

*Last modification date :- August 23, 2016 [new version don't prefer old copy]*

*ratna5256@gmail.com*



# *Corejava*

**By**

***Mr. Ratan***



## Core java syllabus

### **1. Introduction 6-26**

*Basics of java  
Parts of java  
Keywords of java  
Features of java  
Coding conventions  
Identifier  
First application  
Data types in java*

### **2. Flow control statements 25-40**

*If,if-else,else-if,switch  
For,while,do-while  
Break,continue*

### **3. Java class 41-80**

*Variables  
Methods  
Constructors  
Instance blocks  
Static blocks*

### **4. Oops 81-127**

*Class  
Object  
Inheritance  
Polymorphism  
Abstraction  
Encapsulation*

### **5. Packages 128-139**

*Predefined packages  
User defined packages  
Importing packages  
Project modules  
Source file declaration*

### **6. Modifiers**

*Public , private , protected  
,abstract,final,static,native,strictfp,volatile,transient,synchronized, (11 modifiers )*

### **7. Interface 140- 152**

*Interface declarations  
Marker interface*

*Adaptor classes  
Interface vs. inheritance*

**8. Garbage Collector**  
**9. String manipulations 153-164**

*String  
StringBuffer  
StringBuilder  
 StringTokenizer*

**10. Wrapper class 165-170**

*Data types vs Wrapper classes  
All 8 wrapper classes explanations  
Auto boxing vs Autounboxing  
All possible conversions  
toString(), parseXXX(), valueOf(), XXXValue().*

**11. java.io package 171-182**

*introduction  
character Oriented Streams  
Byte oriented stream  
Writing and reading operations on file.  
Normal streams vs Buffered streams.  
File class  
Serialization  
Deserialization*

**12. Exception handling 183-216**

*Types of Exceptions  
Exception vs Error  
Try-catch blocks usage  
Try with resources  
Exception propagation  
Finally block usage  
Throws keyword usage  
Exception handling vs method overriding.  
Throw keyword usage  
Customization of exception handling  
Different types of Exceptions and error*

**13. Multithreading 217-234**

*Thread info  
Single Threaded model vs multithreaded model  
Main Thread vs user Thread  
Creation of user defined Thread  
Life cycle stages of Thread  
Thread naming*

*Thread priority  
Thread synchronization  
Inter Thread communication  
Hook Thread  
Daemon Thread  
Difference between wait() notify() notifyAll()*

**14. Nested classes 235-245**

*Introduction  
Advantages of nested classes  
Nested classes vs inner classes  
Normal Inner classes  
Method local inner classes  
Anonymous inner classes  
Static nested classes*

**15. Functional interfaces & lambda expressions 246-250****16. Annotations 251 - 258**

*Introduction  
Advantages of annotations  
Different annotations working*

**17. Enumeration 259 - 266**

*Introduction  
Advantages of enumeration  
Enum vs enum  
Difference between enum vs class*

**18. Arrays 267-272**

*Introduction  
Declaration of Arrays  
Arrays storing Object data & primitive data.*

**19. Collection framework 273-328**

*Introduction about Arrays  
Advantages of collection over arrays  
Collection vs Collections  
Key interfaces of Collections  
Characteristics of Collection framework classes  
Information about cursors  
Introduction about Map interface  
List interface implementation classes  
Set interface implementation classes  
Map interface implementation classes  
Comparable vs comparator  
Sorting mechanisms of Collection objects*

**20. Generics.**

*Type safety.*

**21. Networking 329-332**

*Introduction  
Socket and ServerSocket  
URL info  
Client-Server programming*

**22. AWT(Abstract Window Tool Kit) 333-357**

*Introduction  
Frame class  
Different layouts  
Components of AWT(TextField, RadioButton, Checkbox....etc)  
Event Handling or Event delegation Model  
Different types of Listeners*

**23. Swings 358 to 366**

*Difference between Awt and swings  
Advantages of swings  
Different components of Swings(TextField, RadioButton, Checkbox....etc)  
Event handling in Swings*

**24. Applet in java 367-370**

**25. INTERNATIONALIZATION (I18N) 371-380**

*Design application to support dif country languages  
Local class  
ResourceBundle  
Date in different formats  
Info about properties file*

**26. JVM architecture 381-382**

*What is JVM  
Structure of the JVM  
Components of JVM*

**27. Interview Questions 383-400**

**28. Regular Expressions.**

**29. Assertions**

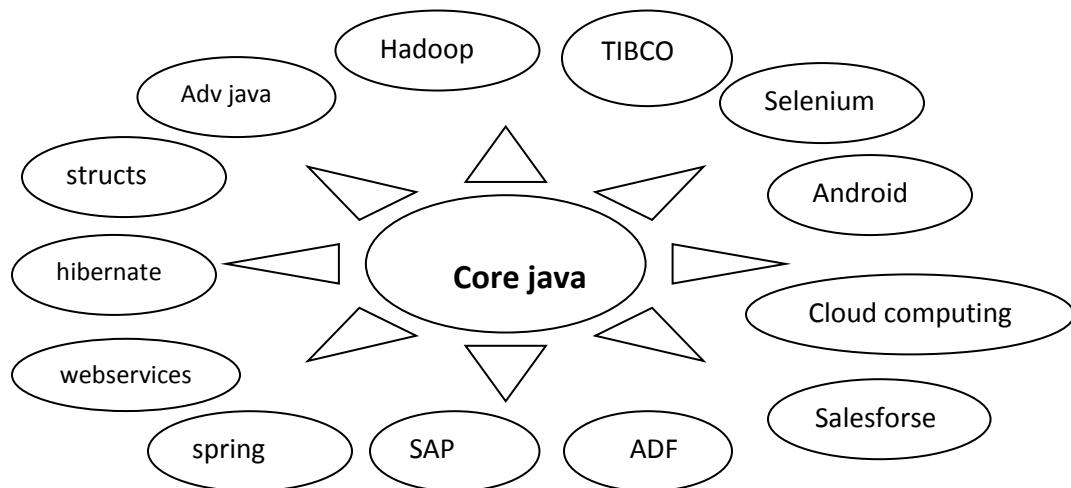
**JAVA introduction:-**

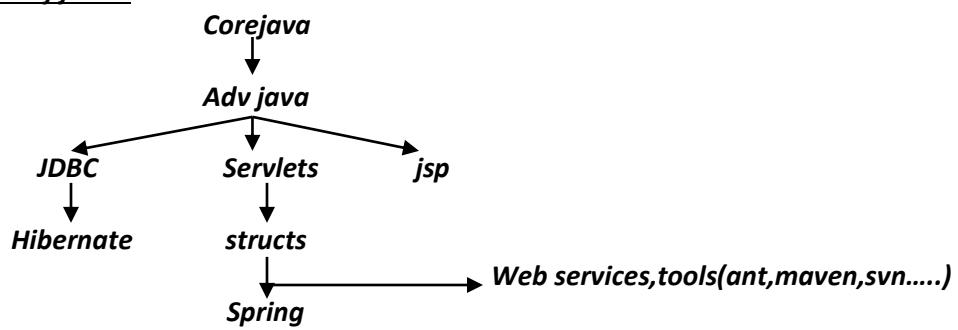
Author	:	<b>James Gosling</b>
Vendor	:	<b>Sun Micro System(which has since merged into Oracle Corporation)</b>
Project name	:	<b>Green Project</b>
Type	:	<b>open source &amp; free software</b>
Initial Name	:	<i>OAK language</i>
Present Name	:	<i>java</i>
Extensions	:	<i>.java &amp; .class &amp; .jar</i>
Initial version	:	<i>jdk 1.0 (java development kit)</i>
Present version	:	<b>java 8 2014</b>
Operating System	:	<i>multi Operating System</i>
Implementation Lang	:	<i>c, cpp.....</i>
Symbol	:	<i>coffee cup with saucer</i>
Objective	:	<i>To develop web applications</i>
SUN	:	<b>Stanford Universally Network</b>
Slogan/Motto	:	<i>WORA(write once run anywhere)</i>

**Importance of core java:-**

According to the SUN 3 billion devices run on the java language only.

- 1) Java is used to develop Desktop Applications such as MediaPlayer, Antivirus etc.
- 2) Java is Used to Develop Web Applications such as savyajobs.com, irctc.co.in etc.
- 3) Java is Used to Develop Enterprise Application such as Banking applications.
- 4) Java is Used to Develop Mobile Applications.
- 5) Java is Used to Develop Embedded System.
- 6) Java is Used to Develop SmartCards.
- 7) Java is Used to Develop Robotics.
- 8) Java is used to Develop Games ..... **etc.**

**Technologies Depends on Core java:-**

**Learning process of java:-****Parts of the java:-**

As per the **sun micro system** standard the java language is divided into three parts

- 1) J2SE/JSE(java 2 standard edition)
- 2) J2EE/JEE(java 2 enterprise edition)
- 3) J2ME/JME(java 2 micro edition)

**Java keywords:-(50)****Data Types**

byte  
short  
int  
long  
float  
double  
char  
boolean  
**(8)**

**Flow-Control:-**

if  
else  
switch  
case  
default  
break  
for  
while  
do  
continue  
**(10)**

**Predefined constants****method-level:-**

void  
return  
**(2)**  
  
**Object-level:-**  
new  
this  
super  
instanceof  
**(4)**

**source-file:**  
class  
extends  
interface  
implements  
package  
import  
**(6)**

True, false, null (3)

**Exception handling:-**

try  
catch  
finally  
throw  
throws  
**(5)**

**1.5 version:-**

enum  
assert  
**(2)**

**unused:-**

goto  
const  
**(2)**

**Modifiers:-**

public  
private  
protected  
abstract  
final  
static  
strictfp  
native  
transient  
volatile  
synchronized  
**(11)**

**Differences between C & CPP & JAVA:-****C-lang**

```
#include<stdio.h>
Void main()
{  Printf("hi ratan");
}
```

Author: **Dennis Ritchie**

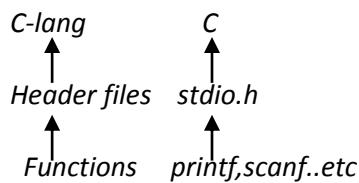
*Implementation languages:*  
COBOL,FORTRAN,BCPL, B...

*In c-lang the predefined support is available in the form of header files.*

**Ex:- stdio.h , conio.h**

*The header files contain predefined functions.*

**Ex:- printf,scanf.....**



*In above first example we are using **printf** predefined function that is present in stdio.h header file hence must include that header file by using #include statement.  
**Ex:#include<stdio.h>***

*In C lang program execution starts from main method called by **Operating system**.*

*To print data use **printf()***

**Cpp-lang**

```
#include<iostream.h>
Void main()
{  Cout<<"hello ratan";
}
```

Author : **Bjarne Stroustrup**

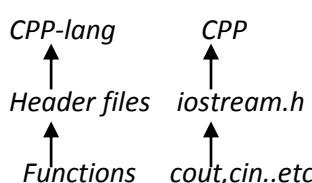
*implementation languages:*  
c ,ada,ALGOL68.....

*cpp language the predefined is maintained in the form of header files.*

**Ex:- iostream.h**

*The header files contains predefined functions.*

**Ex:- cout,cin....**



*In above first example we are using **cout** predefined function that is present in stdio.h header file hence must include that header file by using #include statement.  
**Ex:#include<stdio.h>***

*In C lang program execution starts from main method called by **Operating system**.*

*To print data use **cout***

**Java -lang**

```
Import java.lang.System;
Import java.lang.String;
Class Test
{
  Public static void main (String [] args)
  {
    System.out.println ("hi java");
  }
}
```

Author : **James Gosling**

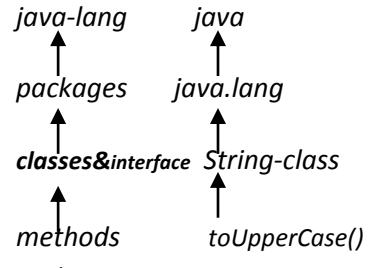
*implementation languages*  
C,CPP,ObjectiveC...

*In java predefined support is available in the form of packages.*

**Ex: java.lang, java.io,java.awt**

*The packages contains predefined classes&interfaces and these class & interfaces contains predefined methods.*

**Ex:- String,System**



*in above exampe we are using two classes(String,System) these classes are present in **java.lang** package must import it by using import keyword.*

**a) Import java.lang.\*;** all lasses  
**b) Import java.lang.System;** required  
**Import java.lang.String;** classes  
*In above two approchaeas 2<sup>nd</sup> good*

*In java execution starts from main called by JVM  
To print data use **System.out.println()***

<b>Version Name</b>	<b>Code Name</b>	<b>Release Date</b>
JDK 1.0	Oak	23 January 1996
JDK 1.1	(none)	19 February 1997
J2SE 1.2	Playground	4 December 1998
J2SE 1.3	Kestrel	8 May 2000
J2SE 1.4	Merlin	13 February 2002
J2SE 5.0	Tiger	29 September 2004
Java SE 6	Mustang	11 December 2006
Java SE 7	Dolphins	28 July 2011
Java SE 8	(Not available)	18 March 2014

**JAVA Features :-(buzz words)**

- 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.
- ✓ Java uses c, cpp syntaxes mainly hence who knows C, CPP for that java is simple language.

**2. Object Oriented:-**

- Java is object oriented technology because it is representing total data of the class in the form of object.
- The languages which are support object, class, Inheritance, Polymorphism, Encapsulation, Abstraction those languages are called Object oriented.

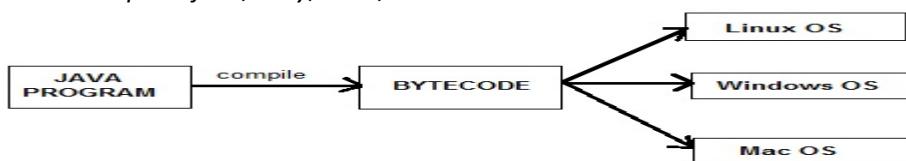
**3. Platform Independent :-**

When we compile the application by using one operating system (windows) that Compiled file can execute only on the same operating system(windows) this behavior is called platform dependency.

