
- wait()**, **notify()** and **notifyAll()** methods are available in **Object class** but not in Thread class because Thread can call these methods on any common object.
- To call **wait()**, **notify()** and **notifyAll()** methods compulsory the current Thread should be owner of that object
i.e., current Thread should have lock of that object
i.e., current Thread should be in synchronized area. Hence we can call **wait()**, **notify()** and **notifyAll()** methods only from synchronized area otherwise we will get runtime exception saying **IllegalMonitorStateException**.
- Once a Thread calls **wait()** method on the given object, First it releases the lock of that object **immediately** and entered into waiting state.
- Once a Thread calls **notify()** (or) **notifyAll()** methods it releases the lock of that object but **may not immediately**.
- Except these (**wait()**,**notify()**,**notifyAll()**) methods there is no other place(method) where the lock release will happen.

Method	Is Thread Releases Lock?
yield()	No
join()	No
sleep()	No
wait()	Yes
notify()	Yes
notifyAll()	Yes

- Once a Thread calls **wait()**, **notify()**, **notifyAll()** methods on any object then it releases the lock of that particular object **but not all locks it has**.
 - public final void **wait()** throws **InterruptedException**
 - public final native void **wait(long ms)** throws **InterruptedException**
 - public final void **wait(long ms,int ns)** throws **InterruptedException**
 - public final native void **notify()**
 - public final void **notifyAll()**



```

Example 1:
class ThreadA
{
    public static void main(String[] args) throws InterruptedException
    {
        ThreadB b=new ThreadB();
        b.start();
        synchronized(b)
        {
            System.out.println("main Thread calling wait() method");//step-1
            b.wait();
            System.out.println("main Thread got notification call");//step-4
            System.out.println(b.total);
        }
    }
}
class ThreadB extends Thread
{
    int total=0;
    public void run()
    {
        synchronized(this)
        {
            System.out.println("child thread starts calcuation");//step-2
            for(int i=0;i<=100;i++)
            {
                total=total+i;
            }
            System.out.println("child thread giving notification call");//step-3
            this.notify();
        }
    }
}
Output:
main Thread calling wait() method
child thread starts calculation
child thread giving notification call
main Thread got notification call
5050

```

Notify vs notifyAll():

- We can use notify() method to give notification **for only one Thread**. If multiple Threads are waiting then only one Thread will get the chance and remaining Threads has to wait for further notification. But which Thread will be notify(inform) we can't expect exactly it depends on JVM.
- We can use notifyAll() method to give the notification **for all waiting Threads**. All waiting Threads will be notified and will be executed one by one, because they are required lock

Note: On which object we are calling wait(), notify() and notifyAll() methods that corresponding object lock we have to get but not other object locks.

Which of the following statements are True ?

1. Once a Thread calls wait() on any Object immediately it will entered into waiting state **without releasing the lock** ?
NO
2. Once a Thread calls wait() on any Object it reduces the lock of that Object but **may not immediately** ?
NO
3. Once a Thread calls wait() on any Object it **immediately releases all locks** whatever it has and entered into waiting state ?
NO
4. Once a Thread calls wait() on any Object it **immediately releases** the lock of that particular Object and entered into waiting state ?
YES
5. Once a Thread calls notify() on any Object it **immediately releases** the lock of that Object ?
NO
6. Once a Thread calls notify() on any Object it releases the lock of that Object but **may not immediately** ?
YES

Daemon Threads:

The Threads which are **executing in the background** are called daemon Threads. The main objective of daemon Threads is to provide support for non-daemon Threads like **main Thread**.

Example:

Garbage collector

When ever the program runs with low memory, the JVM will execute Garbage Collector to provide **free memory**. So that the main Thread can continue it's execution.

- We can check whether the Thread is daemon or not by using **isDaemon() method** of Thread class.
public final boolean isDaemon();
- We can change daemon nature of a Thread by using **setDaemon()** method.
public final void setDaemon(boolean b);
- But we can change daemon nature **before starting Thread only**. That is after starting the Thread if we are trying to change the daemon nature we will get R.E saying **IllegalThreadStateException**.
- **Default Nature** : Main Thread is always non daemon and we can't change its daemon nature because **it's already started at the beginning only**.
- Main Thread is always non daemon and for the remaining Threads **daemon nature will be inheriting from parent to child** that is if the parent is daemon, child is also daemon and if the parent is non daemon then child is also non daemon.
- Whenever the last non daemon Thread terminates automatically all daemon Threads will be terminated.

Example:

```
class MyThread extends Thread
{
}
class DaemonThreadDemo
{
    public static void main(String[] args)
    {
        System.out.println(Thread.currentThread().isDaemon());
        MyThread t=new MyThread();
        System.out.println(t.isDaemon());           1
        t.start();
        t.setDaemon(true);
        System.out.println(t.isDaemon());
    }
}
Output:
false
false
RE:IllegalThreadStateException
```

```

Example:
class MyThread extends Thread
{
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            System.out.println("lazy thread");
            try
            {
                Thread.sleep(2000);
            }
            catch (InterruptedException e)
            {}
        }
    }
}
class DaemonThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.setDaemon(true); //-->1
        t.start();
        System.out.println("end of main Thread");
    }
}
Output:
End of main Thread

```

Deadlock vs Starvation:

- A long waiting of a Thread which never ends is called **deadlock**.
- A long waiting of a Thread which ends at certain point is called **starvation**.
- A low priority Thread has to wait until completing all high priority Threads.
- This long waiting of Thread which ends at certain point is called **starvation**.

How to kill a Thread in the middle of the line?

- We can call **stop()** method to stop a Thread in the middle then it will be entered into dead state immediately.
`public final void stop();`
- **stop()** method has been deprecated and hence not recommended to use.

suspend and resume methods:

- A Thread can suspend another Thread by using **suspend()** method then that Thread will be **paused temporarily**.
- A Thread can resume a suspended Thread by using **resume()** method then suspended Thread will **continue its execution**.
 1. `public final void suspend();`

2. **public final void resume();**
- Both methods are deprecated and not recommended to use.

RACE condition:

Executing multiple Threads simultaneously and causing data inconsistency problems is nothing but **Race condition**

we can resolve race condition by using synchronized keyword.

ThreadGroup in Java

Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call.

Note: Now **suspend()**, **resume()** and **stop()** methods are deprecated.

Java thread group is implemented by *java.lang.ThreadGroup* class.

A ThreadGroup represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.

A thread is allowed to access information about its own thread group, but it cannot access the information about its thread group's parent thread group or any other thread groups.

Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

No.	Constructor	Description
1)	ThreadGroup(String name)	creates a thread group with given name.
2)	ThreadGroup(ThreadGroup parent, String name)	creates a thread group with given parent group and name.

Methods of ThreadGroup class

There are many methods in ThreadGroup class. A list of ThreadGroup methods are given below.

S.N.	Modifier and Type	Method	Description
1)	void	checkAccess()	This method determines if the currently running thread has permission to modify the thread group.
2)	int	activeCount()	This method returns an estimate of the number of active threads in the thread group and its subgroups.
3)	int	activeGroupCount()	This method returns an estimate of the number of active groups in the thread group and its subgroups.
4)	void	destroy()	This method destroys the thread group and all of its subgroups.
5)	int	enumerate(Thread[] list)	This method copies into the specified array every active thread in the thread group and its subgroups.
6)	int	getMaxPriority()	This method returns the maximum priority of

		the thread group.
7)	String getName()	This method returns the name of the thread group.
8)	ThreadGr oup getParent()	This method returns the parent of the thread group.
9)	void interrupt()	This method interrupts all threads in the thread group.
10)	boolean isDaemon()	This method tests if the thread group is a daemon thread group.
11)	void setDaemon(boolean daem on)	This method changes the daemon status of the thread group.
12)	boolean isDestroyed()	This method tests if this thread group has been destroyed.
13)	void list()	This method prints information about the thread group to the standard output.
14)	boolean parentOf(ThreadGroup g	This method tests if the thread group is either the thread group argument or one of its ancestor thread groups.
15)	void suspend()	This method is used to suspend all threads in the thread group.
16)	void resume()	This method is used to resume all threads in the thread group which was suspended using suspend() method.
17)	void setMaxPriority(int pri)	This method sets the maximum priority of the group.
18)	void stop()	This method is used to stop all threads in the thread group.
19)	String toString()	This method returns a string representation of the Thread group.

Let's see a code to group multiple threads.

1. ThreadGroup tg1 = new ThreadGroup("Group A");
2. Thread t1 = new Thread(tg1,new MyRunnable(),"one");
3. Thread t2 = new Thread(tg1,new MyRunnable(),"two");
4. Thread t3 = new Thread(tg1,new MyRunnable(),"three");

Now all 3 threads belong to one group. Here, tg1 is the thread group name, MyRunnable is the class that implements Runnable interface and "one", "two" and "three" are the thread names.

Now we can interrupt all threads by a single line of code only.

1. Thread.currentThread().getThreadGroup().interrupt();

ThreadGroup Example

File: ThreadGroupDemo.java

```
public class ThreadGroupDemo implements Runnable{
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
    public static void main(String[] args) {
        ThreadGroupDemo runnable = new ThreadGroupDemo();
        ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");

        Thread t1 = new Thread(tg1, runnable, "one");
        t1.start();
        Thread t2 = new Thread(tg1, runnable, "two");
        t2.start();
        Thread t3 = new Thread(tg1, runnable, "three");
        t3.start();

        System.out.println("Thread Group Name: "+tg1.getName());
        tg1.list();
    }
}
```

Output:

```
one
two
three
Thread Group Name: Parent ThreadGroup
java.lang.ThreadGroup[name=Parent ThreadGroup,maxpri=10]
    Thread[one,5,Parent ThreadGroup]
    Thread[two,5,Parent ThreadGroup]
    Thread[three,5,Parent ThreadGroup]
```

CHAPTER-6

EVENT HANDLING

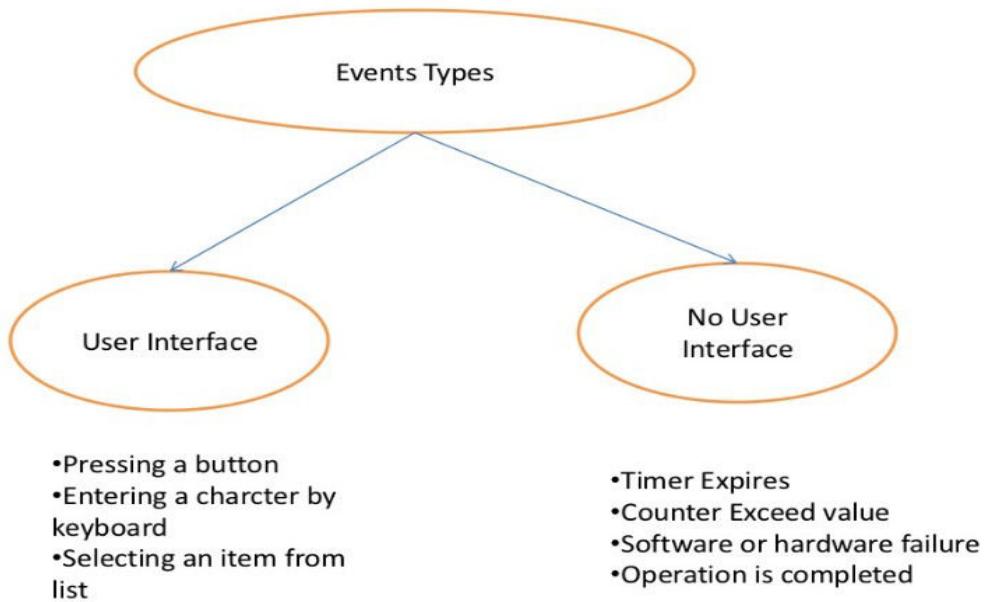
STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

java.awt.BorderLayout	javax.swing.ButtonGroup
CENTER	add
EAST	javax.swing.JCheckBox
NORTH	javax.swing.JComboBox
SOUTH	addItem
WEST	getSelectedItem
java.awt.Component	isEditable
addKeyListener	setEditable
addMouseListener	setSelectedItem
setFocusable	javax.swing.JComponent
java.awt.Container	setBorder
setLayout	setFocusable
java.awt.FlowLayout	setFont
java.awt.Font	javax.swing.JFrame
BOLD	setMenuBar
ITALIC	javax.swing.JMenu
java.awt.GridLayout	add
java.awt.event.KeyEvent	javax.swing.JMenuBar
java.awt.event.KeyListener	add
keyPressed	javax.swing.JMenuItem
keyReleased	javax.swing.JRadioButton
keyTyped	javax.swing.JSlider
java.awt.event.MouseEvent	addChangeListener
getX	getValue
getY	javax.swing.KeyStroke
java.awt.event.MouseListener	getKeyStrokeForEvent
mouseClicked	javax.swing.Timer
mouseEntered	start
mouseExited	stop
mousePressed	javax.swing.border.EtchedBorder
mouseReleased	javax.swing.border.TitledBorder
javax.swing.AbstractButton	javax.swing.event.ChangeEvent
isSelected	javax.swing.event.ChangeListener
setSelected	stateChanged

Event Handling

Introduction:

It's now time to see how to get GUIs to respond to user actions like clicking on a button, typing text or dragging the mouse. These things are called **events**, and *responding* to them is called **event handling**.



Event Handling

Java.awt package

1. **Java.awt** is a package it will provide very good environment to develop graphical user interface applications.
2. AWT means (**Abstract Window Toolkit**). AWT is used to prepare the components but it is not providing any life to that components means by using AWT it is possible to create a static components.
3. To provide the life to the static components, we need to depend upon some other package called **java.awt.event** package.

Note

Java.awt package is used to prepare static components.

Java.awt.event package is used to provide the life to the static components.

GUI(graphical user interface):-

1. It is a mediator between end user and the program.
2. AWT is a package it will provide very good predefined support to design GUI applications.

Component :-

Component is an object which is displayed **pictorially** on the screen.

Ex:-

Button, Label, TextField ... etc

Container:-

Container is a GUI component, it is able to accommodate all other GUI components.

Ex:-

Frame, Applet.

Event:-

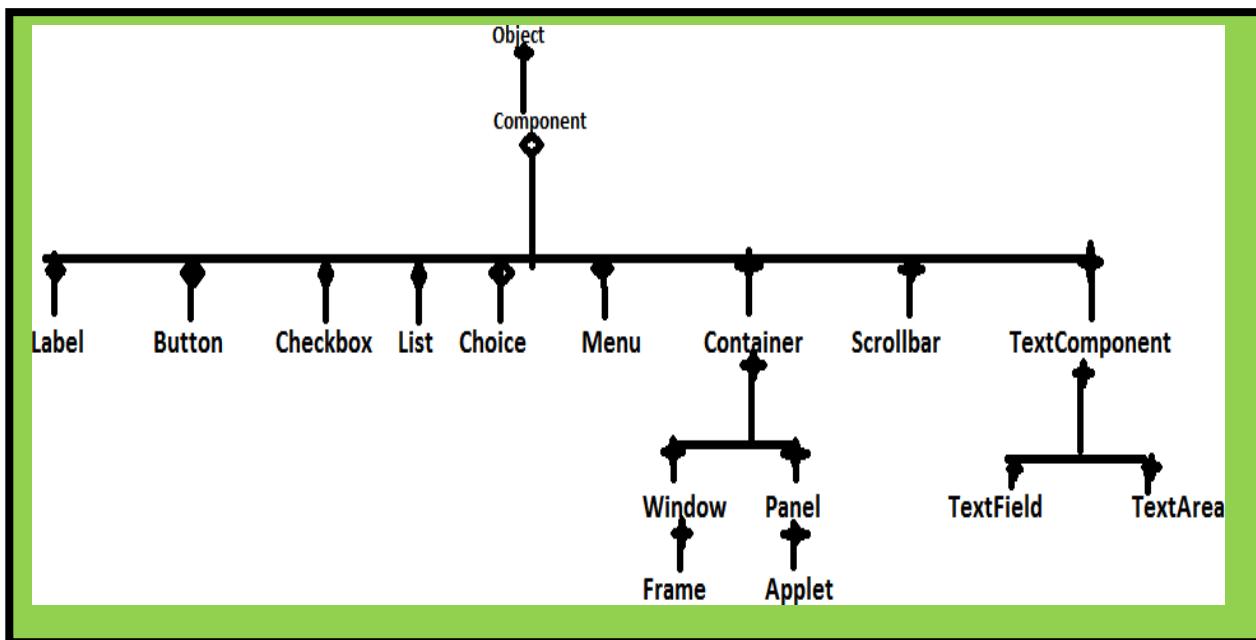
The event nothing but an action generated on the component or the change is made on the state of the object.

Ex:-

Button clicked, Checkboxchecked, Itemselected in the list, Scrollbar scrolled horizontal/vertically.

Classes of AWT:-

The classes present in the AWT package.



Frame:-

- 1) Frame is a class which is present in **java.awt package**.
- 2) Frame is a Basic component in AWT, because all the components displayed in a Frame.
- 3) We are displaying pictures on the Frame.
- 4) It is possible to display some text on the Frame.

Based on the above reasons the frame will become basic component in AWT.

Constructors:-

* create a Frame class object.

```
Frame f=new Frame();
```

* create a Frame class object and pass file

```
Frame f=new Frame("MyFrame");
```

* Take a subclass to the Frame and create object to the subclass.

```
class MyFrame extends Frame  
MyFrame f=new MyFrame();
```

Characteristics of the Frame:-

- 1) When we create a Frame class object the Frame will be created automatically with the invisible mode. To provide visible mode to following method.

public void setVisible(boolean b)

where b==true means visible mode.

where b==false means invisible mode.

Ex: **f.setVisible(true);**

- 2) When we created a Frame the initial size of the Frame is :

0 pixel height

0 pixel width

So it is not visible to use.

- To provide particular size to the Frame we have to use following method.

public void setSize(int width,int height)

Ex: **f.setSize(400,500);**

- 3) To provide title to the Frame explicitly we have to use the following method

public void setTitle(String Title)

Ex: **f.setTitle("MyFrame");**

- 4) When we create a Frame, the default background color of the Frame is white. If you want to provide particular color to the Frame we have to use the following method.

public void setBackground(color c)

Ex: **f.setBackground(Color.red);**

*****CREATION OF FRAME*****

```
import java.awt.*;
class Demo
{
    public static void main(String[] args)
    {
        //frame creation
        Frame f=new Frame();
        //set visibility
        f.setVisible(true);
        //set the size of the frame
        f.setSize(400,400);
        //set the background
        f.setBackground(Color.red);
        //set the title of the frame
        f.setTitle("myframe");
    }
}
```

CREATION OF FRAME BY TAKING USER DEFINED CLASS

```
import java.awt.*;
class MyFrame extends Frame
{
    MyFrame()
    {
        setVisible(true);
        setSize(500,500);
        setTitle("myframe");
        setBackground(Color.red);
    }
}
class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
}
```

To display text on the screen:-

1. If you want to display some textual message or some graphical shapes on the Frame then we have to **override paint()**, which is present in the **Frame class**.
public void paint(Graphics g)

2. To set a particular font to the text,we have to use **Font class** present in **java.awt package**

```
Font f=new Font(String type,int style,int size);
Ex: Font f= new Font("arial",Font.Bold,30);
```

Ex :-

```
import java.awt.*;
class Test extends Frame
{
    public static void main(String[] args)
    {
        Test t=new Test();
        t.setVisible(true);
        t.setSize(500,500);
        t.setTitle("myframe");
        t.setBackground(Color.red);
    }
    public void paint(Graphics g)
    {
        Font f=new Font("arial",Font.ITALIC,25);
```

```

        g.setFont(f);
        g.drawString("hi ratan how r u",100,100);
    }
}

```

Note:-

1. When we create a MyFrame class constructor, jvm executes MyFrame class constructor just before this JVM has to execute Frame class zero argument constructor.
2. In Frame class, zero argument constructor repaint() method will be executed, it will access predefined Frame class paint() method. But as per the requirement overriding paint() method will be executed.
3. Therefore the paint() will be executed automatically at the time of Frame creation.

Preparation of the components:-

Label :-

- 1) Label is a constant text which is displayed along with a **TextField** or **TextArea**.
- 2) **Label is a class** which is present in **java.awt package**.
- 3) To display the label we have to add that label into the frame for that purpose we have to use **add()** method present in the Frame class.