Example :- C, CPP ...etc

When we compile the application by using one operating system (windows) that Compiled file can execute in all operating systems(Windows, Linux, Mac...etc) this behavior is called platform independency.

Example :- java, Ruby, Scala, PHP ...etc



**4. Architectural Neutral:-**

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

**5. Portable:-**

In Java 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 good at two main areas that technology is robust technology.*

- a. Exception Handling
- b. Memory Allocation

*Java is providing predefined support to handle the exceptions.*

*Java provides Garbage collector to support memory management.*

**7. Secure:-**

- To provide implicit security Java provides 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).*

**9. Distributed:-**

*By using java it is possible to develop distributed applications & to develop distributed applications java uses RMI,EJB...etc*

**10. Multithreaded:-**

- Thread is a light weight process and a small task in large program.
- In java it is possible to create user thread & it possible to execute simultaneously is called multithreading.
- The main advantage of multithreading is it shares the same memory & threads are important at multimedia, gaming, web application.

**11. High Performance:-**

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

**Types of java applications:-****1. Standalone applications:**

- ✓ It is also known as window based applications or desktop applications.
- ✓ This type of applications must install in every machine like media player, antivirus ...etc
- ✓ By using AWT & Swings we are developing these type of applications.
- ✓ This type of application does not required client-server architecture.

## **2. Web applications:**

- a. The applications which are executed at server side those applications are called web applications like Gmail, facebook ,yahoo...etc .
- b. All applications present in internet those are called web-applications.
- c. The web applications required client-server architecture.
  - i. Client : who sends the request.
  - ii. Server : it contains application & it process the app & it will generate response.
  - iii. Database : used to store the data.
- d. To develop the web applications we are using servlets ,structs ,spring...etc

## **3. Enterprise applications:-**

- It is a business application & most of the people use the term it big business application.
- Enterprise applications are used to satisfy the needs of an organization rather than individual users. Such organizations are business, schools, government ...etc
- An application designed for corporate use is called enterprise application.
- An application in distributed in nature such as banking applications.
- All j2ee & EJB is used to create enterprise application.

## **4. Mobile applications:-**

- ✓ The applications which are design for mobile platform are called mobile applications.
- ✓ We are developing mobile applications by sing android,IOS,j2me...etc
- ✓ There are three types of mobile applications
  - Web-application (gmail ,online shopping,oracle ...etc)
  - Native (run on device without internet or browser) ex:phonecall,calculator,alarm,games  
These are install from application store& to run these apps internet not required.
  - Hybrid (required internet data to launch) ex:whats up,facebook,LinkedIn...etc  
These are installed form app store but to run this application internet data required.

## **5. Distributed applications:-**

Software that executes on two or more computers in a network. In a client-server environment.  
Application logic is divided into components according to function.

Ex : aircraft control systems ,industrial control systems, network applications...etc

### Types of software :

The set of instructions that makes the computer system do something.

#### 1) Application software

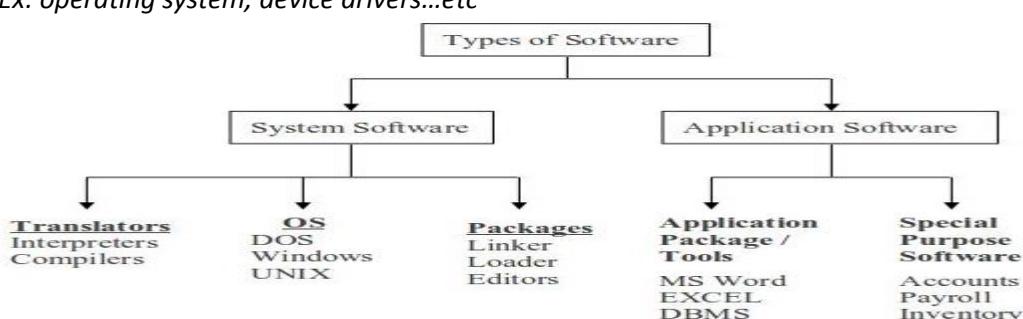
The program that allows the user to perform particular task.

Ex: business apps, entertainment apps,ms-office...etc

#### 2) System software

The programs that allow the hardware to run properly.

Ex: operating system, device drivers...etc



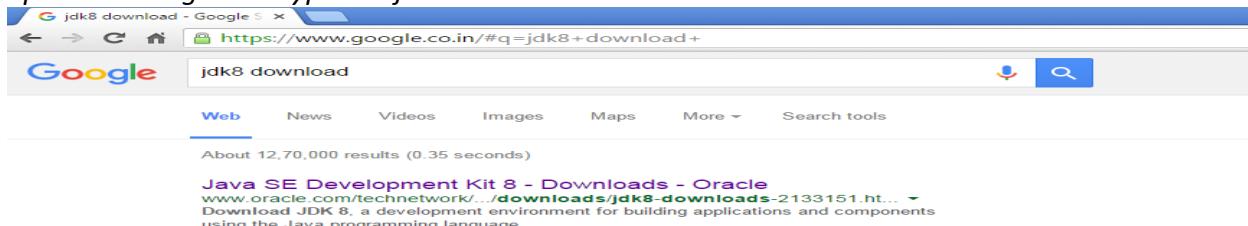
### Install the software and set the path :-

- 1) Download the software.
- 2) Install the java software in your machine.
- 3) Set the environmental variable.

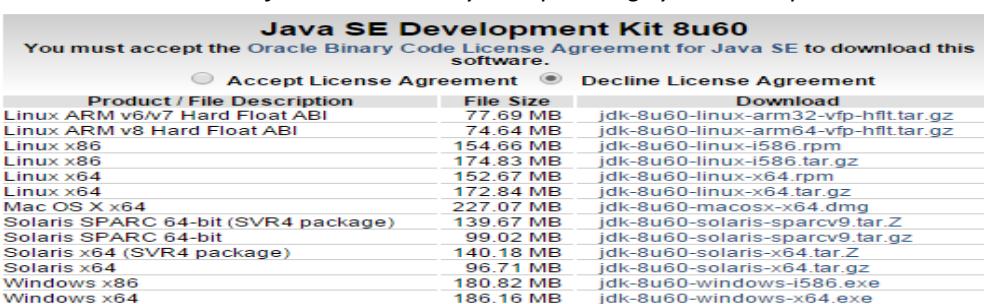
### Download the software:-

- ✓ Download the software from internet based on your operating system & processor because the software is different from operating system to operating system & processor to processor.

Open the Google site type the jdk8 download as shown below.



After clicking above link we will get below window then accept license agreement by clicking radio button then choose the software based on your operating system and processor to download.

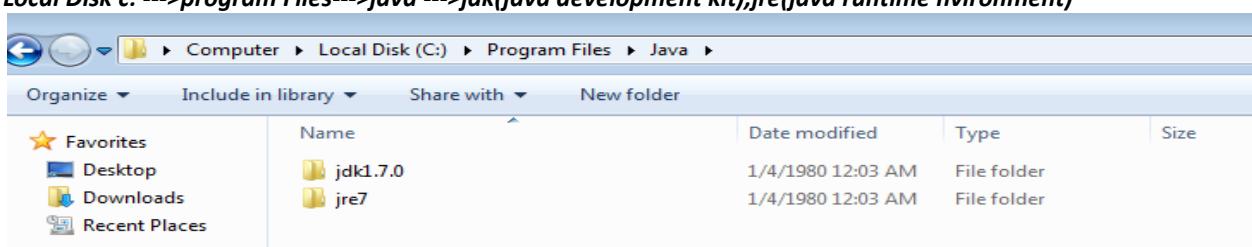


For 32-bit operating system please click on Windows x86

For 64-bit operating system please click on Windowsx64

### Install the java software in your machine:-

- ✓ Install the java software just like media players in your machine.
  - ✓ After installing the software the installation java folder is available in the following location by default.(but it possible to change the location at the time of installation).
- Local Disk c: --->program Files--->java --->jdk(java development kit),jre(java runtime environment)



After installing To check whether the java is installed in your system or not open the command prompt type javac command.

Process to open command prompt: Start --->run---->open:cmd---->ok

C:\Users\RATAN>javac

```
C:\Users\RATAN>javac
'javac' is not recognized as an internal or external command,
operable program or batch file.
```

Whenever we are getting above information then decide in our system java is installed but the java is not working.

#### **Why java is not working Reason:-**

**C:\Users\RATAN>javac**

Whenever we are typing javac command on the command prompt operating system will pickup javac command search for that command,

- in the internal operating system calls but javac is not available in the internal system calls list.
- If it not available in internal system calls list then immediately it won't raise any error, it will search in environmental variables

In above two cases if the javac command is not available then operating system will raise error message "javac is not recognized as an internal or external command"

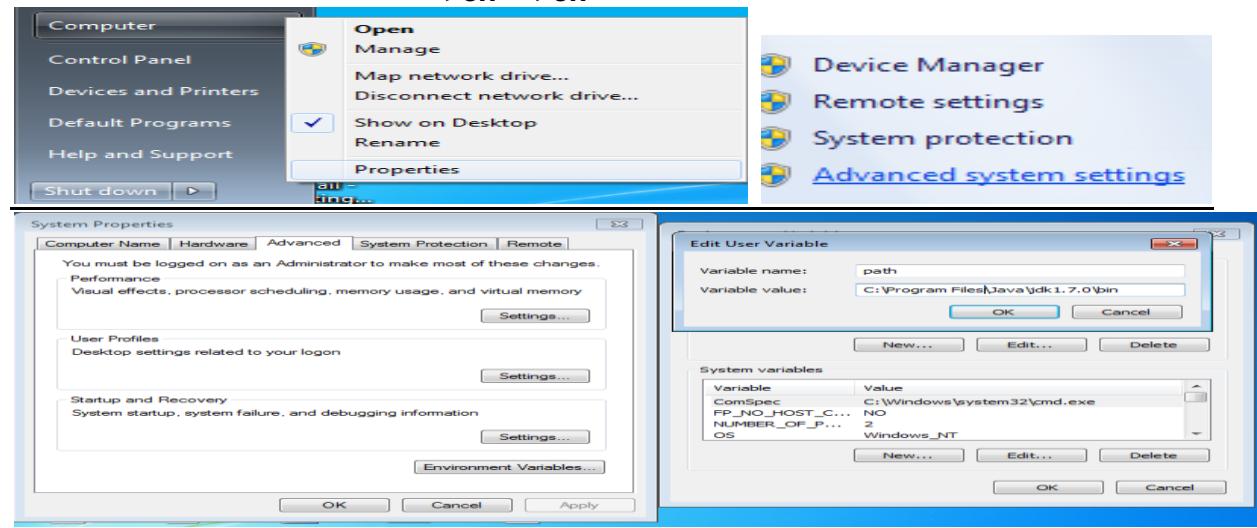
To overcome above problem to make eligible **javac** command operating system set environmental variables.

The location of javac command is : C:\Program Files\Java\jdk1.7.0\bin

Right click on mycomputer --->properties----->Advanced system setting--->Environment Variables --

User variables--->new---> variable name : path

Variable value : C:\programfiles\java\jdk1.6.0\_11\bin;  
---->ok---->ok



Now the java is working in your system to check open the **new** command prompt & type javac command then we will get list of commands then decide in your system java is working.

In your system or your friend system to check java is installed or not open the command prompt & type javac command

- If error message displayed java is not working.('javac' is not recognized as an internal or external command)
- If list of commands are displayed then decide java is working properly.

**Java coding conventions:-****Classes:-**

- ✓ Class name start with upper case letter and every inner word starts with upper case letter.
- ✓ This convention is also known as **camel case** convention.
- ✓ The class name should be nouns.

Ex:- **String**      **StringBuffer**      **InputStreamReader** .....etc

**Interfaces :-**

- ❖ Interface name starts with upper case and every inner word starts with upper case letter.
- ❖ This convention is also known as **camel case** convention.
- ❖ The class name should be nouns.

Ex: **Serializable**    **Cloneable**    **RandomAccess**

**Methods :-**

- ✓ Method name starts with lower case letter and every inner word starts with upper case letter.
- ✓ This convention is also known as mixed case convention
- ✓ Method name should be verbs.

Ex:- **post()**    **charAt()**    **toUpperCase()**    **compareToIgnoreCase()**

**Variables:-**

- ❖ Variable name starts with lower case letter and every inner word starts with upper case letter.
- ❖ This convention is also known as mixed case convention.

Ex :- **out**   **in**   **pageContext**

**Package :-**

- ✓ Package name is always must written in lower case letters.
- Ex :- **java.lang**    **java.util**    **java.io** ...etc

**Constants:-**

- ❖ While declaring constants all the words are uppercase letters .
- Ex: **MAX\_PRIORITY**    **MIN\_PRIORITY**    **NORM\_PRIORITY**

**NOTE:-** The coding standards are mandatory for predefined library & optional for user defined library but as a java developer it is recommended to follow the coding standards for user defined library also.

**Steps to Design a First Application:-**

- Step-1:- **Select an Editor.**
- Step-2:- **Write the application.**
- Step-3:- **Save the application.**
- Step-4:- **Compilation Process.**
- Step-5:- **Execution process.**

**Step1:- Select an Editor**

*Editor is a tool or software it will provide very good environment to develop java application.*

*Ex :- Notepad, Notepad++,edit Plus.....etc*

**IDE:- ( Integrated development Environment )**

*IDE is providing very good environment to develop the application.*

*Ex:- Eclipse,MyEclipse,Netbeans,JDeveloper....etc*

IDE is a real-time standard but don't use IDE to develop core java applications because 75% work is done by IDE & remaining 25 % work is down by developer.

**75% work of IDE is:-**

- ✓ Automatic compilation.
- ✓ Automatic package import.
- ✓ It shows all the predefined methods of classes.
- ✓ Automatically generate try catch blocks and throws (Exception handling)
- ✓ It is showing the information about how to fix the bug.....etc

**Note :- Do the practical's of core java only by using Edit Plus software.**

**Step 2:- Write a program.**

- Write the java program based on the java API (Application Programming Interface) rule and regulations .