Constructor:-

```

Label l=new Label();
Label l=new Label("user name");

```

Ex :-

```

import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);
        Label l=new Label("user name:");
        f.add(l);
    }
}

```



TextField:-

- 1) TextField is an **editable area**.
 - 2) In TextField, we are able to provide **single line of text**.
 - 3) Enter Button doesn't work on TextField. To add TextField into the Frame we have to use **add()** method.
-
1. To set Text to the textarea we have to use the following method.

```
t.setText("Durga");
```

2. To get the text form the TextArea we have to use following method.

```
String s=t.getText();
```

3. To append the text into the TextArea.

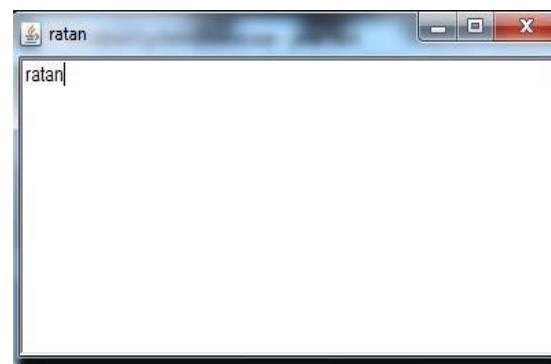
```
t.appendText("ratan");
```

Constructor:-

```
TextField tx=new TextField();
TextField tx=new TextField("ratan");
```

Ex :-

```
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);
//TextField tx=new TextField(); empty TextField
        TextField tx=new TextField("ratan");
//TextField with data
        f.add(tx);
    }
}
```



TextArea:-

- 1) TextArea is a class present in java.awt.package.
- 2) TextArea is a Editable Area. Enter button will work on TextArea.
- 3) To add the TextArea into the frame we have to use the add()

Constructors:-

```
TextArea t=new TextArea();
TextArea t=new TextArea(int rows,int columns);
```

4. To set Text to the textarea we have to use the following method.

```
t.setText("Durga");
```

5. To get the text form the TextArea we have to use fallowing method.

```
String s=t.getText();
```

6. To append the text into the TextArea.

```
t.appendText("ratan");
```

```
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);
        f.setLayout(new FlowLayout());
        Label l=new Label("user name:");
        TextArea tx=new TextArea(4,10);//4 character height 10 character width
        tx.appendText("ratan");
        tx.setText("aruna");
        System.out.println(tx.getText());
        f.add(l);
        f.add(tx);
    }
}
```



Choice:-

- 1) Choice is a class present in java.awt package.
- 2) List is allows to select multiple items but choice is allow to select single Item.

Constructor:-

```
Choice ch=new Choice();
```

Methods :-

1. To add items to the choice we have to use following method.

```
ch.add("HYD");
ch.add("Chennai");
ch.add("BANGALORE");
```

2. To remove item from the choice based on the string.

```
ch.remove("HYD");
ch.remove("BANGALORE");
```

3. To remove the item based on the index position
`ch.remove(2);`
4. To remove the all elements
`ch.removeAll();`
5. To inset the data into the choice based on the particular position.
`ch.insert(2,"ratan");`
6. To get selected item from the choice we have to use following method.
`String s=ch.getSelectedItem();`
7. To get the selected item index number we have to use fallowing method
`int a=ch.getSelectedIndex();`

ex:-

```
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);

        Choice ch=new Choice();
        ch.add("c");
        ch.add("cpp");
        ch.add("java");
        ch.add(".net");
        ch.remove(".net");
        ch.remove(0);
        ch.insert("ratan",0);
        f.add(ch);

        System.out.println(ch.getItem(0));
        System.out.println(ch.getSelectedItem());
        System.out.println(ch.getSelectedIndex());
        //ch.removeAll();
    }
}
```

List:-

- 1) List is a class it is present in `java.awt` package
- 2) List is providing list of options to select. Based on your requirement we can select any number of elements. To add the List to the frame we have to use `add()` method.

CONSTRUCTOR:-

- 1) List l=new List();



D:\bank>javac Test.java
D:\bank>java Test
ratan
ratan
0

It will creates the list by default size is four elements. And it is allow selecting the only one item at a time.

2) List l=new List(3);

It will display the three items size and it allows selecting the only single item.

3) List l=new List(5,true);

It will display the five items and it allows selecting the multiple items.

Methods:-

1. To add the elements to the List we have to use following method.

```
l.add("c");
l.add("cpp");
l.add("java");
l.add("ratan",0);
```

2. To remove element from the List we have to use following method.

```
l.remove("c");
l.remove(2);
```

3. To get selected item from the List we have to use following method.

```
String x=l.getSelectedItem();
```

4. To get selected items from the List we have to use following method.

```
String[] x=s.getSelectedItems()
```

Ex:-

```
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);
        f.setLayout(new FlowLayout());

        List l=new List(4,true);
        l.add("c");
        l.add("cpp");
        l.add("java");
        l.add(".net");
        l.add("ratan");
        l.add("arun",0);
        l.remove(0);
        f.add(l);
        System.out.println(l.getSelectedItem());
    }
}
```



Checkbox:-

- 1) Checkbox is a class present in `java.awt` package.
- 2) The user can select more than one checkbox at a time. To add the checkbox to the frame we have to use `add()` method.

Constructor:-

- 1) `Checkbox cb1=new CheckBox();`
`cb1.setLabel("BTECH");`
- 2) `Checkbox cb1=new CheckBox("MCA");`
- 3) `Checkbox cb3=new CheckBox("BSC",true);`

Methods:-

1. To set a label to the CheckBox explicitly and to get label from the CheckBox we have to use the following method.
`cb.setLabel("BSC");`
2. To get the label of the checkbox we have to use fallowing method.
`String str=cb.getLabel();`
3. To get state of the CheckBox and to set state to the CheckBox we have to use following method.
`Boolean b=ch.getState();`

Ex:-

```
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);
        Checkbox cb1=new Checkbox("BTECH",true);
        f.add(cb1);
        System.out.println(cb1.getLabel());
```

```
System.out.println(cb1.getState());
    }
}
```

```
D:\bank>javac Test.java
```

```
D:\bank>java Test
```

```
BTECH  
true
```



RADIO BUTTON:-

- 1) AWT does not provide any predefined support to create RadioButtons.
- 2) It is possible to select only item is selected from group of item. To add the RadioButton to the frame we have to use `add()` method.

By using two classes we create Radio Button those are

- a)CheckBoxgroup
- b)CheckBox

step 1:- Create CheckBox group object.

```
CheckBoxGroup cg=new CheckBoxGroup();
```

step 2:- pass Checkbox object to the CheckboxGroup class then the radio buttons are created.

```
CheckBox cb1=new CheckBox("male",cg,false);
CheckBox cb2=new CheckBox("female",cg,false);
```

Methods:-

- 1) To set status and to get status we have to use `setState()` and `getState()` methods.

```
String str=cb.getState();
Cb.setState();
```

- 2) To get Label and to set Label we have to use following methods.

```
String str=getLabel()
setLabel("female").
```

Ex:-

```
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);
```

```
CheckboxGroup cg=new CheckboxGroup();
Checkbox cb1=new Checkbox("male",cg,true);
f.add(cb1);
System.out.println(cb1.getLabel());
System.out.println(cb1.getState());
}
```

```
D:\bank>javac Test.java
```

```
D:\bank>java Test
```

```
male  
true
```



Layout Managers:-

```
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);
        Label l1=new Label("user name:");
        TextField tx1=new TextField();
        Label l2=new Label("password:");
        TextField tx2=new TextField();
        Button b=new Button("login");
        f.add(l1);
        f.add(tx1);
        f.add(l2);
        f.add(tx1);
        f.add(b);
    }
}
```

LAYOUT MANAGERS:

A layout manager is a Java object associated with a particular component, almost always a *background* component. The layout manager controls the components contained *within* the component the layout manager is associated with. In other words, if a frame holds a panel, and the panel holds a button, the panel's layout manager controls the size and placement of the button, while the frame's layout manager controls the size and placement of the panel. The button, on the other hand, doesn't need a layout manager because the button isn't holding other components.

If a panel holds five things, even if those five things each have their own layout managers, the size and location of the five things in the panel are all controlled by the panel's layout manager. If those five things, in turn, hold *other* things, then those *other* things are placed according to the layout manager of the thing holding them.

When we say hold we really mean add as in, a panel holds a button because the button was added to the panel using something like:

```
myPanel.add(button);
```

Layout managers come in several flavors, and each background component can have its own layout manager. Layout managers have their own policies to follow when building a layout. For example, one layout manager might insist that all components in a panel must be the same size, arranged in a grid, while another layout manager might let each component choose its own size, but stack them vertically. Here's an example of nested layouts:

```
JPanel panelA = new JPanel();
JPanel panelB = new JPanel();
panelB.add(new JButton("button 1"));
panelB.add(new JButton("button 2"));
panelB.add(new JButton("button 3"));
panelA.add(panelB);
```

How does the layout manager decide?

Different layout managers have different policies for arranging components (like, arrange in a grid, make them all the same size, stack them vertically, etc.) but the components being laid out do get at least *some* small say in the matter. In general, the process of laying out a background component looks something like this:

A layout scenario:

1. Make a panel and add three buttons to it.
2. The panel's layout manager asks each button how big that button prefers to be.
3. The panel's layout manager uses its layout policies to decide whether it should respect all, part, or none of the buttons' preferences.
4. Add the panel to a frame.
5. The frame's layout manager asks the panel how big the panel prefers to be.
6. The frame's layout manager uses its layout policies to decide whether it should respect all, part, or none of the panel's preferences.

Different layout managers have different policies.

Some layout managers respect the size the component wants to be. If the button wants to be 30 pixels by 50 pixels, that's what the layout manager allocates for that button. Other layout managers

respect only part of the component's preferred size. If the button wants to be 30 pixels by 50 pixels, it'll be 30 pixels by however wide the button's background panel is. Still other layout managers respect the preference of only the largest of the components being layed out, and the

rest of the components in that panel are all made that same size. In some cases, the work of the layout manager can get very complex, but most of the time you can figure out what the layout manager will probably do, once you get to know that layout manager's policies.

The Big Three layout managers: border, flow, and box.

BorderLayout

- 1) A BorderLayout manager divides a background component into five regions.
- 2) You can add only one component per region to a background controlled by a BorderLayout manager.
- 3) Components laid out by this manager usually don't get to have their preferred size.
- 4) BorderLayout is the default layout manager for a frame!**
- 5) BorderLayout cares about five regions: east, west, north, south, and center.