***Open editplus --->file ---->new ----->click on java (it display sample java application)***

- Java is a case Sensitive Language so while writing the program you must take care about the case (Alphabet symbols).

**Example application:-**

```
import java.lang.System;
import java.lang.String;
class Test      //class declaration
{
    //class starts
    public static void main(String[] args)    //program execution starting point
    {
        //main starts
        System.out.println("hi Ratan"); //printing statement
    }
    //main ends
}
//class ends
class A
{
}
class B
{}
```

In above example **String & System** classes are present predefined java.lang package hence must import that package by using import statement.

There are two approaches to import the classes in java,

- 1) Import all class of particular package.
  - a. Import java.lang.\*; //it is importing all classes of java.lang package.
- 2) Import required classes
  - a. Import java.lang.System;
  - b. Import java.lang.String;

In above two approaches second approach is best approach because we are importing application required classes.

**Note:** The source file is allows declaring multiple java classes.

### Step3:- save the application.

- After writing the application must save the application by using (**.java**) extension.
- While saving the application must fallow two rules
  - If the source file contains public class then public class name & Source filename must be same (**publicClassName.java**). Otherwise compiler generate error message.
  - if the source file does not contain public class then save the source file with any name (**anyName.java**) like A.java , Ratan.java, Anu.java .....etc .

**Note:** - The source file allowed only one public class, if we are trying to declare multiple public classes then compiler generate error message.

#### example 1:- invalid

```
//Ratan.java
public class Test
{
};
class A
{
};
```

#### **Application location:-**

```
D:
|--rattan
   |--Sravya.java
```

#### example 2:- valid

```
//Test.java
public class Test
{
};
class A
{
};
```

(any disk)  
(any folder)  
(your file name)

#### example 3:- invalid

```
//Test.java
public class Test
{
};
public class A
{
};
```

**Step-4:- Compilation process.**

Compile the java application by using **javac** command.

**Syntax:-**      **Javac filename**  
**Javac Test.java**

***Process of moving application saving location:-***

**C:\Users\hp>**                    initial cursor location  
**C:\Users\hp>d:**                    move to local disk D  
**D:\>cd ratan**                    changing directory to ratan  
**D:\ratan>javac Sravya.java**    compilation process

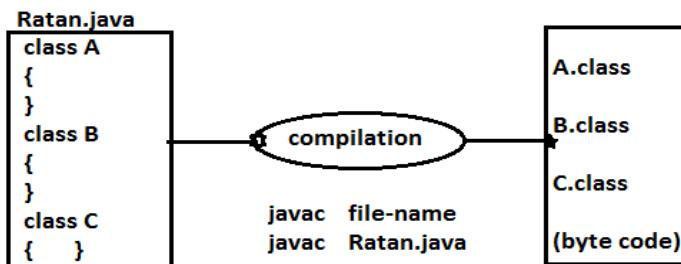
Whenever we are performing compilation the compiler will check the syntax errors.

- If the application contains syntax errors then compiler will generate error message in the form of compilation error.
- If the application does not contain syntax errors then compiler will generate .class files.(conversion of .java to .class)

**Note: - in java .class files generated by compiler at compilation time and .class file generation based on number of classes present in source file.**

If the source file contains 100 classes after compilation compiler generates 100 .class files

The compiler generates .class file and .class file contains byte code instructions it is intermediate code.

**Process of compiling multiple files:-**

**D:**

**/-->ratan**

**/-->Sravya.java**  
**/-->A.java**  
**/-->B.java**  
**/-->C.java**

**javac A.java**  
**javac B.java C.java**  
**javac \*.java**

**one file is compiled(A.java)**  
**two files are compiled**  
**all files are compiled**

**Step-5:- Execution process.**

Run /execute the java application by using **java** command.

**Syntax:-**      Java class-name  
                  Java Test

Whenever you are executing particular class file then JVM perform fallowing actions.

- JVM wills loads corresponding .class file byte code into memory.
- After loading .class file JVM calls main method to start the execution process.

In above two cases if the class file or main method is not available then at runtime JVM will generate error message.

If the main method is not available: “**Main method not found in class A, please define the main method**”.

If the .class is not available : “**Could not find main class**”.

**Executing all generated .class files based on example given in second step:-**

<b>Test class</b>	--->	class is loaded & main is present
<b>A class</b>	--->	class is loaded but main is not present
<b>B class</b>	--->	class is loaded but main is not present
<b>XXX class</b>	--->	XXX class is not present

D:\ratan>java Test

Hi Ratan

D:\ratan>java A

Error: Main method not found in class A, please define the main method as:

D:\ratan>java B

Error: Main method not found in class B, please define the main method as:

D:\ratan>java XXX

Error: Could not find or load main class XXX

**Note 1:-** compiler is translator it is translating .java file to .class where as JVM is also a translator it is translating .class file to machine code.

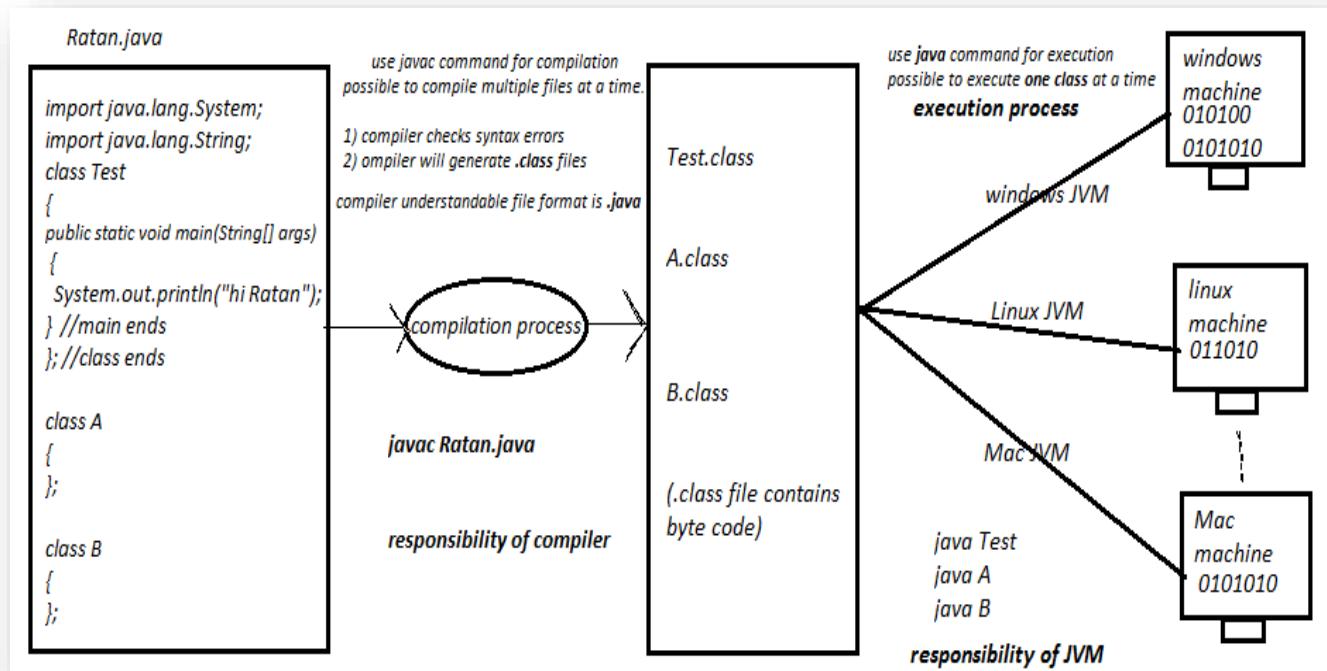
**Note 2:-** Compiler understandable file format is .java file but JVM understandable file format is .class file.

**Note 3:-** It is possible to compile multiple files at a time but it is possible to execute only one .class file at a time.

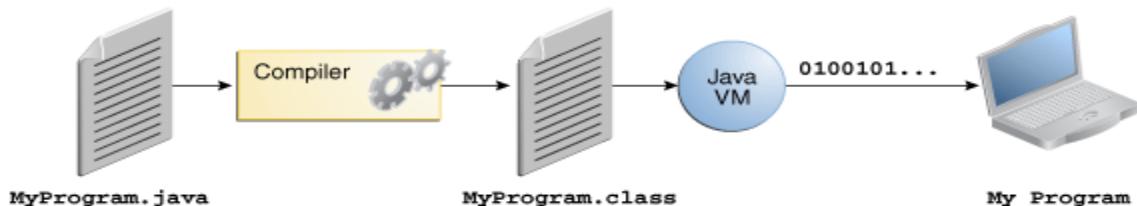
**Note 4:-** The .java file contains high level language (English) but .class file contains byte code instructions.

**Note 5:-** java is a platform independent language but JVM is platform dependent.

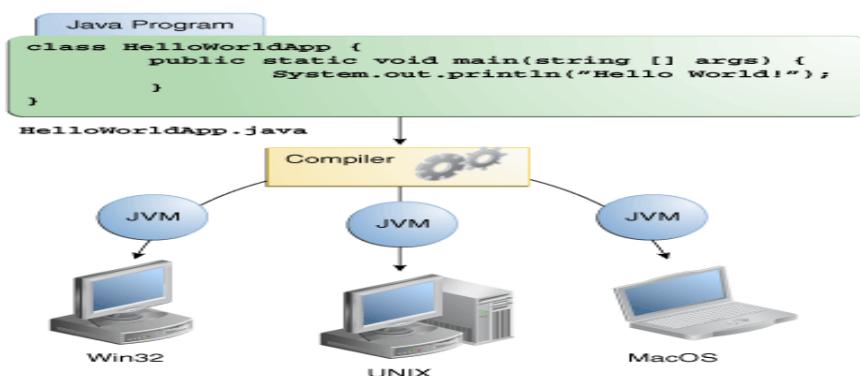
### Overview of first application



### Environment of the java programming development:-



### First program development :-



**Note: - java is a platform independent language but JVM is platform dependent.**

**Conclusion 1:-** Java contains 14 predefined packages but the default package in java is **java.lang**

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

**Conclusion 2:-**

The class contains main method is called **Main class** and java allows to declare multiple main class in a single source file.

```
class Test1
{
    public static void main(String[] args)
    {
        System.out.println("Test1 World!");
    }
}

class Test2
{
    public static void main(String[] args)
    {
        System.out.println("Test2 World!");
    }
}

class Test3
{
    public static void main(String[] args)
    {
        System.out.println("Test3 World!");
    }
}
```

D:\morn11>java Test1

Test1 World!

D:\morn11>java Test2

Test2 World!

D:\morn11>java Test3

Test3 World!

**Conclusion 3:-** The source file is allows to declare only one public class, if you are declaring morethan one public class compiler generate error message.

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

public class A{ }
```

**error: class A is public, should be declared in a file named A.java**

**conclusion-4** The below example compiled & executed but it is not recommanded because the class name starting with lower case letters.

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

G:\>java test
Ratan World!
```

**Class Elements:-**

```
Class Test{
    1. variables          int a = 10;
    2. methods            void add() {business logic }
    3. constructors       Test() {business logic }
    4. instance blocks   {business logic }
    5. static blocks      static {business logic }
}
```

**Java Comments:-**

- Comments are used to write the detailed description about application logics to understand the logics easily.
- Comments are very important in real time because today we are developing the application but that application maintained by some other person so to understand the logics by everyone writes the comments.
- Comments are non-executable code these are ignored at compile time.

There are 3 types of comments.

**1) Single line Comments:-**

By using single line comments it is possible to write the description about our programming logics within a single line & these comments are Starts with // (double slash) symbol.

Syntax:-    //description

**2) Multi line Comments:-**

This comment is used to provide description about our program in more than one line & these commands are start with /\* & ends with \*/

Syntax: -    /\*----statement-1  
              ---statement-2  
              \*/

**3) Documentation Comments:-**

By using documentation comments it possible to prepare API(Application programming interface) documents.(e will discuss later chapter)

Syntax: -    /\*  
              \*statement-1  
              \*statement-2  
              \*/

**Example:-**

```
/*project name:-green project
team size:- 6
team lead:- ratan
*/
class Test //class declaration
{
    //class starts
    public static void main(String[] args) // execution starting point
    {
        //main starts
        System.out.println("ratan"); //printing statement
    } //main ends
} //class ends
```

**Separtors in java:-**

<b>Symbol</b>	<b>name</b>	<b>usage</b>
( )	parentheses	used to contains list of parameters & contains expression.
{ }	braces	block of code for class, method, constructors & local scopes.
[ ]	brackets	used for array declaration.
;	semicolon	terminates statements.
,	comma	separate the variables declaration & chain statements in for.
.	period	used to separate package names from sub packages. And also used for separate a variable,method from a reference type.

**Print() vs Println ():-**

**Print():-** used to print the statement in console and the control is present in the same line.

**Example:-**      `System.out.print("Sravyainfotech");`  
                   `System.out.print("core java");`  
**Output:-** *Sravyainfotechcorejava*

**Println():-** used to print the statements in console but the control is there in next line.

**Example:-**      `System.out.println("Sravyainfotech");`  
                   `System.out.println("core java");`  
**Output:-** *Sravyainfotech*  
*Core java*

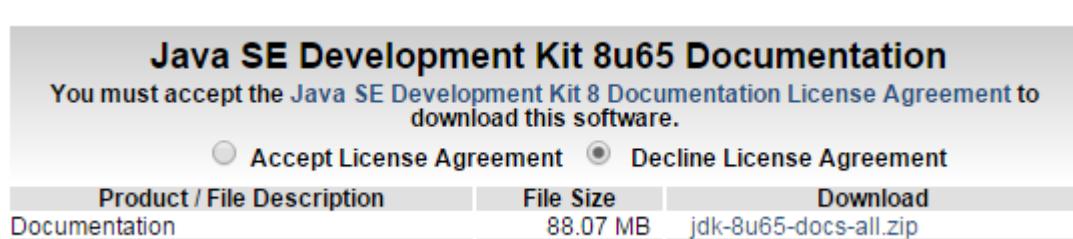
**Downloading Api document:-**

To download java api document use fallowing link

<http://docs.oracle.com/javase/8/docs/>



click on **JDK 8 documentation** then you will get below page.



**Java SE Development Kit 8u65 Documentation**  
 You must accept the Java SE Development Kit 8 Documentation License Agreement to download this software.  
 Accept License Agreement    Decline License Agreement  

Product / File Description	File Size	Download
Documentation	88.07 MB	<a href="#">jdk-8u65-docs-all.zip</a>

Accept the license agreement and download the file.

**Java Tokens:-**

- ✓ Smallest individual part of a java program is called Token.
- ✓ It is possible to provide any number of spaces in between two tokens.

**Example:-**

```

Class           Test
{
    Public      static     void      main
(      String[]   args   )
    {      int       a       =       10
        System . out      .
                    "java tokens");
    }
}  Tokens are ----- → class , test , { , } , [ .....etc

```

**Escape Sequences:-**

A character preceded by a backslash (\) is an escape sequence and has special meaning to the compiler.  
The following table shows the Java escape sequences

**Escape Sequences**

Escape Sequence	Description
\t	Insert a tab in the text at this point.
\b	Insert a backspace in the text at this point.
\n	Insert a newline in the text at this point.
\r	Insert a carriage return in the text at this point.
\f	Insert a formfeed in the text at this point.
\'	Insert a single quote character in the text at this point.
\"	Insert a double quote character in the text at this point.
\\\	Insert a backslash character in the text at this point.

```

class Test
{
    public static void main(String[] args)
    {
        System.out.println("Hello \t World");
        System.out.println("Hello\bWorld");
        System.out.println("Hello\nWorld");
        System.out.println("hi Hello\rWorld xxxx hi anu");
        System.out.println("Hello\fWorld\fhi anu");
        System.out.println("Hello\World\'ratan\"");
        System.out.println("Hello\World\"Anu\"");
        System.out.println("Hello\\World");
    }
}

```

**Java identifiers:-**

Every name in java is called identifier such as,

- ✓ Class-name
- ✓ Method-name
- ✓ Variable-name

**Rules to declare identifier:**

1. An identifier contains group of Uppercase & lower case characters, numbers ,underscore & dollar sign characters but not start with number.

int abc=10; ---> valid	int _abc=30; ---> valid	int \$abc=40; ---> valid
int a-bc=50; --->not valid	int 2abc=20; ---> Invalid	int not/ok=100 --->invalid

2. Java identifiers are case sensitive of course java is case sensitive programming language. The below three declarations are different & valid.

```
class Test
{
    int NUMBER=10;
    int number=10;
    int Number=10;
};
```

3. The identifier should not duplicated & below example is invalid because it contains duplicate variable name.

```
class Test
{
    int a=10;
    int a=20;
};
```

4. In the java applications it is possible to declare all the predefined class names & interfaces names as a identifier but it is not recommended to use.

```
class Test
{
    public static void main(String[] args)
    {
        int String=10;
        float Exception=10.2f;
        System.out.println(String);
        System.out.println(Exception);
    }
};
```

5. It is not possible to use keywords as a identifiers.

```
class Test
{
    int if=10;
    int try=20;
};
```

6. There is no length limit for identifiers but is never recommended to take lengthy names because it reduces readability of the code.

## ***Java primitive Data Types:-***

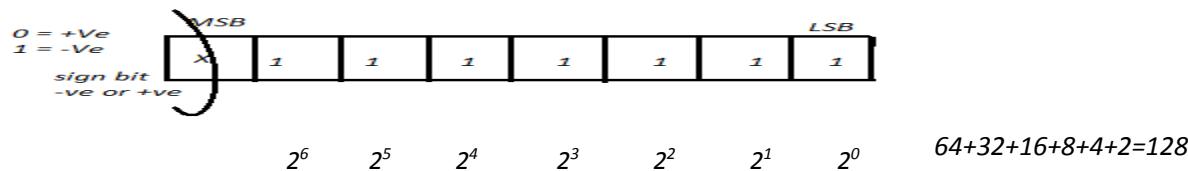
1. Data types are used to represent type of the variable & expressions.
  2. Representing how much memory is allocated for variable.
  3. Specifies range value of the variable.

*There are 8 primitive data types in java*

<u>Data Type</u>	<u>size(in bytes)</u>	<u>Range</u>	<u>default values</u>
<i>byte</i>	1	-128 to 127	0
<i>short</i>	2	-32768 to 32767	0
<i>int</i>	4	-2147483648 to 2147483647	0
<i>long</i>	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0
<i>float</i>	4	-3.4e38 to 3.4e	0.0
<i>double</i>	8	-1.7e308 to 1.7e308	0.0
<i>char</i>	2	0 to 6553	single space
<i>Boolean</i>	no-size	no-range	false

*Byte :-*

<i>Size</i>	:	<i>1-byte</i>
<i>MAX_VALUE</i>	:	<i>127</i>
<i>MIN_VALUE</i>	:	<i>-128</i>
<i>Range</i>	:	<i>-128 to 127</i>
<i>Formula</i>	:	$-2^n \text{ to } 2^{n-1}$
		$-2^8 \text{ to } 2^{8-1}$



**Note :-**

- To represent numeric values (10,20,30...etc) use **byte,short,int,long**.
  - To represent decimal values(floating point values 10.5,30.6...etc) use **float,double**.
  - To represent character use **char** and take the character within single quotes.
  - To represent true ,false use **Boolean**.

*Except Boolean and char remaining all data types consider as a signed data types because we can represent both +ve & -ve values.*

### **Float vs double:-**

*Float will give 5 to 6 decimal places of accuracy but double gives 14 to 15 places of accuracy.*

*Float will follow single precision but double will follow double precision.*

**Syntax:-** *data-type name-of-variable = value/literal;*

Ex- int a=10;

Int	-----→	Data Type
a	-----→	variable name
=	-----→	assignment
10	-----→	constant value
;	-----→	statement terminator

**printing variables :-**

```
int a=10;
System.out.println(a);      //valid
System.out.println("a");    //invalid
System.out.println('a');    //invalid
```

**User provided values are printed**

```
int a = 10;
System.out.println(a);//10
boolean b=true;
System.out.println(b);//true
char ch='a';
System.out.println(ch);//a
double d=10.5;
System.out.println(d);//10.5
```

**Default values(JVM assigned values)**

```
int a;
System.out.println(a);//0
boolean b;
System.out.println(b);//false
char ch;
System.out.println(ch);//single space
double d;
System.out.println(d)//0.0
```

**Example :-//Test.java**

```
class Test
{
    public static void main(String[] args)
    {
        float f=10.5;
        System.out.println(f);
        double d=20.5;
        System.out.println(d);
    }
}
```

D:\ratan>javac Test.java

Test.java:3: error: possible loss of precision

**float f=10.5;**

**required: float found: double**

in above example decimal value(10.5) by default double value hence compiler generating error message so to represent float value use f constant or perform type casting.

```
float f=10.5f;           //using f constant (valid)
float f =(float)10.5;    //using type casting (valid)
```

**variable declarations:**

int a=10;	----> integer variable
double d=10.5;	----> double variable
char ch='a';	----> char variable
boolean b=true;	----> boolean variable
float f=10.5f;	----> float variable
String str="ratan";	----> String variable

**Note: String is not a data type & it is a class present in java.lang package to represent group of characters or character array enclosed with in double quotes.**

## java flow control Statements

There are three types of flow control statements in java

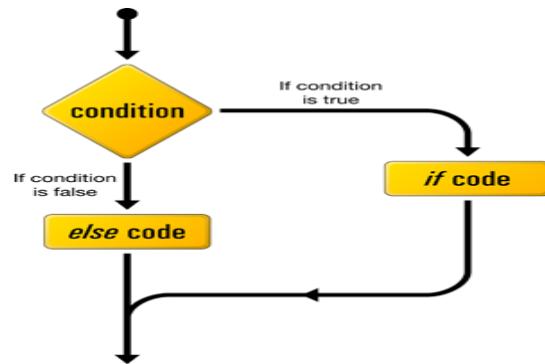
- Selection Statements
- Iteration statements
- Transfer statements

### 1. Selection Statements

- If
- If-else
- Switch
- else-if

#### If syntax:-

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



- ❖ If statmenet is taking condition that condition must be Boolean condition. Otherwise compiler will generate error message.

#### Example-1:- Normal example

```
class Test
{
    public static void main(String[] args)
    {
        int age=20;
        if (age>22)
        {
            System.out.println("if body / true body");
        }
        else
        {
            System.out.println("else body/false body ");
        }
    }
}
```

#### Example -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");
        }
    }
}
```

**Example-3:-in c-language 0-false & 1-true but these conventions 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");
        }
    }
}
```

**Example-4:**

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

*error: incompatible types: int cannot be converted to Boolean*

**Example-5:-** The curly braces are options but without curly braces it is possible to take only one statement that should not be a initialization.

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

*Case 1: valid  
If(true)  
SOP("hi ratan");*

*Case 2: valid  
If(true);*

*Note : In java semicolon is empty statement*

*Case 3: Invalid  
If(true)  
Int x=100;*

*Case 4: valid  
If(true)  
{Int x=100;}*

**Switch statement:-**

- ✓ Switch statement is used to declare multiple selections
- ✓ Switch is taking the argument, the allowed arguments are
  - **Byte,short,int,char (primitives)**
  - **Byte, Short, Integer, Character, enum (1.5 version)**
  - **String (1.7 version).**
- ✓ Inside the switch it is possible to declare more than one case but it is possible to declare only one default.
- ✓ Based on the provided argument the matched case will be executed if the cases are not matched default will be executed.
- ✓ While declaring switch statement braces are mandatory otherwise compiler will generate error message.

Note : Float and double and long is not allowed for a switch argument because these are having too large values.

**Syntax:-**

```
switch(argument)
{
    case label1:      statements;
                      break;
    case label2 :     statements;
                      break;
    /
    /
    default       :     statements;
                      break;
}
```

**Example-1: Normal input and normal output.**

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

**Example-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("anushka");
            break;
            case 10:System.out.println("nazriya");
            break;
            default:System.out.println("ubantu");
            break;
        }
    }
}
```

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

```
class Test
{
    public static void main(String[] args)
    {
        int a=99;
        switch (a+1)
        {
            case 10+20+70 :System.out.println("anushka");
            break;
            case 10+5 :System.out.println("nazriya");
            break;
            case 30/6 :System.out.println("samantha");
            break;
            default :System.out.println("ubantu");
            break;
        }
    }
}
```

**Example-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;
        switch (a)
        {
            case a:System.out.println("anushka"); break;
            case b:System.out.println("nazriya"); break;
            default:System.out.println("ubantu"); break;
        }
    }
}
```

**Example 5:** It is possible to declare final variables as a case label. Because the variables are replaced with constants during compilation.

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

**Example-6:- 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;
        }
    }
};
```

**Example 7:- 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;
        }
    }
};
```

**Example 8:- inside the switch both cases and default Is optional.**

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

**Example -9:- 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");
        }
    }
}
```

**Example-10:- internal conversion for unicode. 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;
            default: System.out.println("default"); break;
        }
    }
};
```

**Example -11: internal conversion of unicodes.**

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

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

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

**Example -13:-**

- ✓ Inside the switch statement break is optional.
- ✓ If we are not providing break statement then from the matched case onwards up to break statement will be executed, if there is no break statement then end of the switch will be 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;
        }
    }
};
```

**Example -14:** The advantage of fall through is used to execute common actions for multiple cases.

```
class Test
{
    public static void main(String[] args)
    {
        int a=2;
        switch (a)
        {
            case 1:
            case 2:
            case 3:System.out.println("Q-1");
            break;
            case 4:
            case 5:
            case 6:System.out.println("Q-2");
            break;
        }
    }
};
```

**Example-15:-** Inside the switch it is possible to declare the default statement at 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");
        }
    }
};
```

**Example -16:-** The below example compiled and executed in 1.7 version & above 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");
            case "bbb" : System.out.println("Hello");
            default     : System.out.println("what");
        }
    }
}
```

**Example -17:-** 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=126;
        switch (b)
        {
            case 126:System.out.println("20");
            case 127:System.out.println("30");
            case 128:System.out.println("40");
            default:System.out.println("default");
        }
    }
};
```

#### Else-if ladder:-

```
class Test
{
    public static void main(String[] args)
    {
        int a=20;
        if (a==10)
        {
            System.out.println("ten");
        }
        else if (a==20)
        {
            System.out.println("twenty");
        }
        else if (a==30)
        {
            System.out.println("thirty");
        }
        else
        {
            System.out.println("default condition");
        }
    }
};
```

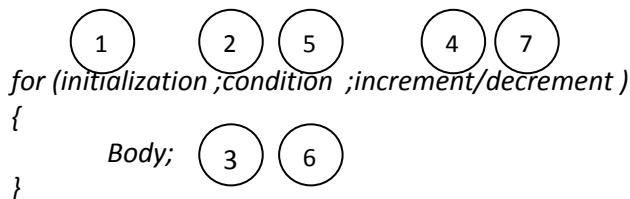
**Iteration Statements:-**

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

- 1) For
- 2) For-each
- 3) while
- 4) do-while

**for syntax:-**

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

**Flow of execution in for loop:-****With out for loop**

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

**By using for loop**

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

**Initialization part of for loop:-**

**Example- 1: 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("Rattaiah");
        }
    }
};
```

**Example-2:- In the initialization part it is possible to take any number of System.out.println("ratna") statements and each and every statement is separated by comma(,).**

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

**Example 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("Rattaiah");
        }
    }
}
```

**Ex :-declaring two variables are possible.**

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

#### Conditional part of for loop:-

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

```
for (int i=0; ;i++)
{
    System.out.println("Rattaiah");
}
```

#### Increment/decrement:-

**Example-1:- 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("Rattaiah");
        }
    }
}
```

**Example 2:- in the increment/decrement it is possible to take the any number of SOP() statements and each and every statement is separated by comma(,).**

```
for (int i=0;i<10;System.out.println("aruna"),System.out.println("sravya"))
{
    System.out.println("Rattaiah");
    i++;
}
```

**Note : Inside the for loop each and every part is optional.**

**for(;;)-----→represent infinite loop because the condition is always true.**

**Unreachable statement:-**

**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("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");
    }
}
```

**Note:- When you provide the condition even though that condition is represent infinite loop compiler is unable to find unreachable statements,(because that compiler is thinking that condition may fail).**

**When you provide Boolean value as a condition then compiler is identifying unreachable statement because compiler knows that condition never change.**

**While loop:-**

**Syntax:-**      **while (condition) //condition must be Boolean & mandatory.**  
**{**            **body;**  
**}**

**Example-1 :-**

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

**Example-2 :- compilation error unreachable statement.**

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

**Do-While:-**

- ✓ If we want to execute the loop body at least one time them we should go for do-while statement.
- ✓ In do-while first body will be executed then only condition will be checked.
- ✓ In the do-while the while must be ends with semicolon otherwise compilation error.