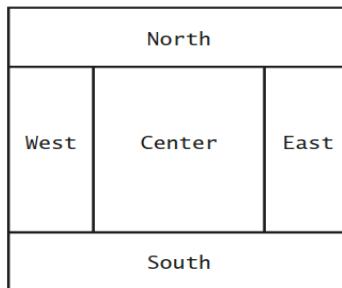


Figure: Components expand to fill space in BorderLayout

// program for BorderLayout

```
import java.awt.*;
class MyFrame extends Frame
{
    Button b1,b2,b3,b4,b5;
    MyFrame()
    {
        this.setBackground(Color.green);
        this.setSize(400,400);
        this.setVisible(true);
        this.setLayout(new BorderLayout());
        b1=new Button("Boys");
        b2=new Button("Girls");
        b3=new Button("management");
        b4=new Button("Teaching Staff");
        b5=new Button("non-teaching staff");
        this.add("North",b1);
        this.add("Center",b2);
        this.add("South",b3);
        this.add("East",b4);
        this.add("West",b5);
    }
}
class Demo33
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
}
```

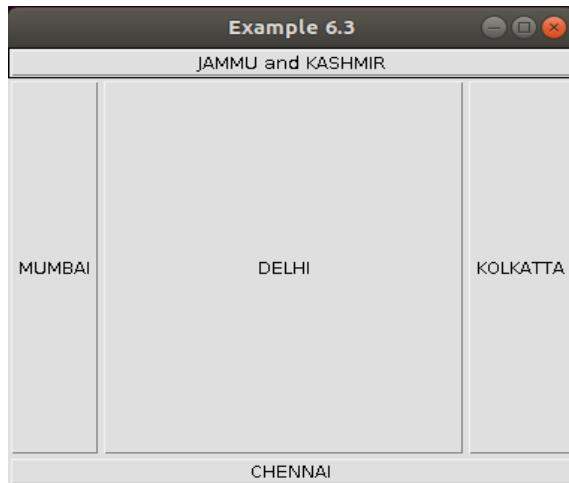
output:



Example 2 of BorderLayout Manager:

```
// PROGRAM USING BorderLayout Manager
import java.awt.*;
class ButtonFrame extends Frame
{
    ButtonFrame(String s)
    {
        super(s);
        setSize(150,100);
        setLayout(new BorderLayout());
        add(new Button("JAMMU and KASHMIR"),BorderLayout.NORTH);
        add(new Button("KOLKATTA"),BorderLayout.EAST);
        add(new Button("CHENNAI"),BorderLayout.SOUTH);
        add(new Button("MUMBAI"),BorderLayout.WEST);
        add(new Button("DELHI"),BorderLayout.CENTER);
        setVisible(true);
    }
}
class TestButtonFrame2
{
    public static void main(String[] args)
    {
        ButtonFrame f=new ButtonFrame("Example 6.3");
    }
}
```

Output:



The add () method declared in the Container class has five versions. The one used in Example is declared as

public void add(Component component, Object constraint)

When the container's layout is a BorderLayout object, the add() method expects the constraint argument to be one of the five objects defined in the BorderLayout class: NORTH, EAST, SOUTH, WEST, or CENTER. These determine which of the five possible positions the component will be given.

FlowLayout:

- 1) A FlowLayout manager acts kind of like a word processor, except with components instead of words.
- 2) Each component is the size it wants to be, and they're laid out left to right in the order that they're added, with "word-wrap" turned on.
- 3) So when a component won't fit horizontally, it drops to the next "line" in the layout.
- 4) **FlowLayout is the default layout manager for a panel!**
- 5) FlowLayout cares about the flow of the components: left to right, top to bottom, in the order they were added.

Example 1:

//PROGRAM ON FLOWLAYOUT

```
import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame
{
    Label l1,l2;
    TextField tx1,tx2;
    Button b;
    MyFrame()
    {
        this.setVisible(true);
        this.setSize(340,500);
        this.setBackground(Color.green);
        this.setTitle("Epass");
        l1=new Label("user name:");
        l2=new Label("password:");
        tx1=new TextField(25);
        tx2=new TextField(25);
        b=new Button("login");
        tx2.setEchoChar('*');
        this.setLayout(new FlowLayout());
        this.add(l1);
        this.add(tx1);
        this.add(l2);
        this.add(tx2);
        this.add(b);
    }
}
class Demo44
{
```

OUTPUT:

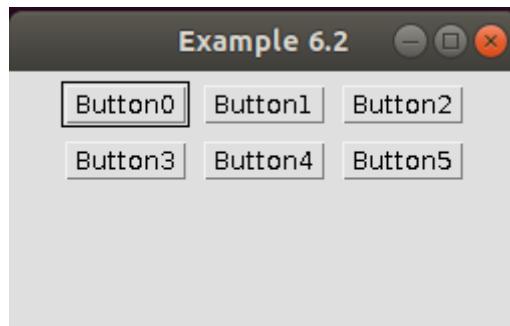


Example 2 of FlowLayout Manager:

This program creates a ButtonFrame object named b and six anonymous **Button objects** which are added to the frame as components. It passes an anonymous **FlowLayout object** to the frame's **setLayout()** method to arrange the buttons in a flow layout:

```
// PROGRAM USING FlowLayout Manager
import java.awt.Button;
import java.awt.Frame;
import java.awt.FlowLayout;
class ButtonFrame extends Frame
{
    ButtonFrame(String s)
    {
        super(s);
        setSize(200,100);
        setLayout(new FlowLayout());
        for(int i=0;i<6;i++)
        {
            add(new Button("Button"+i));
        }
        setVisible(true);
    }
}
class TestButtonFrame
{
    public static void main(String[] args)
    {
        ButtonFrame b=new ButtonFrame("Example 6.2");
    }
}
```

Output:



The displayed frame looks like the picture here. The for loop creates the six buttons and makes them components of the frame. Drag on the edge of the frame window to resize it, making it taller and narrower. See how the buttons flow within the frame, rearranging themselves but still maintaining their "words-on-a-page" arrangement. Note that the size of each button is set by default according to the size of its label.

EXAMPLE Using the GridLayout Manager:

```
// PROGRAM USING GridLayout Manager to arrange 12 buttons in a 4-by-3 grid
import java.awt.*;
class ButtonFrame extends Frame
{
    ButtonFrame(String s)
    {
        super(s);
        setSize(300,200);
        setLayout(new GridLayout(4,3));
        for(int i=0;i<12;i++)
        {
            add(new Button("Button"+i));
        }
        setVisible(true);
    }
}
class TestButtonFrame3
{
    public static void main(String[] args)
    {
        ButtonFrame f=new ButtonFrame("Example 6.4");
    }
}
```



The arguments 4 and 3 are passed to the GridLayout constructor, telling it to use 4 rows and 3 columns in the grid.

Note that we have switched to using the wild card character * in the import statement
import java.awt.*;
to avoid listing the AWT classes separately.

1. Event delegation model:-

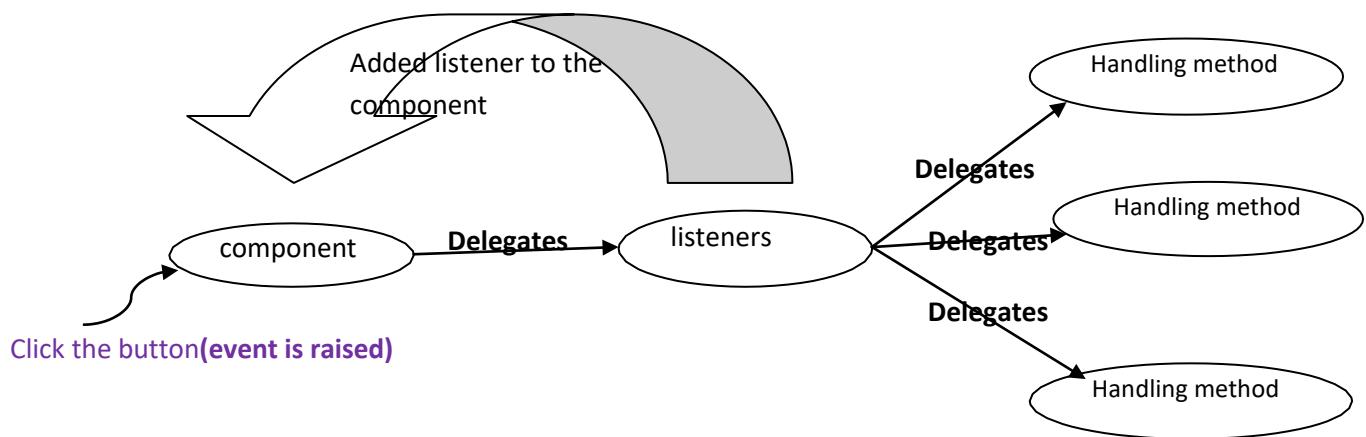
1. When we create a component, the components visible on the screen. But it is not possible to perform any action. for example: button.
2. Whenever we create a Frame it can be minimized and maximized and resized but it is not possible to close the Frame even if we click on Frame close Button.
3. The Frame is a static component so it is not possible to perform actions on the Frame.
4. To make **static component into dynamic component** we have to add some actions to the Frame.
5. To attach actions to the Frame component, we need event delegation model.

Whenever we click on button no action will be performed clicking like this is called event.

Event: - Event is nothing but a particular action generated on the particular component.

1. When an event generates on the component the component is unable to respond because **component can't listen the event**.
2. To make the component listen the event, we have to **add listeners to the component**.
3. Wherever we are adding listeners to the component, the component is able to respond based on the generated event.
4. A listener is a interface which contain **abstract methods** and it is present in **java.awt.event** package
5. The listeners are different from component to component.

A component delegate event to the listener and listener is designates the event to appropriate method by executing that method only the event is handled. This is called Event Delegation Model.



Note: -

To attach a particular listener to the Frame we have to use following method

Public void AddxxxListener(xxxListener e)

Where xxx may be ActionListener,windowListener

The Appropriate Listener for the Frame is “windowListener”

S.no.	GUI Component	Event Name	Listener Name	Listener Methods
1	Frame	WindowEvent	WindowListener	1.public void windowOpened(WindowEvent e) 2.public void windowActivated(WindowEvent e) 3.public void windowDeactivated(WindowEvent e) 4.public void windowClosing(WindowEvent e) 5.public void windowClosed(WindowEvent e) 6.public void windowIconified(WindowEvent e) 7.public void windowDeiconified(WindowEvent e)
2	TextField	ActionEvent	ActionListener	1.public void actionPerformed(ActionEvent ae)
3	TextArea	ActionEvent	ActionListener	1.public void actionPerformed(ActionEvent ae)
4	Menu	ActionEvent	ActionListener	1.public void actionPerformed(ActionEvent ae)
5	Button	ActionEvent	ActionListener	1.public void actionPerformed(ActionEvent ae)
6	Checkbox	ItemEvent	ItemListener	1.public void itemStateChanged(ItemEvent e)
6	Radio	ItemEvent	ItemListener	1.public void itemStateChanged(ItemEvent e)
7	List	ItemEvent	ItemListener	1.public void itemStateChanged(ItemEvent e)
8	Choice	ItemEvent	ItemListener	1.public void itemStateChanged(ItemEvent e)
9	Scrollbar	AdjustmentEvent	AdjustmentListener	1.public void adjustmentValueChanged(AdjustmentEvent e)
10	Mouse	MouseEvent	MouseListener	1.public void mouseEntered(MouseEvent e) 2.public void mouseExited(MouseEvent e) 3.public void mousePressed(MouseEvent e) 4.public void mouseReleased(MouseEvent e) 5.public void mouseClicked(MouseEvent e)
11	Keyboard	KeyEvent	KeyListener	1.public void keyTyped(KeyEvent e) 2.public void keyPressed(KeyEvent e) 3.public void keyReleased(KeyEvent e)

Table: Appropriate listeners for components

To register a listener:

Method name should be prefixed with **add**.