**Syntax:-**    **do**

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

***Example-1:-***

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

***Example-2 :- unreachable statement***

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

***Example-3 :-***

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

**For-each loop:-** (introduced in 1.5 version)

- ❖ For loop is used to print the data & it is possible to apply conditions.
- ❖ For-each loop is used to print the data but it is not possible to apply the conditions. To print the data starting element to ending element without conditions use for-each loop.

```
class Test
{
    public static void main(String[] args)
    {
        int[] a={10,20,30,40};
        //printing data by using for-loop
        for (int i=0;i<a.length;i++)
        {
            System.out.println(a[i]);
        }
        //printing data by using for-each loop
        for (int aa: a)
        {
            System.out.println(aa);
        }
    }
};
```

**Iterator vs Iterable:-**

- ✓ The target elements in for-each loop should be iterable. And it is interface present in java.lang package.
- ✓ If an object is iterable that class should implements Iterable interface.
- ✓ Iterator is cursor we will discussed in Collections framework.

**Transfer statements:-**

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

- **break**
- **continue**
- **return**
- **try**
- **goto**

**break:-**

Break is used to stop the execution. And is possible to use the break statement only two areas.

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

**Example-1 :- 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-2 :-** 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("ratan");
            break;
            System.out.println("nandu");
        }
    }
};
```

**Continue:** skip the current iteration and it is continue the rest of the iterations normally.

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

**Java Variables:-**

- ✓ Variables are used to store the constant values by using these values we are achieving project requirements.
- ✓ Variables are also known as **fields** of a class or **properties** of a class.
- ✓ All variables must have a type. You can use primitive types such as int, float, boolean, etc. Or array type or class type or enum type or interface type.
- ✓ Variable declaration is composed of three components in order,
  - Zero or more modifiers.
  - The variable type.
  - The variable name.

**Example : public final int x=100;**

```
public int a=10;
public ----> modifier (specify permission)
int       ----> data type (represent type of the variable)
a         ----> variable name
10        ----> constant value or literal;
;          ----> statement terminator
```

There are three types of variables in java

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

**Local variables:-**

- ❖ The variables which are declare inside a **method or constructor or blocks** those variables are called local variables.

```
class Test
{
    public static void main(String[] args)    //execution starts from main method
    {
        int a=10;                      //local variables
        int b=20;
        System.out.println(a);
        System.out.println(b);
    }
}
```

- ❖ It is possible to access local variables only inside the method or constructor or blocks only, it is not possible to access outside of method or constructor or blocks.

```
void add()
{
    int a=10;      //local variable
    System.out.println(a); //possible
}
void mul()
{
    System.out.println(a); //not-possible
}
```

- ❖ For the local variables memory allocated when method starts and memory released when method completed.
- ❖ The local variables are stored in stack memory.

**Instance variables (non-static variables):-**

- ✓ The variables which are declare inside a class but outside of methods those variables are called instance variables.
- ✓ The scope (permission) of instance variable is inside the class having global visibility.
- ✓ For the instance variables memory allocated during object creation & memory released when object is destroyed.
- ✓ Instance variables are stored in heap memory.

**Areas of java language:-**

There are two types areas in java.

- 1) Instance Area.
- 2) Static Area.

**Instance Area:-**

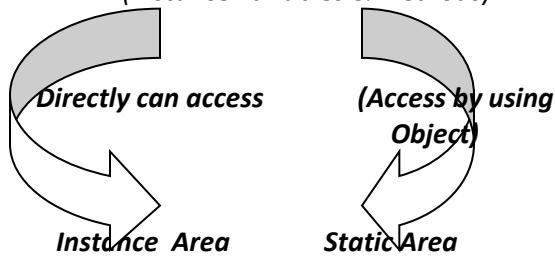
```
void m1()      //instance method
{   Logics here //instance area
}
```

**Static Area:-**

```
Static void m1() //static method
{   Logics here //static area
}
```

**Instance variable accessing:-**

(Instance variables & methods)

**Example:-**

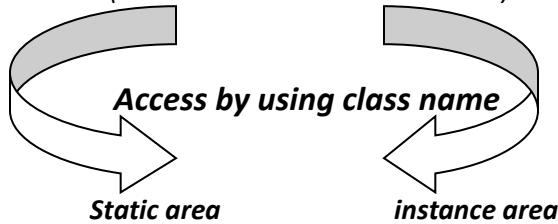
```
class Test
{
    //instance variables
    int a=10;
    int b=20;
    //static method
    public static void main(String[] args)
    {
        //Static Area
        Test t=new Test();
        System.out.println(t.a);
        System.out.println(t.b);
        t.m1(); //instance method calling
    }
    // instance method
    void m1() //user defined method must called by user inside main method
    {
        //instance area
        System.out.println(a);
        System.out.println(b);
    }
} //main ends
}//class ends
```

**Static variables (class variables):-**

- ❖ The variables which are declared inside the class but outside of the methods with static modifier those variables are called static variables.
- ❖ Scope of the static variables with in the class global visibility.
- ❖ Static variables memory allocated during .class file loading and memory released at .class file unloading time.
- ❖ Static variables are stored in non-heap memory.

**Static variables & methods accessing:-**

(Static variables&amp; static methods)



```
class Test
{
    //static variables
    static int a=1000;
    static int b=2000;
    public static void main(String[] args)      //static method
    {
        System.out.println(Test.a);
        System.out.println(Test.b);
        Test t = new Test();
        t.m1();           //instance method calling
    }
    //instance method
    void m1()          //user defined method called by user inside main method
    {
        System.out.println(Test.a);
        System.out.println(Test.b);
    }
};
```

**Static variables calling:-** We are able to access the static members inside the static area in three ways.

- ✓ Direct accessing.
- ✓ By using class name.
- ✓ By using reference variable.

In above three approaches second approach is best approach .

```
class Test
{
    static int x=100;           //static variable
    public static void main(String[] args)
    {
        System.out.println(a);           //1-way(directly possible)
        System.out.println(Test.a);      //2-way(By using class name)
        Test t=new Test();
        System.out.println(t.a);         //3-way(By using reference variable)
    }
};
```

**Example: - When we create object inside method that object is destroyed when method completed, if any other method required object then create the object inside that method.**

```
class Test
{
    //instance variable
    int a=10;
    int b=20;
    static void m1()
    {
        Test t = new Test();
        System.out.println(t.a);
        System.out.println(t.b);
    }
    static void m2()
    {
        Test t = new Test();
        System.out.println(t.a);
        System.out.println(t.b);
    }
    public static void main(String[] args)
    {
        Test.m1();      //static method calling
        Test.m2();      //static method calling
    }
};
```

**Example:-**

```
class Test
{
    int a=10;
    int b=20;      // instance variables
    static int c=30; static int d=40; //static variables
    void m1() //instance method
    {
        System.out.println(a);
        System.out.println(b);
        System.out.println(Test.c);
        System.out.println(Test.d);
    }
    static void m2() //static method
    {
        Test t = new Test();
        System.out.println(t.a);
        System.out.println(t.b);
        System.out.println(Test.c);
        System.out.println(Test.d);
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();          //instance method calling
        Test.m2();       //static method calling
    }
};
```

**Variables VS default values:-****Case 1:- for the instance variables JVM will assign default values.**

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

**Case 2:- for the static variables JVM will assign default values.**

```
class Test
{
    static int a;
    static float b;
    public static void main(String[] args)
    {
        System.out.println(Test.a);
        System.out.println(Test.b);
    }
};
```

**Case 3:-**

- For the instance and static variables JVM will assign default values but for the local variables the JVM won't provide default values.
- In java before using local variables must initialize some values to the variables otherwise compiler will raise compilation error "variable a might not have been initialized".

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

**D:\>javac Test.java****Test.java:6: variable a might not have been initialized****System.out.println(a);****Q) The below example valid or Invalid?**

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

**Class Vs Object:-**

- **Class is a logical entity it contains logics whereas object is physical entity it is representing memory.**
- **Class is blue print it decides object creation without class we are unable to create object.**
- **Based on single class (blue print) it is possible to create multiple objects but every object occupies memory.**
- **We are declaring the class by using class keyword but we are creating object by using new keyword.**
- **We will discuss object creation in detailed in constructor concept.**

**Instance vs. Static variables:-**

- ❖ **In case of instance variables the JVM will create separate memory for each and every object it means separate instance variable value for each and every object.**
- ❖ **In case of static variables irrespective of object creation per class single memory is allocated, here all objects of that class using single copy.**

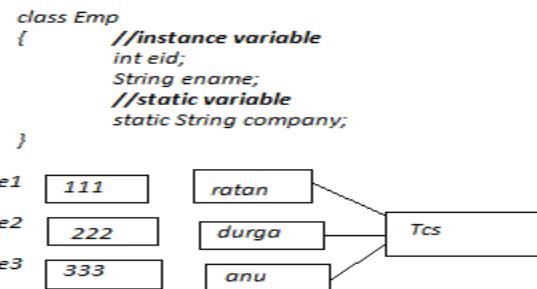
**Example :-**

```
class Test
{
    int a=10;          //instance variable
    static int b=20; //static variable
    public static void main(String[] args)
    {
        Test t = new Test();
        System.out.println(t.a); //10
        System.out.println(t.b); //20
        t.a=111;      t.b=222;
        System.out.println(t.a); //111
        System.out.println(t.b); //222
        Test t1 = new Test();   //10 222
        System.out.println(t1.a); //10
        System.out.println(t1.b); //222
        t1.b=444;
        Test t2 = new Test();   //10 444
        System.out.println(t2.b); //444
    }
}
```

**Instance variable vs static variable :-**

```
class Emp
{
    //instance variable
    int eid;
    String ename;
    String company;
}

e1 [111]      ratan      Tcs
e2 [222]      durga      Tcs
e3 [333]      anu        Tcs
```



**Different ways to initialize the variables :-**

```

class Test
{
    int s=10;
    int a,b,c;
    int x=10,y,z;
    int i=10,j=20,k;
    int p=10,q=20,r=30;
    public static void main(String[] args)
    {
        Test t = new Test();
        System.out.println(t.s);
        System.out.println(t.a+" "+t.b+" "+t.c);
        System.out.println(t.x+" "+t.y+" "+t.z);
        System.out.println(t.i+" "+t.j+" "+t.k);
        System.out.println(t.p+" "+t.q+" "+t.r);
    }
}

```

**Summary of variables:-**

<b><u>Characteristic</u></b>	<b><u>Local variable</u></b>	<b><u>instance variable</u></b>	<b><u>static variables</u></b>
<b>where declared</b>	inside method or Constructor or block.	inside the class outside Of methods	inside the class outside of methods .
<b>Usage</b>	within the method	inside the class.	inside the class all
<b>When memory allocated</b>	when method starts	when object created	when .class file loading
<b>When memory destroyed</b>	when method ends.	When object destroyed	when .class unloading.
<b>Initial values</b>	none, must initialize the value before first use.	default values are Assigned by JVM.	default values are Assigned by JVM.
<b>Relation with Object</b>	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.
<b>Accessing</b>	directly possible.	By using object name. <i>Test t = new Test();</i> <i>System.out.println(t.a);</i>	by using class name. <i>System.out.println(Test.a);</i>
<b>Memory</b>	stored in stack memory.	Stored in heap memory	method area.

### Java Methods (behaviors):-

- ❖ Inside the classes it is not possible to write the business logics directly hence inside the class declare the method inside that method writes the logics of the application.
- ❖ Methods are used to write the business logics of the project.
- ❖ **Coding convention:** method name starts with lower case letter and every inner word starts with uppercase letter(**mixed case**).

Example:- `post()` , `charAt()` , `toUpperCase()` , `compareToIgnoreCase()`.....etc

There are two types of methods in java,

1. Instance method
2. Static method

- ❖ Inside the class it is possible to declare any number of instance & 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-name (reference-variable).

### Method calling Syntax:-

```
Void m1() { logic here } //instance method
Objectnameinstancemethod(); //calling instance method
Test t = new Test();
t.m1();
```

### static method:-

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

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

### Method calling syntax:-

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

**Every method contains three parts.**

1. Method declaration
2. Method implementation (logic)
3. Method calling

Example:- void m1() -----> **method declaration**  
 { Body (Business logic); -----> **method implementation**  
 }  
 Test t = new Test(); t.m1(); -----> **method calling**

#### Method Syntax:-

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

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

Example:-      **Public void m1(){ logics...}**  
**Private int m2(int a,int b) { logics...}**

#### Method Signature:-

Method-name & parameters list is called method signature.

Syntax:-      **Method-name(parameter-list)**

Example:-      **m1(int a)**  
**m1(int a,int b)**

#### **Example-1 :- instance & static methods without arguments.**

- ✓ Instance methods are bounded with objects hence call the instance methods by using object name(reference variable).
- ✓ Static methods are bounded with class hence access the static methods by using class-name.

```
class Test
{
    void m1()
    {
        System.out.println("m1 instance method");
    }
    static void m2()
    {
        System.out.println("m2 static method");
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();           //calling of instance method by using object-name
        Test.m2();        //calling of static method by using class-name
    }
}
```

**Example-2:-instance & static methods with parameters.**

- If the method is expecting parameters (inputs to functionality) then while calling that method must pass the values to that parameters then only that method will be executed.
- While passing parameters, number of arguments & order of arguments important.

```

void m1(int a)                      -->t.m1(10);          -->valid
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,'a',10.6f); -->valid
void m5(int a,char ch,float f)      -->t.m3(10,'c');       -->invalid
class Test
{
    void m1(int a,char ch) //local variables
    {
        System.out.println("m1 instance method");
        System.out.println(a);
        System.out.println(ch);
    }
    static void m2(boolean b,double d)
    {
        System.out.println("m2 static method");
        System.out.println(b);
        System.out.println(d);
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1(10,'a');           //calling of instance method by passing inputs
        Test.m2(true,10.5);     //calling of static method by passing inputs
    }
}

```

**Example-3 :- while calling methods it is possible to provide variables as argument values.**

```

class Test
{
    void m1(int a,char ch,boolean b)
    {
        System.out.println(a);
        System.out.println(ch);
        System.out.println(b);
    }
    public static void main(String[] args)
    {
        int x=100;
        char ch='a';
        boolean y=false;
        Test t = new Test();
        t.m1(x,ch,y);
    }
}

```

**Example-4 :- For java methods it is possible to provide Objects as a parameters(in real time project).**

```

class X{}
class Emp{}
class Y{}
class Test
{
    void m1(X x ,Emp e)
    {
        System.out.println("m1 method");
    }
    static void m2(int a,Y y)
    {
        System.out.println("m2 method");
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        X x = new X();
        Emp e = new Emp();
        t.m1(x,e);           //calling of instance method by passing objects as an input
        Y y = new Y();
        Test.m2(10,y);       //calling of static method by passing objects as an input
    }
}

```

**Main method project code at realtime project level**

```

public static void main(String[] args)
{
    new Test().m1(new X(),new Emp());
    Test.m2(10,new Y());
}

```

**Example -5:-**

- Inside the class it is not possible to declare two methods with same signature , if we are trying to declare two methods with same signature compiler will raise compilation error message “**m1() is already defined in Test** ”(Java class not allowed Duplicate methods)

```

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

```

**Example-6:-** For java methods return type is mandatory otherwise the compilation will generate error message “**invalid method declaration; return type required**”.

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

**Example-7 :-**

- Declaring the class inside another class is called inner classes, java supports inner classes.
- Declaring the methods inside other methods is called inner methods but java not supporting inner methods concept if we are trying to declare inner methods compiler generate error message “**illegal start of expression**”.

```
class Test
{
    void m1()
    {
        void m2() //inner method
        {
            System.out.println("m2() inner method");
        }
        System.out.println("m1() outer method");
    }
    public static void main(String[] args)
    {
        Test t1=new Test();
        t.m1();
    }
};
```

**Example 8:-operator overloading**

- ✓ One operator with more than one behavior is called operator over loading.
- ✓ Java is not supporting operator overloading concept but only one implicit overloaded operator in java is + operator.
  - If two operands are integers then **plus (+)** perform addition.
  - If at least one operand is String then plus (+) perform concatenation.

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println(10+20);
        System.out.println("ratan"+ "anushka" + 2 + 2 + "kids");
        int a=10,b=20,c=30;
        System.out.println(a);
        System.out.println(a+"---");
        System.out.println(a+"---"+b);
        System.out.println(a+"---"+b+"----");
        System.out.println(a+"---"+b+"----"+c);
    }
}
```

**Example-9 :- methods vs. All data- types**

- ✓ By default the numeric values are integer values but to represent other format like byte, short perform typecasting.
- ✓ By default the decimal values are double values but to represent float value perform typecasting or use "f" constant. (double d=10.5; float f=20.5f;).

```
class Test
{
    void m1(byte a) { System.out.println("Byte value-->" + a); }
    void m2(short b) { System.out.println("short value-->" + b); }
    void m3(int c) { System.out.println("int value-->" + c); }
    void m4(long d) { System.out.println("long value is-->" + d); }
    void m5(float e) { System.out.println("float value is-->" + e); }
    void m6(double f) { System.out.println("double value is-->" + f); }
    void m7(char g) { System.out.println("character value is-->" + g); }
    void m8(boolean h) { System.out.println("Boolean value is-->" + h); }

    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1((byte)10);           t.m2((short)20);
        t.m3(30);                t.m4(40);
        t.m5(10.6f);             t.m6(20.5);
        t.m7('a');                t.m8(true);
    }
}
```

**Example-10:-java method calling**

- In java one method is calling another method by using method name.
- one java method is able to call more than one method. But once the method is completed the control returns to caller method.

**m1() --->calling ---->m2() ----> calling ---> m3()**  
**m1() <----- after completion-m2() <-----after completion m3()**

```
class Test
{
    void m1()
    {
        m2(); //m2() method calling
        System.out.println("m1");
        m2(); //m2() method calling
    }

    void m2()
    {
        m3(100); //m3() method calling
        System.out.println("m2 ");
        m3(200); //m3() method calling
    }

    void m3(int a)
    {
        System.out.println("m3 ");
    }

    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1(); //m1() method calling
    }
}
```

**Example-11 :-****Case 1:- This keyword not required**

In below example instance variables and local variables having different names so this keyword not required.

```
class Test
{
    //instance variables
    int a=100;
    int b=200;
    void add(int i,int j)
    {
        System.out.println(a+b);      //instance variables addition
        System.out.println(i+j);      //local variables addition
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.add(10,20);
    }
}
```

**Case 2:- This keyword required:-**

In below example instance & local variables having same name, then to represent instance variables use **this** keyword.

```
class Test
{
    //instance variables
    int a=100;
    int b=200;
    void add(int a,int b)
    {
        System.out.println(a+b);      //local variables addition
        System.out.println(this.a+this.b); //instance variables addition
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.add(10,20);
    }
}
```

**Example-12 :-**

- ✓ In java **this** keyword is instance variable hence it is not possible to use inside static area.  
If we are using **this** variable inside static context then compiler will generate error message “**non-static variable this cannot be referenced from a static context**”.
- ✓ In the static context it is not possible to use **this & super** keywords.

```
class Test
{
    int a=100;
    static void add(int a)
    {
        System.out.println(this.a);
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.add(10);
    }
}
```

**Compilation error:- non-static variable this cannot be referenced from a static context.**

***Example 13:- Inside the static area if you want access static variables use object name.***

```
class Test
{
    int a=100;
    static void add(int a)
    {
        Test t = new Test();
        System.out.println(t.a);
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.add(10);
    }
}
```

**Example-14 :- Conversion of local variables to instance variables to improve the scope of the variable.**

```

class Test
{
    //instance variables
    int val1;
    int val2;
    void values(int val1,int val2)//local variables
    {
        System.out.println(val1);
        System.out.println(val2);
        //conversion of local to instance (passing local variables values to instance variables)
        this.val1=val1;
        this.val2=val2;
    }
    void add()
    {
        System.out.println(val1+val2);
    }
    void mul()
    {
        System.out.println(val1*val2);
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.values(10,20);
        t.add();
        t.mul();
    }
}

```

*Observation :- if the instance variable names and static variable names are different we can use directly without this keyword.*

```

//instance variables
int a;
int b;
void values(int val1,int val2)//local variables
{
    System.out.println(val1);
    System.out.println(val2);
    //conversion of local to instance (passing local variables values to instance variables)
    a=val1;
    b=val2;
}

```

**Example-15 :- methods vs return type.**

- ✓ For java methods return type is mandatory & void represent return nothing.
- ✓ Methods can have any return type like
  - primitive type such as byte,short,int,long,float....etc
  - Arrays type
  - Class type
  - Interface type
  - Enum type.
- ✓ If the method is having return type other than void then must return the value by using **return** keyword otherwise compiler will generate error message "**missing return statement**"

Below syntax invalid because method must return int value by using return statement.

```
int m1()
{
    System.out.println("Anushka");
}
```

The below example is valid because it is returning int value by using return statement.

```
int m1()
{
    System.out.println("Anushka");
    return 100;
}
```

- ✓ Inside the method we are able to declare only one return statement that statement must be last statement of the method otherwise compiler will generate error message "**unreachable statement**".

The below example is invalid because return statement is must be last statement.

```
int m1()
{
    return 100;
    System.out.println("Anushka");
}
```

The below example valid because return statement is last statement.

```
int m1()
{
    System.out.println("Anushka");
    return 100;
}
```

- ✓ Every method is able to returns the value but holding (storing) that return value is optional, but it is recommended to hold the return value check the status o the method.

```

class Test
{
    int m1(int a,char ch)
    {
        System.out.println("****m1 method****");
        System.out.println(a+"---"+ch);
        return 100;
    }
    boolean m2(String str1,String str2)
    {
        System.out.println("****m2 method****");
        System.out.println(str1+"---"+str2);
        return true;
    }
    static String m3(double d,boolean b)
    {
        System.out.println("****m3 method****");
        System.out.println(d+"---"+b);
        return "ratan";
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        int x = t.m1(10,'a');
        System.out.println("m1() return value-->" +x);
        boolean b = t.m2("ratan","anu");
        System.out.println("m2() return value-->" +b);
        String str = Test.m3(10.5,true);
        System.out.println("m3() return value-->" +str);
    }
}

```

**Example 16 :-** It is possible to print return value of the method in two ways,

2. Hold the value print that value. 2. Directly print call method by using `System.out.println()`

If the method is having return type is `void` but if we are trying to call method by using

`System.out.println()` then compiler will generate error message.

```

class Test
{
    int m1()
    {
        System.out.println("m1 method");
        return 10;      }
    void m2()
    {
        System.out.println("m2 method");      }
    public static void main(String[] args)
    {
        Test t =new Test();
        int x = t.m1();
        System.out.println("return value=" +x);           //1-way
        System.out.println("return value=" +t.m1());       //2-way
        t.m2(); //valid
        //System.out.println(t.m2()); Compilation error:'void' type not allowed here
    }
}

```

**Example 17 :- Java.util.Scanner**

- ✓ Scanner class present in **java.util** package and it is introduced in 1.5 versions & it is used to take dynamic input from the keyboard.
- ✓ To communicate with system resources use System class & to take input from keyboard use **in** variable(**in=input**).

**Scanner s = new Scanner(System.in); //Scanner object creation**

<b>to get int value</b>	----> <b>s.nextInt()</b>
<b>to get float value</b>	----> <b>s.nextFloat()</b>
<b>to get byte value</b>	----> <b>s.nextByte()</b>
<b>to get String value</b>	----> <b>s.next()</b>
<b>to get single line</b>	----> <b>s.nextLine()</b>
<b>to close the input stream</b>	----> <b>s.close()</b>

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Scanner s=new Scanner(System.in);
        System.out.println("enter emp hobbies");
        String ehobbies = s.nextLine();
        System.out.println("enter emp no");
        int eno=s.nextInt();
        System.out.println("enter emp name");
        String ename=s.next();
        System.out.println("enter emp salary");
        float esal=s.nextFloat();
        System.out.println("*****emp details*****");
        System.out.println("emp no----->"+eno);
        System.out.println("emp name----->"+ename);
        System.out.println("emp sal----->"+esal);
        System.out.println("emp hobbies----->"+ehobbies);
        s.close();      //used to close the stream
    }
}
```

**Example 18:- The \s represents whitespace.**

```
import java.util.*;
public class Test
{
    public static void main(String args[])
    {
        String input = "7 tea 12 coffee";
        Scanner s = new Scanner(input).useDelimiter("\s");
        System.out.println(s.nextInt());
        System.out.println(s.next());
        System.out.println(s.nextInt());
        System.out.println(s.next());
        s.close();  }}
```

**Example 19:- retrun statement vs if-else**

```

import java.util.*;
class Test
{
    static String status(int age)
    {
        if(age>20)
        {
            return "eligible for marriage";
        }
        else
        {
            return "not eligible for marriage";
        }
    }
    public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in);
        System.out.println("enter your age:");
        int age = s.nextInt();
        String str = Test.status(age);
        System.out.println("your status is="+str);
    }
}

```

**Example-20:- methods vs. return variables**Returns local variable as a return value

```

class Test
{
    int a=10;
    int m1(int a)
    {
        System.out.println("m1() method");
        return a; //return local variable
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        int x = t.m1(100);
        System.out.println(x);
    }
}

```

D:\>java Test  
m1() method  
100

Returns instance variable as a return value(no local variable)

```

class Test
{
    int a=10;
    int m1()
    {
        System.out.println("m1() method");
        return a; //returns instance value
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        int x = t.m1();
        System.out.println(x);
    }
}

```

D:\>java Test  
m1() method  
10

If the application contains both local & instance variables with same name then first priority goes to local variables but to return instance value use this keyword.

```

class Test
{
    int a=10;
    int m1(int a)
    {
        System.out.println("m1() method");
        return this.a; //return instance variable as a return value.
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        int x = t.m1(100);
        System.out.println("m1() return value is → "+x); //printing return value
    }
}

```

**Example 21:-** The java class is able to return user defined class as a return value.

```

class X{};
class Emp{};
class Test
{
    X m1()
    {
        System.out.println("m1 method");
        X x = new X();
        return x;
    }
    Emp m2()
    {
        System.out.println("m2 method");
        Emp e = new Emp();
        return e;
    }
    static String m3()
    {
        System.out.println("m3 method");
        return "ratan";
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        X x1 = t.m1();
        Emp e = t.m2();
        String str = Test.m3();
    }
}

```

Note: when we print object reference variable it always print hash code of the object (we will discuss later).

**Example 22:** this keyword representing current class objects.

Java method is able to return current class object in two ways.

- 1) Creating object & return reference variable.
- 2) Return **this** keyword.

In above two approaches 2<sup>nd</sup> approach is recommended to return the current class object.