1. public void addMyActionListener(MyActionListener l) **(valid)**
2. public void registerMyActionListener(MyActionListener l) **(invalid)**
3. public void addMyActionListener(ActionListener l) **(invalid)**

To unregister a listener:

The method name should be prefixed with **remove**.

1. public void removeMyActionListener(MyActionListener l) **(valid)**
2. public void unregisterMyActionListener(MyActionListener l) **(invalid)**
3. public void removeMyActionListener(ActionListener l) **(invalid)**
4. public void deleteMyActionListener(MyActionListener l) **(invalid)**

Note :- by using WindowAdaptor class we can close the frame. Internally WindowAdaptor class implements WindowListener interface. Hence WindowAdaptor class contains empty implementation of abstract methods.

```
//PROVIDING CLOSING OPTION TO THE FRAME
import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame
{
    MyFrame()
    {
        this.setVisible(true);
        this.setSize(500,500);
        this.setBackground(Color.red);
        this.setTitle("RANGAMMA MANGAMMA FRAME");
        this.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {
                System.exit(0);
            }
        });
    }
}
class FrameEx5
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
}
```

OUTPUT:



```
***WRITE SOME TEXT INTO THE FRAME*****
import java.awt.*;
class MyFrame extends Frame
{
    MyFrame()
    {
        this.setVisible(true);
        this.setSize(500,500);
        this.setBackground(Color.red);
        this.setTitle("rattaiah");
    }
    public void paint(Graphics g)
    {
        Font f=new Font("arial",Font.BOLD,20);
        g.setFont(f);
        this.setForeground(Color.green);
        g.drawString("HI BTECH ",100,100);
        g.drawString("good boys &",200,200);
        g.drawString("good girls",300,300);
    }
}
class FrameEx
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
}
```

Adapter class

Java adapter classes provide the default implementation of listener [interfaces](#). If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces. So it saves code.

The adapter classes are found in **java.awt.event**, **java.awt.dnd** and **javax.swing.event packages**. The Adapter classes with their corresponding listener interfaces are given below.

java.awt.event Adapter classes

Adapter class	Listener interface
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
HierarchyBoundsAdapter	HierarchyBoundsListener

java.awt.dnd Adapter classes

Adapter class	Listener interface
DragSourceAdapter	DragSourceListener
DragTargetAdapter	DragTargetListener

javax.swing.event Adapter classes

Adapter class	Listener interface
MouseInputAdapter	MouseInputListener
InternalFrameAdapter	InternalFrameListener

Java WindowAdapter Example

```
import java.awt.*;
import java.awt.event.*;
public class AdapterExample
{
    Frame f;
    AdapterExample()
    {
        f=new Frame("Window Adapter");
        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e) {
                f.dispose();
            }
        });

        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new AdapterExample();
    }
}
```

Output:



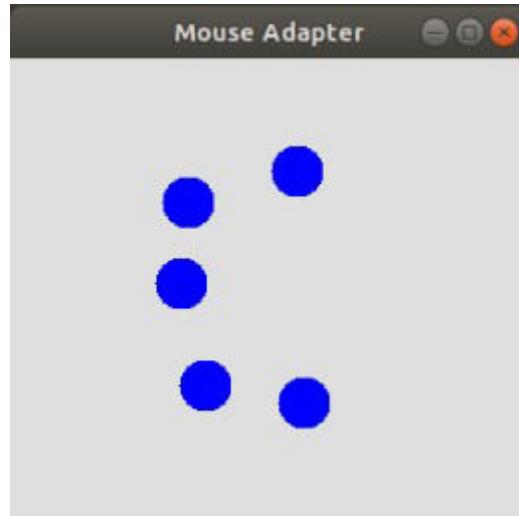
Java MouseAdapter Example

```
import java.awt.*;
import java.awt.event.*;
public class MouseAdapterExample extends MouseAdapter
{
    Frame f;
    MouseAdapterExample()
    {
        f=new Frame("Mouse Adapter");
        f.addMouseListener(this);

        f.setSize(300,300);
        f.setLayout(null);
        f.setVisible(true);
    }
    public void mouseClicked(MouseEvent e)
    {
        Graphics g=f.getGraphics();
        g.setColor(Color.BLUE);
        g.fillOval(e.getX(),e.getY(),30,30);
    }

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

Output:

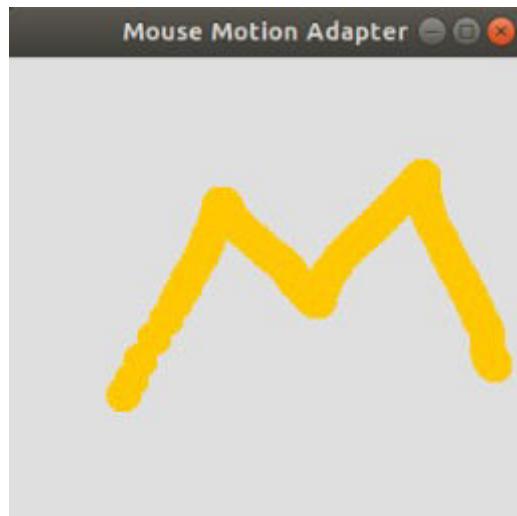


Java MouseMotionAdapter Example

```
import java.awt.*;
import java.awt.event.*;
public class MouseMotionAdapterExample extends MouseMotionAdapter
{
    Frame f;
    MouseMotionAdapterExample()
    {
        f=new Frame("Mouse Motion Adapter");
        f.addMouseMotionListener(this);

        f.setSize(300,300);
        f.setLayout(null);
        f.setVisible(true);
    }
    public void mouseDragged(MouseEvent e)
    {
        Graphics g=f.getGraphics();
        g.setColor(Color.ORANGE);
        g.fillOval(e.getX(),e.getY(),20,20);
    }
    public static void main(String[] args)
    {
        new MouseMotionAdapterExample();
    }
}
```

Output:



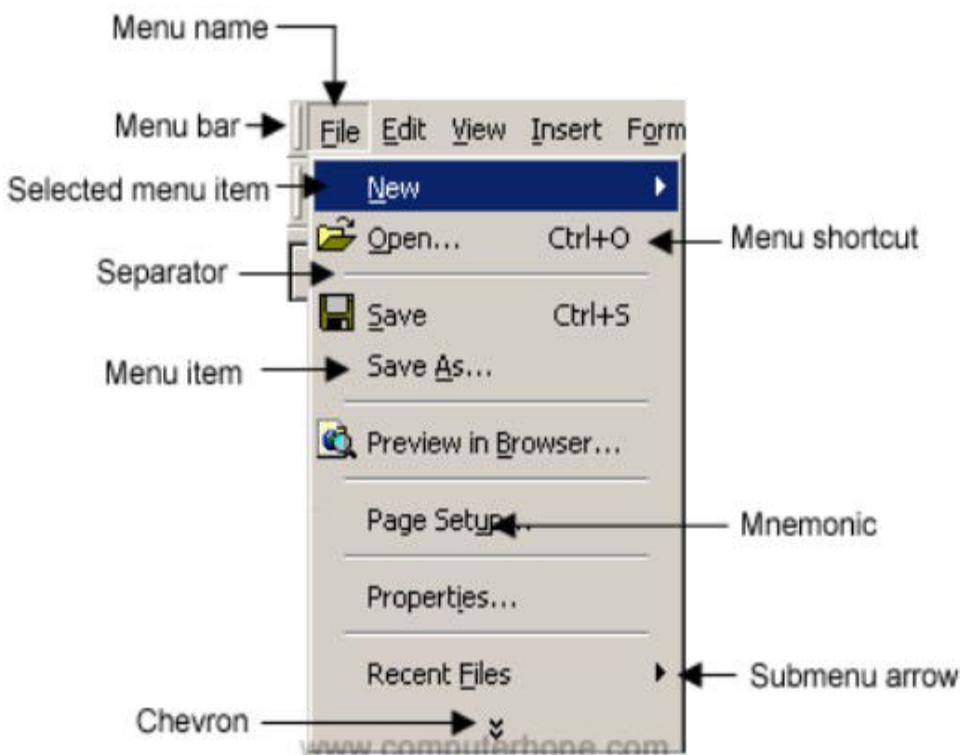
Menu, MenuItems andMenuBar class

In this article, we are going to understand how to add a menu bar, menu and its menu items to the window application.

- A menu bar can be created using **MenuBar** class.
- A menu bar may contain one or multiple menus, and these menus are created using **Menu** class.
- A menu may contain one of multiple menu items and these menu items are created using **MenuItem** class.

Simple constructors ofMenuBar, Menu and MenuItem

Constructor	Description
public MenuBar()	Creates a menu bar to which one or many menus are added.
public Menu(String title)	Creates a menu with a title.
public MenuItem(String title)	Creates a menu item with a title.



EXAMPLE: Write a java program to create a menubar, 3 menus new ,option, edit, and 3 menuitems open,save,saveas in new menu

```
//MENU ITEMS
import java.awt.*;
import java.awt.event.*;
class myframe extends Frame implements ActionListener
{
    String label="";
    MenuBar mb;
    Menu m1,m2,m3;
    MenuItem mil,mi2,mi3;
    myframe()
    {
        this.setSize(300,300);
        this.setVisible(true);
        this.setTitle("myframe");
        this.setBackground(Color.green);

        mb=new MenuBar();
        this.setMenuBar(mb);

        m1=new Menu("new");
        m2=new Menu("option");
        m3=new Menu("edit");

        mb.add(m1);
        mb.add(m2);
        mb.add(m3);

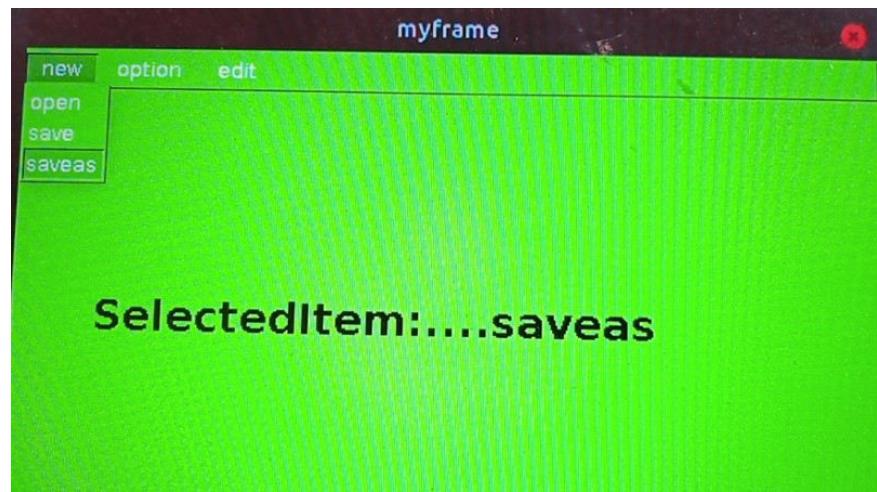
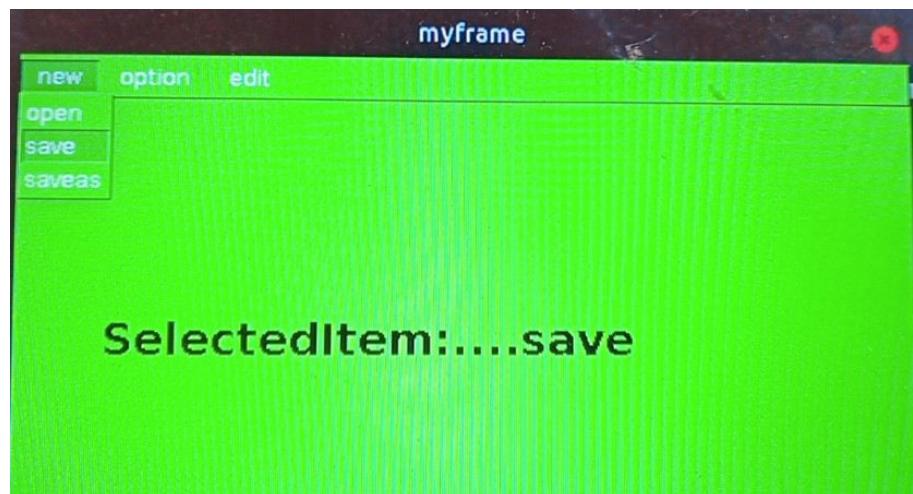
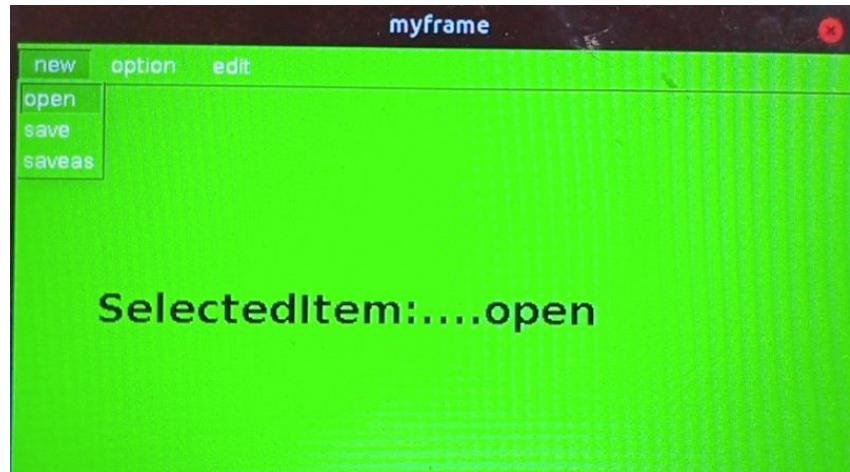
        mil=new MenuItem("open");
        mi2=new MenuItem("save");
        mi3=new MenuItem("saveas");

        mil.addActionListener(this);
        mi2.addActionListener(this);
        mi3.addActionListener(this);

        m1.add(mil);
        m1.add(mi2);
        m1.add(mi3);
    }
    public void actionPerformed(ActionEvent ae)
    {
        label=ae.getActionCommand();
        repaint();
    }

    public void paint(Graphics g)
    {
        Font f=new Font("arial",Font.BOLD,25);
        g.setFont(f);
        g.drawString("SelectedItem:...."+label,50,200);
    }
}
class Demo7
{
    public static void main(String[] args)
    {
        myframe f=new myframe();
    }
}
```

OUTPUT:



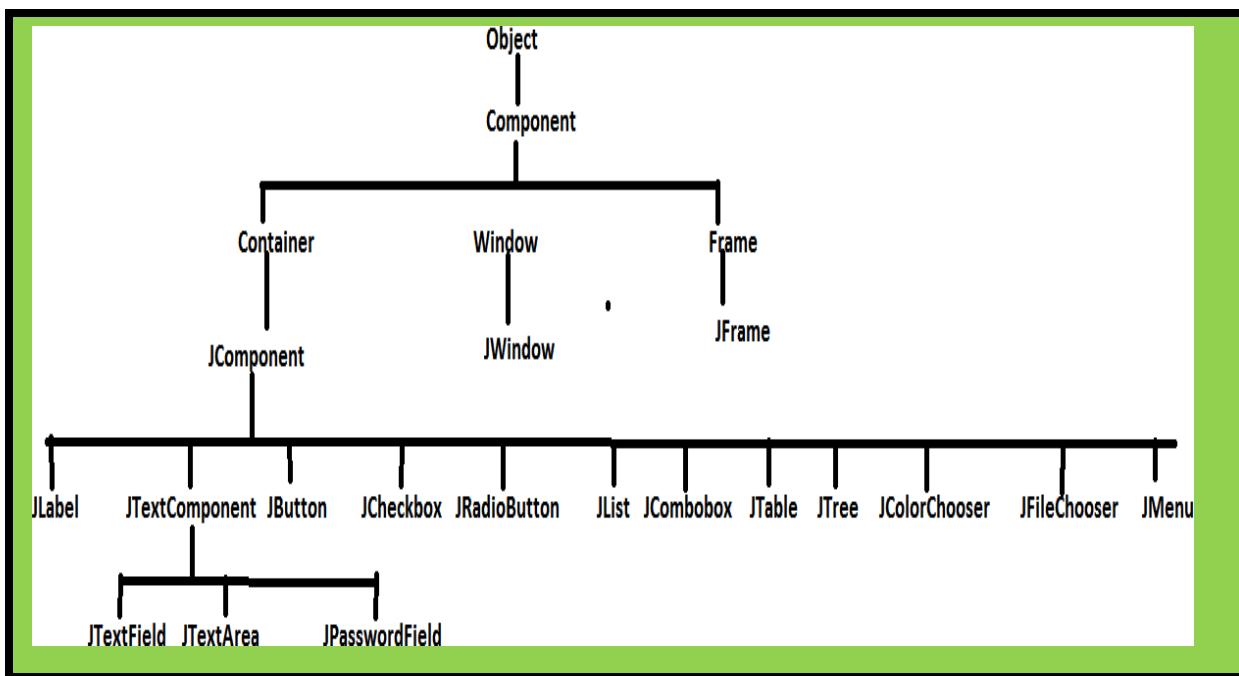
SWINGS

1. Sun Micro Systems introduced AWT to prepare GUI applications.
2. awt components not satisfy the client requirement.
3. An alternative to AWT Netscape Communication has provided set of GUI components in the form of IFC(Internet Foundation Class).
4. IFC also provide less performance and it is not satisfy the client requirement.
5. In the above contest[sun+Netscape] combine and introduced common product to design GUI applications.

Differences between awt and Swings:

1. AWT components are heavyweight component but swing components are light weight component.
2. AWT components consume more number of system resources Swings consume less number of system resources.
3. AWT components are platform dependent but Swings are platform independent.
4. AWT is provided less number of components where as swings provides more number of components.
5. AWT doesn't provide Tooltip Test support but swing components have provided Tooltip test support.
6. In awt for only window closing :
`windowListener`
`windowAdaptor`
In case of swing use small piece of code.
 - i. `f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`
7. AWT will not follow MVC but swing follows MVC Model View Controller It is a design pattern to provide clear separation b/w controller part,model part,view part.
 - a. Controller is a normal java class it will provide controlling.
 - b. View part provides presentation
 - c. Model part provides required logic.
8. In case of AWT we will add the GUI components in the Frame directly but Swing we will add all GUI components to panes to accommodate GUI components.

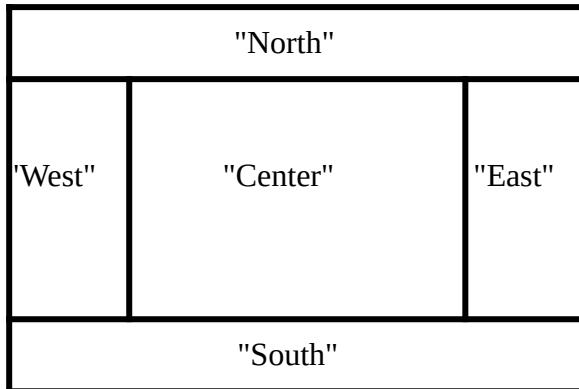
Classes of swing:-



JFrame:

A window with a title bar, a resizable border, and possibly a menu bar.

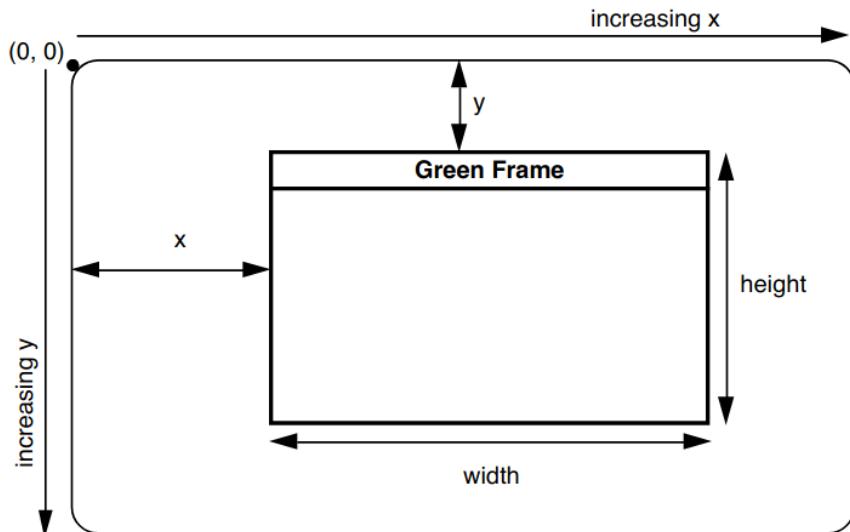
- A frame is not attached to any other surface.
- It has a content pane that acts as a container.
- The container uses BorderLayout by default.



- A layout manager, an instance of one of the layout classes, describes how components are placed on a container.

Creating a JFrame