```

class Test
{
    Test m1()
    {
        System.out.println("m1 method");
        Test t = new Test();
        return t;
    }
    Test m2()
    {
        System.out.println("m2 method");
        return this;
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        Test t1 = t.m1();
        Test t2 = t.m2();
    }
};

```

**Example 23:- Template method**

- Let's assume to complete your task you must call four methods then,
  - you must remember number of method names.
  - Order of calling (which one is first & which one is later).
- To overcome above limitation take **x()** method it is calling four methods internally to complete our task then instead of calling four methods every time call **x()** method that perform our task that **x()** method is called template method.

```
class Test
{
    void customer() { System.out.println("customer"); }
    void product() { System.out.println("product"); }
    void selection() { System.out.println("selection"); }
    void billing() { System.out.println("billing"); }

    void deliveryManager() //template method
    {
        System.out.println("****Template method****");
        //template method is calling four methods in order to complete our task.
        customer(); product(); selection(); billing();
    }

    public static void main(String[] args)
    {
        Test t = new Test(); //normal approach
        t.customer(); t.product();
        t.selection(); t.billing();

        //by using template method
        t.deliveryManager(); //this method is calling four methods to complete our task.
    }
};
```

**Example 24:- Method recursion** A method is calling itself during execution is called recursion.

**case 1:- (normal output)**

```
class RecursiveMethod
{
    static void recursive(int a)
    {
        System.out.println("number is :- "+a);
        if (a==0)
        {
            return;
        }
        recursive(--a); //same method is calling [recursion]
    }

    public static void main(String[] args)
    {
        RecursiveMethod.recursive(10);
    }
};
```

**case 2:- (StackOverflowError)**

```
class RecursiveMethod
{
    static void recursive(int a)
    {
        System.out.println("number is :- "+a);
        if (a==0)
        {
            return;
        }
        recursive(++a);
```

```

    }
    public static void main(String[] args)
    {
        RecursiveMethod.recursive(10);
    }
}

```

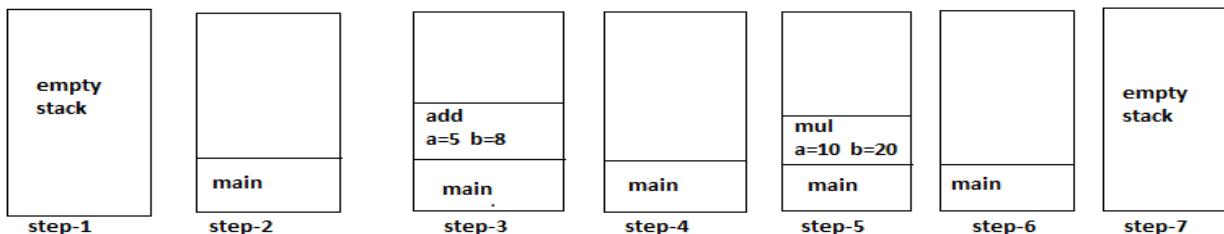
**Example 25 :- Stack Mechanism:-**

- ✓ In java program execution starts from main method called by JVM & just before calling main method JVM will creates one empty stack memory for that application.
- ✓ When JVM calls particular method then that method entry and local variables of that method stored in stack memory & when the method completed, that particular method entry and local variables are destroyed from stack memory & that memory becomes available to other methods.
- ✓ The jvm will create stack memory just before calling main method & jvm will destroyed stack memory after completion of main method.

```

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);
    }
}

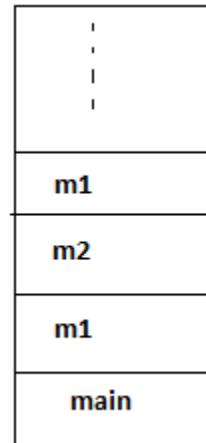
```

**Example 26 :-when we call methods recursively then we will get StackOverflowError.**

```

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

```



## CONSTRUCTORS

### Class Vs Object:-

- **Class is a logical entity it contains logics whereas object is physical entity it is representing memory.**
- **Class is blue print it decides object creation without class we are unable to create object.**
- **Based on single class (blue print) it is possible to create multiple objects but every object occupies memory.**
- **We are declaring the class by using class keyword but we are creating object by using new keyword.**
- **We are able to create object in different ways like**
  - **By using new operator**
  - **By using clone() method**
  - **By using new Instance()**
  - **By using instance factory method.**
  - **By using static factory method**
  - **By using pattern factory method**
  - **By using deserialization....etc**

But we are able to declare the class by using class keyword.

### Object creation syntax:-

**Class-name reference-variable = new class-name ();**

**Test t = new Test();**

**Test** ---> **class Name**

**t** ---> **Reference variables**

**=** ---> **assignment operator**

**new** ---> **keyword used to create object**

**Test ()** ---> **constructor**

**;** ---> **statement terminator**

When we create new instance (Object) of a class using new keyword, a constructor for that class is called.

### New :-

- **new keyword is used to create object in java.**
- **When we create object by using new operator after new keyword that part is constructor then constructor execution will be done.**

### Rules to declare constructor:-

- 1) Constructor name class name must be same.
- 2) It is possible to provide parameters to constructors (just like methods).
- 3) Constructor not allowed explicit return type. (return type declaration not possible even **void**).

There are two types of constructors,

- 1) **Default Constructor (provided by compiler).**
- 2) **User defined Constructor (provided by user) or parameterized constructor.**

**Default Constructor:-**

- ✓ Inside the class if we are not declaring at least one constructor then compiler generates zero argument constructors with empty implementation at the time of compilation.
- ✓ The compiler generated constructor is called **default constructor**.
- ✓ Inside the class default constructor is invisible mode.
- ✓ To check the default constructor provided by compiler open the .class file code by using java decompiler software.

**Application before compilation :-**

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

In above application when we create object by using new keyword “**Test t = new Test ()**” then compiler is searching for “**Test()**” constructor inside the class since not available hence compiler generate default constructor at the time of compilation.

The default constructor is always 0-argument constructor with empty implementations.

**Application after compilation :-**

```
class Test
{
    void m1()
    {
        System.out.println("m1 method");
    }
    //default constructor generated by compiler
    Test()
    {
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
    }
}
```

In above example at run time JVM will execute compiler provide default constructor during object creation.

**User defined constructor:-**

The constructors which are declared by user are called user defined constructor.

**Example :-**

```
class Test
{
    //constructor declared by user
    Test()
    {
        System.out.println("0-arg constructor");
    }
    Test(int i)
    {
        System.out.println("1-arg constructor");
    }
    Test(int a,int b)
    {
        System.out.println("2-arg constructor");
    }
    public static void main(String[] args)
    {
        Test t1=new Test();
        Test t2=new Test(10);
        Test t3=new Test(100,200);
    }
}
```

**Example:-**

- Inside the class if we are declaring at least one constructor (either 0-arg or parameterized) then compiler won't generate default constructor.
- Inside the class if we are not declaring at least one constructor (either 0-arg or parameterized) then compiler will generate default constructor.
- if we are trying to compile below application the compiler will generate error message "Cannot find symbol" because compiler is unable to generate default constructor.

```
class Test
{
    Test(int i)
    {
        System.out.println("1-arg constructor");
    }
    Test(int a,int b)
    {
        System.out.println("2-arg constructor");
    }
    public static void main(String[] args)
    {
        Test t1=new Test();
        Test t2=new Test(10);
        Test t3=new Test(100,200);
    }
}
```

E:\>javac Test.java  
 Test.java:9: cannot find symbol  
 Symbol: constructor Test ()  
 Location: class Test

**Note :-** default constructor is zero argument constructor but all zero argument constructors are not default constructors.

### **Object creation formats:-**

2-formats of object creation.

- 1) Named object (having reference variable)
- 2) Nameless object (without reference variable)

**Test t = new Test();**  
**new Test();**

```
class Test
{
    void m1()
    {
        System.out.println("m1 method");
    }
    public static void main(String[] args)
    {
        //named object      [having reference variable]
        Test t = new Test();
        t.m1();
        //nameless object   [without reference variable]
        new Test().m1();
    }
}
```

2- formats of object creation.

- 1) Eager object creation.
- 2) Lazy object creation.

```
class Test
{
    void m1(){System.out.println("m1 method");}
    public static void main(String[] args)
    {
        //Eager object creation approach
        Test t = new Test();
        t.m1();
        //lazy object creation approach
        Test t1;
        ;;;;;;; //some code here
        t1=new Test();
        t1.m1();
    }
}
```

**Advantages of constructors:-**

- 1) Constructors are used to write block of java code that code will be executed during object creation.
- 2) Constructors are used to initialize instance variables during object creation.

**Example :- default constructor execution process.**

```
class Employee
{
    //instance variables
    int eid;
    String ename;
    double esal;
    void display()
    {
        //printing instance variables values
        System.out.println("****Employee details****");
        System.out.println("Employee name :-->" + ename);
        System.out.println("Employee eid :-->" + eid);
        System.out.println("Employee sal :-->" + esal);
    }
    public static void main(String[] args)
    {
        Employee e1 = new Employee();
        e1.display();
    }
}
```

```
D:\morn11>java Employee
****Employee details****
Employee name :-->null
Employee eid :-->0
Employee sal :-->0.0
```

**Problems in above example:-**

- ✓ In above example during object creation time default constructor is executed with empty implementation and initial values of instance variables (default values) printed .
- ✓ In above example Emp object is created but default values are printing hence to overcome this limitation use user defined constructor to initialize some values to instance variables.

**Example 2:- user defined o-argument constructor execution process.**

```
class Employee
{
    //instance variables
    int eid;
    String ename;
    double esal;
    Employee() //user defined O-argument constructor
    {
        //assigning values to instance values during object creation
        eid=111;
        ename="ratan";
        esal =60000;
    }
}
```

```

void display()
{ //printing instance variables values
    System.out.println("****Employee details****");
    System.out.println("Employee name :-->" +ename);
    System.out.println("Employee name :-->" +eid);
    System.out.println("Employee name :-->" +esal);
}
public static void main(String[] args)
{
    Employee e1 = new Employee();
    e1.display();
}
}

```

*Compilation & execution process:-*

D:\morn11>javac Employee.java

D:\morn11>java Employee

\*\*\*\*Employee details\*\*\*\*

Employee name :-->ratan

Employee name :-->111

Employee name :-->60000.0

*In above example during object creation user provided 0-arg constructor executed used to initialize some values to instance variables.*

**Problem in above example:-**

*In above example when we create object it initializing values to instance variables but when we create multiple object for every object same 0-argument constructors is executing it initializing same values for all objects*

```

public static void main(String[] args)
{
    Emp e1 = new Emp();
    e1.display();
    Emp e2 = new Emp();
    e2.display();
}

```

D:\morn11>java Employee

\*\*\*\*Employee details\*\*\*\*

Employee name :-->ratan

Employee name :-->111

Employee name :-->60000.0

Employee name :-->ratan

Employee name :-->111

Employee name :-->60000.0

*To overcome above limitation use parameterized constructor to initialize different values to different objects*

**Example 3:- User defined parameterized constructors:-**

- Inside the class if the default constructor is executed & object is created but initial values of variables only printed.
- To overcome above limitation inside the class we are declaring user defined 0-argument constructor to assign some values to instance variables but when we create multiple objects for every object same constructor is executing and same values are initializing.
- To overcome above limitation declare the parameterized constructor and pass the different values to different objects.

**Example :-**

```

class Employee
{
    //instance variables
    int eid;
    String ename;
    double esal;
    Employee(int eid, String ename, double esal)      //local variables
    {
        //conversion (passing local values to instance values)
        this.eid = eid;
        this.ename = ename;
        this.esal = esal;
    }
    void display()
    {
        //printing instance variables values
        System.out.println("****Employee details****");
        System.out.println("Employee name :-->" + ename);
        System.out.println("Employee name :-->" + eid);
        System.out.println("Employee name :-->" + esal);
    }
    public static void main(String[] args)
    {
        // during object creation parameterized constructor executed
        Employee e1 = new Employee(111, "ratan", 60000);
        e1.display();
        Employee e2 = new Employee(222, "anu", 70000);
        e2.display();
    }
}
E:\>javac Test.java
E:\>java Employee
****Employee details****
Employee name :-->ratan
Employee name :-->111
Employee name :-->60000.0
****Employee details****
Employee name :-->anu
Employee name :-->222
Employee name :-->70000.0

```

**Example :- assign values to instance variables [constructor vs. method]**

```

class Student
{
    //instance variables
    int sid;
    String sname;
    int smarks;

    //constructor assigning values to instance variables
    Student(int sid,String sname,int smarks)
    {
        this.sid=sid;
        this.sname=sname;
        this.smarks=smarks;
    }

    //method assigning values to instance variables
    void assign(int sid,String sname,int smarks)
    {
        this.sid=sid;
        this.sname=sname;
        this.smarks=smarks;
    }

    void disp()
    {
        System.out.println("****student Details****");
        System.out.println("student name = "+sname);
        System.out.println("student id = "+sid);
        System.out.println("student mrks = "+smarks);
    }

    public static void main(String[] args)
    {
        Student s = new Student(111,"ratan",100);
        s.assign(222,"anu",200);
        s.disp();
    }
}

E:\>java Student
****student Details****
student name = anu
student id = 222
student mrks = 200

```

**Example :- primitive variable vs reference variable**

- **a** is variable of primitive type such as int,char,double,boolean...etc
- **t** is reference variable & it is the memory address of object.



```

class Test
{
    public static void main(String[] args)
    {
        //a is primitive variable
        int a=10;
        System.out.println(a);
        //t is reference variable
        Test t = new Test ();
        System.out.println(t);
    }
}

```

**Constructor calling:-**

To call Current class constructor use this keyword

<b>this();</b>	----> current class 0-arg constructor calling
<b>this(10);</b>	----> current class 1-arg constructor calling
<b>this(10 , true);</b>	----> current class 2-arg constructor calling
<b>this(10 , "ratan" , 'a')</b>	----> current class 3-arg constructor calling

**Example-1:-**

Call the java methods by using method name but to call the current class constructor use this keyword.