```
JFrame jf = new JFrame("title");      // or JFrame()  
  
jf.setSize(300, 200);                // width, height in pixels (required)  
jf.setVisible(true);                 // (required)  
  
jf.setTitle("New Title");  
jf.setLocation(50, 100);              // x and y from upper-left corner
```



Placing Components

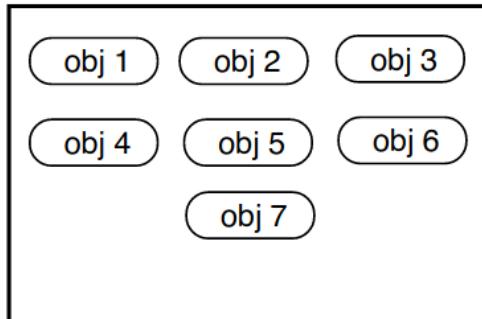
```
Container cp = jf.getContentPane();  
  
cp.add(c1, "North");  
cp.add(c2, "South");  
  
or      cp.add(c1, BorderLayout.NORTH);  
  
// Java has five constants like this
```

The constant `BorderLayout.NORTH` has the value "North".

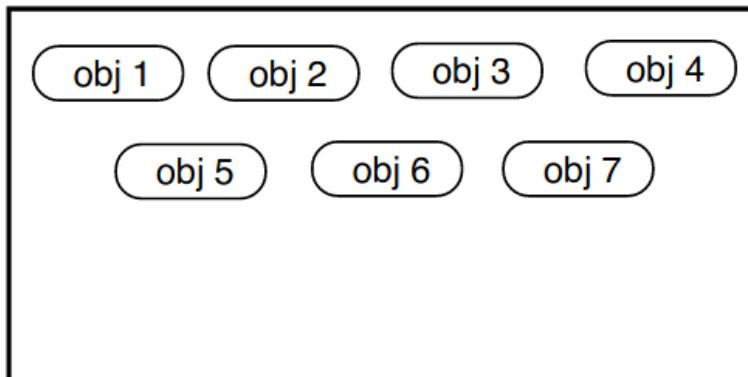
JPanel

An invisible Container used to hold components or to draw on.

- Uses FlowLayout by default (left-to-right, row-by-row).



- If the container is resized, the components will adjust themselves to new positions.



- Any component may be placed on a panel using add.

```
JPanel jp = new JPanel();
jp.add(componentObj);
```

- A panel has a Graphics object that controls its surface.

Subclass JPanel and override the method

```
void paintComponent(Graphics g) to
describe the surface of the panel.
```

Default: Blank background matching existing background color.

Set the background to a particular color using:

```
jp.setBackground(new Color(255, 204, 153));
```

The Graphics class:

The class Graphics supplies a number of commands for drawing.

```
void drawRect(int x, int y, int width, int height);  
  
void drawRoundRect(int x, int y, int w, int h,  
                    int arcWidth, int arcHeight);  
  
void drawOval(int x, int y, int width, int height);  
  
void fillRect(int x, int y, int width, int height);  
  
void fillRoundRect(int x, int y, int w, int h, int arcWidth, int arcHeight);  
  
void fillOval(int x, int y, int width, int height);  
  
void drawString(String text, int x, int y);  
  
void drawLine(int x1, int y1, int x2, int y2);  
  
void draw3DRect(int x, int y, int w, int h, boolean raised);  
  
void fill3DRect(int x, int y, int w, int h, boolean raised);  
  
void drawArc(int x, int y, int w, int h, int startAngle, int arcAngle);  
  
void fillArc(int x, int y, int w, int h, int startAngle, int arcAngle);  
  
void setColor(Color c);  
  
void setFont(Font f);  
  
void drawPolygon(int [] x, int [] y, int numPoints);  
  
void fillPolygon(int [] x, int [] y, int numPoints);
```

The coordinates are in pixels in the Java coordinate system. The shapes are drawn in the order in which the graphic commands are executed. If shapes overlap, the one drawn later covers those drawn previously.

Drawing Methods of the Graphics Class

Sr.No	Method	Description
1.	clearRect()	Erase a rectangular area of the canvas.
2.	copyArea()	Copies a rectangular area of the canvas to another area
3.	drawArc()	Draws a hollow arc
4.	drawLine()	Draws a straight line
5.	drawOval()	Draws a hollow oval
6.	drawPolygon()	Draws a hollow polygon
7.	drawRect()	Draws a hollow rectangle
8.	drawRoundRect()	Draws a hollow rectangle with rounded corners
9.	drawString()	Display a text string
10.	fillArc()	Draws a filled arc
11.	fillOval()	Draws a filled Oval
12.	fillPolygon()	Draws a filled Polygon
13.	fillRect()	Draws a filled rectangle
14.	fillRoundRect()	Draws a filled rectangle with rounded corners
15.	getColor()	Retrieves the current drawing color
16.	getFont()	Retrieves the currently used font
17.	getFontMetrics()	Retrieves the information about the current font
18.	setColor()	Sets the drawing color
19.	setFont()	Sets the font

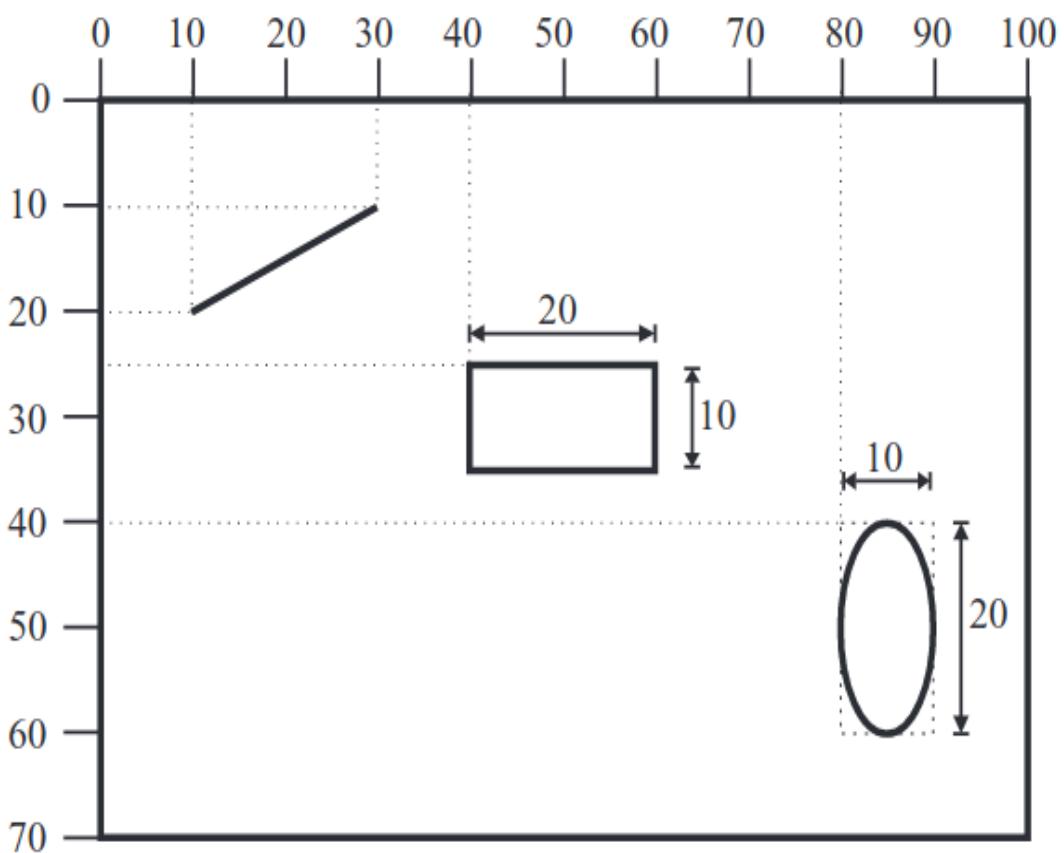


Figure: The effect of the commands `drawLine(10,20,30,10)`, `drawRect(40,25,20,10)` and `drawOval(80,40,10,20)`. The dotted lines and the dimensions do not appear in the graphic. The bounding rectangle for the oval is shown as a dotted line

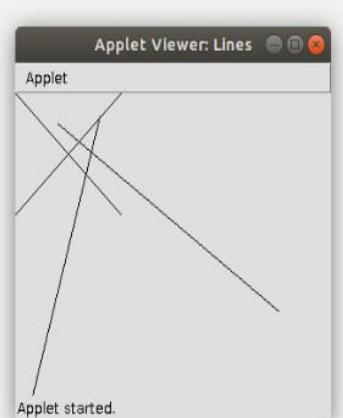
Lines :

drawLine(xstart,ystart,xend,yend)

draws a line segment between the points with coordinates (xstart,ystart) and (xend,yend).

Program 1:

```
//Drawing Lines
import java.awt.*;
import java.applet.*;
/*
<applet code="Lines" width=300 Height=250>
</applet>
*/
public class Lines extends Applet
{
    public void paint(Graphics g)
    {
        g.drawLine(0,0,100,100);
        g.drawLine(0,100,100,0);
        g.drawLine(40,25,250,180);
        g.drawLine(5,290,80,19);
    }
}
```



Save As: Lines.java

Compilation: javac Lines.java

Run: appletviewer Lines.java

Rectangles:

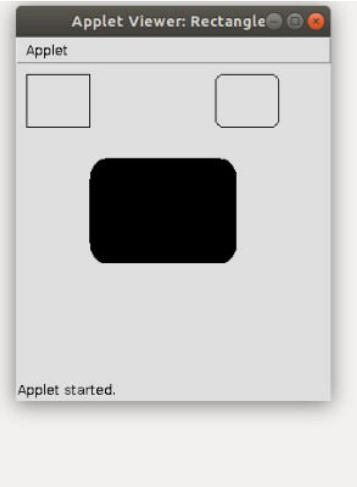
drawRect(xleft,ytop,width,height)

draws the contour of an axes-aligned rectangle. The upper left corner of the rectangle is at (xleft, ytop). It is width pixels wide and height pixels tall.

FillRect draws a filled rectangle using the current colour, otherwise like **drawRect**.

Program 2:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="Rectangle" width=300 Height=300>
</applet>
*/
public class Rectangle extends Applet
{
    public void paint(Graphics g)
    {
        g.drawRect(10,10,60,50);
        g.fillRect(100,100,100,0);
        g.drawRoundRect(190,10,60,50,15,15);
        g.fillRoundRect(70,90,140,100,30,40);
    }
}
```



Save As: **Rectangle.java**

Compile: **javac Rectangle.java**

Run: **appletviewer Rectangle.java**

Circles and Ellipses:

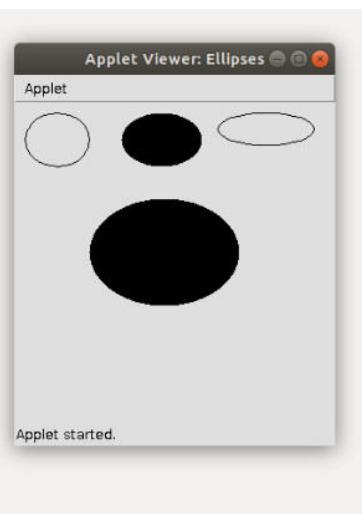
drawOval(xleft,ytop,width,height)

draws the contour of an ellipse. The axes of the ellipse are parallel to the coordinate axes. The upper left corner of the bounding rectangle of the ellipse is at(xleft, ytop)and is called the reference point. The horizontal axis of the ellipse has length width,the vertical one height.

FillOval draws a filled ellipse using the current colour, otherwise like drawOval.

Program 3:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="Ellipses" width=300 Height=300>
</applet>
*/
public class Ellipses extends Applet
{
    public void paint(Graphics g)
    {
        g.drawOval(10,10,60,50);
        g.fillOval(100,10,75,50);
        g.drawOval(190,10,90,30);
        g.fillOval(70,90,140,100);
    }
}
```



Save As : Ellipses.java

Compile: javac Ellipses.java

Run: appletviewer Ellipses.java

Drawing Arcs

An arc is a part of oval. Arcs can be drawn with drawArc() and fillArc() methods.

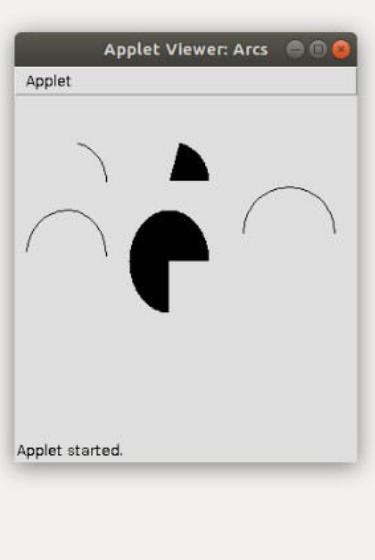
Syntax

```
void drawArc(int top, int left, int width, int height, int startAngle, int sweepAngle)  
void fillArc(int top, int left, int width, int height, int startAngle, int sweepAngle)
```

The arc is bounded by the rectangle whose upper-left corner is specified by (top, left) and whose width and height are specified by width and height. The arc is drawn from startAngle through the angular distance specified by sweepAngle. Angles are specified in degree. '0°' is on the horizontal, at the 30' clock's position. The arc is drawn counterclockwise if sweep Angle is positive, and clockwise if sweepAngle is negative.

The following applet draws several arcs:

```
import java.awt.*;  
import java.applet.*;  
/*  
<applet code="Arcs" width=300 Height=300>  
</applet>  
*/  
public class Arcs extends Applet  
{  
    public void paint(Graphics g)  
    {  
        g.drawArc(10,40,70,70,0,75);  
        g.fillArc(100,40,70,70,0,75);  
        g.drawArc(10,100,70,80,0,175);  
        g.fillArc(100,100,70,90,0,270);  
        g.drawArc(200,80,80,80,0,180);  
    }  
}
```



SavaAs: Arcs.java

Compile: javac Arcs.java

Run: appletviewer Arcs.java

Drawing Polygons

Polygons are shapes with many sides. It may be considered a set of lines connected together. The end of the first line is the beginning of the second line, the end of the second line is the beginning of the third line, and so on. Use drawPolygon() and fillPolygon() to draw arbitrarily shaped figures.

Syntax

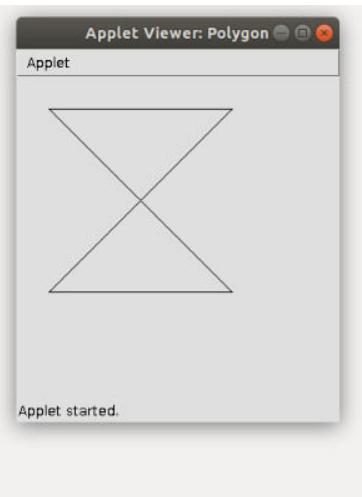
```
void drawPolygon(int x[], int y[], int numPointer)  
void fillPolygon(int x[], int y[], int numPointer)
```

The polygon's endpoints are specified by the coordinate pairs contained within the x and y arrays. The number of points defined by x and y is specified by numPoints.

It is obvious that x and y arrays should be of the same size and we must repeat the first point at the end of the array for closing the polygon.

The following applet draws an hourglass shape:

```
import java.awt.*;  
import java.applet.*;  
/*  
<applet code="Polygon" width=300 Height=300>  
</applet>  
*/  
public class Polygon extends Applet  
{  
    public void paint(Graphics g)  
    {  
        int xpoints[]={30,200,30,200,30};  
        int ypoints[]={30,30,200,200,30};  
        int num=5;  
        g.drawPolygon(xpoints,ypoints,num);  
    }  
}
```



SaveAs: **Polygon.java**

Compile: **javac Polygon.java**

Run: **appletviewer Polygon.java**

*****SWING*****

```
import java.awt.*;
import javax.swing.*;

class MyFrame extends JFrame
{
    JLabel l1,l2,l3,l4,l5,l6,l7;
    JTextField tf;
    JPasswordField pf;
    JCheckBox cb1,cb2,cb3;
    JRadioButton rb1,rb2;
    JList l;
    JComboBox cb;
    JTextArea ta;
    JButton b;
    Container c;

    MyFrame()
    {
        this.setVisible(true);
        this.setSize(150,500);
        this.setTitle("SWING GUI COMPONENTS EXAMPLE");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        c=this.getContentPane();
        c.setLayout(new FlowLayout());
        c.setBackground(Color.green);

        l1=new JLabel("User Name");
        l2= new JLabel("password");
        l3= new JLabel("Qualification");
        l4= new JLabel("User Gender");
        l5= new JLabel("Technologies");
        l6= new JLabel("UserAddress");
        l7= new JLabel("comments");

        tf=new JTextField(15);
        tf.setToolTipText("TextField");
        pf=new JPasswordField(15);
        pf.setToolTipText("PasswordField");

        cb1=new JCheckBox("BSC",false);
        cb2=new JCheckBox("MCA",false);
        cb3=new JCheckBox("PHD",false);
        rb1=new JRadioButton("Male",false);
        rb2=new JRadioButton("Female",false);
```

```
ButtonGroup bg=new ButtonGroup();
bg.add(rb1);
bg.add(rb2);

String[] listitems={"cpp","c","java"};
l=new JList(listitems);

String[] cbitems={"hyd","pune","bangalore"};
cb=new JComboBox(cbitems);

ta=new JTextArea(5,20);

b=new JButton("submit");

c.add(l1);
c.add(tf);
c.add(l2);
c.add(pf);
c.add(l3);
c.add(cb1);
c.add(cb2);
c.add(cb3);
c.add(l4);
c.add(rb1);
c.add(rb2);
c.add(l5);
c.add(l);
c.add(l6);
c.add(cb);
c.add(l7);
c.add(ta);
c.add(b);
}

}

class SwingDemo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
}
```

*****JCOLORCHOOSEN*****

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
class MyFrame extends JFrame implements ChangeListener
{
    JColorChooser cc;
    Container c;
    MyFrame()
    {
        this.setVisible(true);
        this.setSize(500,500);
        this.setTitle("SWING GUI COMPONENTS EXAMPLE");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        c=getContentPane();
        cc=new JColorChooser();
        cc.getSelectionModel().addChangeListener(this);
        c.add(cc);
    }
    public void stateChanged(ChangeEvent c)
    {
        Color color=cc.getColor();
        JFrame f=new JFrame();
        f.setSize(400,400);
        f.setVisible(true);
        f.getContentPane().setBackground(color);
    }
}
class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
}
```

*****JFILECHOOSER*****

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
class MyFrame extends JFrame implements ActionListener
{
    JFileChooser fc;
    Container c;
    JLabel l;
    JTextField tf;
    JButton b;

    MyFrame()
    {
        this.setVisible(true);
        this.setSize(500,500);
        this.setTitle("SWING GUI COMPONENTS EXAMPLE");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        c=ContentPane();

        l=new JLabel("Select File:");
        tf=new JTextField(25);
        b=new JButton("BROWSE");
        this.setLayout(new FlowLayout());
        b.addActionListener(this);
        c.add(l);
        c.add(tf);
        c.add(b);
    }
    public void actionPerformed(ActionEvent ae)
    {
        class FileChooserDemo extends JFrame implements ActionListener
        {
            FileChooserDemo()
            {
                Container c=ContentPane();
                this.setVisible(true);
                this.setSize(500,500);

                fc=new JFileChooser();
                fc.addActionListener(this);
                fc.setLayout(new FlowLayout());
                c.add(fc);
            }
        }
    }
}
```

```

        public void actionPerformed(ActionEvent ae)
        {
            File f=fc.getSelectedFile();
            String path=f.getAbsolutePath();
            tf.setText(path);
            this.setVisible(false);
        }
    }
    new FileChooserDemo();
}
}

class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
}
-----
```

*****JTABLE*****

```

import javax.swing.*;
import java.awt.*;
import javax.swing.table.*;
class Demo1
{
    public static void main(String[] args)
    {
        JFrame f=new JFrame();
        f.setVisible(true);
        f.setSize(300,300);
        Container c=f.getContentPane();

        String[] header={"ENO","ENAME","ESAL"};
        Object[][] body={{"111","aaa",5000}, {"222","bbb",6000}, {"333","ccc",7000}, {"444","ddd",8000}};

        JTable t=new JTable(body,header);
        JTableHeader th=t.getTableHeader();
        c.setLayout(new BorderLayout());
        c.add("North",th);
        c.add("Center",t);
    }
}
```