```
class Test
{
    Test()
    {
        this(100);      //current class 1-arg constructor calling
        System.out.println("0-arg constructor logics");
    }
    Test(int a)
    {
        this('g',10);   //current class 2-arg constructor calling
        System.out.println("1-arg constructor logics");
        System.out.println(a);
    }
    Test(char ch,int a)
    {
        System.out.println("2-arg constructor logics");
        System.out.println(ch+"---"+a);
    }
    public static void main(String[] args)
    {
        new Test();
    }
}
```

**Example 2:-**

Inside the constructor this keyword must be first statement otherwise compiler generate error message “call to this must be first statement in constructor”.

**No compilation error:-(this keyword first statement)**

```
Test()
{
    this(10); //current class 1-argument constructor calling
    System.out.println("0 arg");
}
```

**Compilation error:-(this keyword not a first statement)**

```
Test()
{
    System.out.println("0 arg");
    this(10); //current class 1-argument constructor calling
}
```

**Example-3:-**

1. In java one constructor is able to call only one constructor at a time but not possible to call more than one constructor.

**Compilation error:-**

```
Test()
{
    this(100); //1-arg constructor calling
    this('g',10); //2-arg constructor calling[compilation error]
    System.out.println("0-arg constructor logics");
}
```

**Object creation parts:-** Every object creation having three parts.

**1) Declaration:-**

```
Test t;                      //t is Test type
Student s;                    //s is Student type
A a;                         //a is A type
```

**2) Instantiation:-(just object creation)**

```
new Test();                  //Test object
new Student();                //student object
new A();                     //A object
```

**3) Initialization:-(during object creation perform initialization)**

```
new Test(10,20);             //during object creation 10,20 values initialized
new Student("ratan",111);    //during object creation values are initialized
new A('a',true)              //during object creation values are initialized
```

**Example :-** By using constructors copy the values of one object to another object.

```
class Student
{
    int sid;
    String sname;
    int smarks;
    Student(int sid,String sname,int smarks)
    {
        this.sid=sid;      this.sname=sname;      this.smarks=smarks;
    }
    Student(Student s)    //constructor expected Student object
    {
        this.sid=s.sid;   this.sname=s.sname;   this.smarks=s.smarks;
    }
    void disp()
    {
        System.out.println("****student Details****");
        System.out.println("student name = "+sname);
        System.out.println("student id = "+sid);
        System.out.println("student mrks = "+smarks);
    }
    public static void main(String[] args)
    {
        Student s = new Student(111,"ratan",100);
        Student s1 = new Student(s); //constructor is taking Student object
        s.disp();      s1.disp();
    }
}
```

**Constructor chaining :-**

- ✓ One constructor is calling same class constructor is called constructor calling.
- ✓ We are achieving constructor calling by using this keyword.
- ✓ Inside constructor we are able to declare only one this keyword that must be first statement of the constructor.

```
class Test
{
    Test()
    {
        this(10);
        System.out.println("0-arg cons");
    }
    Test(int i)
    {
        this(10,20);
        System.out.println("1-arg cons");
    }
    Test(int i,int j)
    {
        System.out.println("2-arg cons");
    }
    public static void main(String[] args)
    {
        new Test();
    }
}
```

Example:- Assignment

```
class Student
{
    String sname;
    int sid;
    static String college="SSCET";

    Student(String sname,int sid)
    {
        this.sname=sname;
        this.sid=sid;
    }
    void display()
    {
        System.out.println("student id="+sid);
        System.out.println("student name="+sname);
        System.out.println("student college="+college);
    }
    public static void main(String[] args)
    {
        Student s1 = new Student("ratan",111);
        s1.display();
        Student s2 = new Student("anu",222);
        s2.display();
    }
}
```

**Difference between methods and constructors:-**

<b><u>Property</u></b>	<b><u>methods</u></b>	<b><u>constructors</u></b>
<b>1)Purpose</b>	methods are used to write logics but these logics will be executed when we call that method.	Constructor is used write logics of the project but the logics will be executed during Object creation.
<b>2)Variable initialization</b>	It is initializing variable when We call that method.	It is initializing variable during object creation.
<b>3)Return type</b>	Return type not allowed Even void.	It allows all valid return Types(void,int,Boolean...etc)
<b>4)Name</b>	Method name starts with lower Case & every inner word starts With upper case. Ex: charAt(),toUpperCase()....	Class name and constructor name must be matched.
<b>5)types</b>	a) instance method b)static method	a)default constructor b)user defined constructor
<b>6)inheritance</b>	methods are inherited	constructors are not inherited.
<b>7)how to call</b>	To call the methods use method Name.	to call the constructor use <b>this</b> keyword.
<b>8)able to call how many Methods or constructors</b>	one method is able to call multiple methods at a time.	one constructors able to Call only one constructor at a time.
<b>9)this</b>	to call instance method use this Keyword but It is not possible to call static method.	To call constructor use this keyword but inside constructor use only one this statement.
<b>10)Super</b>	used to call super class methods.	Used to call super class constructor
<b>11)Overloading</b>	it is possible to overload methods	it is possible to overload cons.
<b>12)compiler generate Default cons or not</b>	yes	does not apply
<b>13)compiler generate Super keyword.</b>	yes	does not apply.

**Instance Blocks:-**

- Instance blocks are used to write the logics of projects & these logics are executed during object creation just before constructor execution.
- Instance blocks execution depends on object creation it means if we are creating 10 objects 10 times constructors are executed just before constructors instance blocks are executed.
- Instance block syntax  
`{ //logics here }`

**Example 1:-**

```
class Test
{
    //instance block
    {
        System.out.println("instance block:logics");
    }
    Test()
    {
        System.out.println("constructor:logics");
    }
    public static void main(String[] args)
    {
        new Test();
    }
}
```

**Example 2:-**

Inside the class it is possible to declare multiple instance blocks but the execution order is top to bottom.

```
class Test
{
    {
        System.out.println("instance block-1:logics");
    }
    Test()
    {
        System.out.println("0-arg constructor:logics");
    }
    {
        System.out.println("instance block-2:logics");
    }
    Test(int a)
    {
        System.out.println("1-arg constructor:logics");
    }
    public static void main(String[] args)
    {
        new Test();
        new Test();
        new Test(10);
    }
}
```

**Example 3:-**

- ✓ Instance block execution depends on object creation but not constructor exaction.
- ✓ In below example two constructors are executing but only one object is creating hence only one time instance block is executed.

```
class Test
{
    {
        System.out.println("instance block-1:logics");
    }
    Test()
    {
        this(10);
        System.out.println("0-arg constructor:logics");
    }
    Test(int a)
    {
        System.out.println("1-arg constructor:logics");
    }
    public static void main(String[] args)
    {
        new Test();
    }
}
```

**Example 4:-**

**Case 1:-** When we declare instance block & instance variable the execution order is top to bottom.  
In below example instance block is declared first so instance block is executed first.

```
class Test
{
    {
        System.out.println("instance block");           }      //instance block
    int a=m1();          //instance variables
    int m1()
    {
        System.out.println("m1() method called by variable");
        return 100;
    }
    public static void main(String[] args)
    {
        new Test();
    }
}
```

D:\morn11>java Test  
**instance block**  
**m1() method called by variable**

**case 2:-** When we declare instance block & instance variable the execution order is top to bottom.  
In below example instance variable is declared first so instance block is executed first.

```
class Test
{
    int a=m1();    //instance variables
    int m1()
    {
        System.out.println("m1() method called by variable");
```

```

        return 100;
    }
    {
        System.out.println("instance block");
    }
    public static void main(String[] args)
    {
        new Test();
    }
}

D:\morn11>java Test
m1() method called by variable
instance block

```

**Example-5 :-**

- Instance blocks are used to initialize instance variables during object creation but just before constructor execution.
- In java it is possible to initialize the values in different ways
  - By using constructors
  - By using instance blocks
  - By using methods
  - By using setter methods.....etc

```

class Emp
{
    int eid;
    //instance block initializing values
    {
        eid=111;
    }
    //constructor initializing values
    Emp()
    {
        eid=222;
    }
    //method initializing values
    void assign()
    {
        eid=333;
    }
    void disp()
    {
        System.out.println("****Employee Details****");
        System.out.println("emp id=" +eid);
    }
    public static void main(String[] args)
    {
        Emp e = new Emp();
        e.disp();
    }
};

```

**static block:-**

- Static blocks are used to write the logics of project that logics are executed during .class file loading time.
- In java .class file is loaded only one time hence static blocks are executed once per class.
- Static block syntax,

```
static
{ //logics here
}
```

Note : instance blocks execution depends on object creation but static blocks execution depends on .class file loading.

**Example-1 :-**

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

**Example-3 :-static block is initializing variable**

Static blocks are used to initialize static variables during .class file loading.

```
class Emp
{
    static int eid;
    static
    {
        eid=111;
    }
    static void assign()
    {
        eid=333;
    }
    static void disp()
    {
        System.out.println(eid);
    }
    public static void main(String[] args)
    {
        Emp e = new Emp();
        e.assign();
        e.disp();
    }
}
```

**Example-4 :- To execute static blocks inside the class main() method is mandatory or optional.**

```
class Test
{
    static
    {
        System.out.println("static block");
    }
}
```

- ✓ When we execute above code up to 1.5 version the code is compiled & executed it will generate output.

Note : Based on above point up to 1.5 version it is possible to print some statements in output console without using main method inside the class.

- ✓ When we execute the above code from 1.6 version onwards the code is not compiled it will generate compilation error.

Note:- based on above point from 1.6 version onwards it is not possible to print some statements in output console without using main method inside the class.

**Example-5:-**

- ✓ When we execute the .class file manually then to execute static blocks inside the class main() method mandatory.
- ✓ When we load the .class file into memory programmatically then to execute the static blocks main() method is optional.
- ✓ In below example both files must present in same folder (same location).

**File 1:- Demo.java**

```
class Demo
{
    static
    {
        System.out.println("Demo class static block");
    }
    void m1()
    {
        System.out.println("Demo class m1 method");
    }
};
```

**File -2:- Test.java**

```
class Test
{
    static
    {
        System.out.println("Test class static blocks");
    }
    public static void main(String[] args) throws Exception
    {
        Class c = Class.forName("Demo");
        Demo d = (Demo)c.newInstance();
        d.m1();
    }
}
```

### **Object-Oriented Programming Concepts**

- The first object oriented programming is : **Simula, smalltalk**
- In object oriented programming languages everything represented in the form of object.
- Object is real world entity that has state & behavior.  
Examples:- such as pen, chair, table, house....etc.

Every object contains three characteristics,

- 1) **State** : well defined condition of an item (instance variable/fields/properties)
- 2) **Behavior** : effects on an item (methods/behavior)
- 3) **identity** : identification number of an item(hash code)

Example :-

<i>Object</i>	:	<i>Car</i>
<i>State</i>	:	<i>gear, speed, color...etc</i>
<i>Behavior</i>	:	<i>current speed, current gear, Accelerate...etc</i>
<i>Identity</i>	:	<i>car number</i>
<i>Object</i>	:	<i>house</i>
<i>State</i>	:	<i>location</i>
<i>Behavior</i>	:	<i>doors open/close.</i>
<i>Identity</i>	:	<i>house no</i>

The main building blocks of oops are,

1. **Inheritance**
2. **Polymorphism**
3. **Abstraction**
4. **Encapsulation....etc**

Inheritance:-

1. The process of acquiring properties (variables) & methods (behaviors) from one class to another class is called inheritance.
2. We are achieving inheritance concept by using **extends** keyword. Inheritance is also known as **is-a** relationship.
3. Extends keyword is providing relationship between two classes..
4. The main objective of inheritance is code extensibility whenever we are extending the class automatically code is reused.
5. In inheritance one class providing properties & another class is acquiring the properties.
6. In inheritance parent class is giving properties & Child is acquiring properties from Parent.

**Application code before inheritance**

```

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(){}
};

```

- a. Duplication of code.
- b. Code length is increased.

**Note:** - To reduce length of the code and redundancy of the code & to improve re-usability of code sun people introduced inheritance concept.

**Object creation of parent & child classes:-**

In java it is possible to create objects for both parent and child classes,

- ✓ If we are creating object for parent class it is possible to access only parent specific methods.

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

- ✓ if we are creating object for child class it is possible to access both parent & child specific methods.

**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();**

**Important points:-**

```

class A
{
}

class B extends A
{
}

class C extends B
{
}

```

- 1) in java if we are extending the class then it will be parent class , if we are not extending the class then **object** class will become parent class.
- 2) In above example A class is direct child class of object & B,C classes are indirect child classes of object.it represent in java every class is child of **object** either directly(A) or indirectly(B,C).
- 3) The root class of all java classes is “**object**” class.
- 4) Every java class contain parent class except **object** class.

**Application code after inheritance**

```

class A //parent class or super class or base
{
    void m1(){}
    void m2(){}
};

class B extends A //child class or sub or derived
{
    void m3(){}
    void m4(){}
};

class C extends B
{
    void m5(){}
    void m6(){}
};

```

- a. Eliminated duplication.
- b. Length of the code is decreased.
- c. Reusing properties in child classes.

- 5) Object class present in **java.lang** package and it contains 11 methods & all java classes able to use these 11 methods because Object class is root class of all java classes.

To check the predefined support use javap command.

E:\>javap java.lang.Object

```
public class java.lang.Object
```

```
{     public final native java/lang/Class<?> getClass();
      public native int hashCode();
      public boolean equals(java.lang.Object);
      protected native java.lang.Object clone() throws ava.lang.CloneNotSupportedException;
      public java.lang.String toString();
      public final native void notify();
      public final native void notifyAll();
      public final native void wait(long) throws java.lang.InterruptedException;
      public final void wait(long, int) throws java.lang.InterruptedException;
      public final void wait() throws java.lang.InterruptedException;
      protected void finalize() throws java.lang.Throwable;
}
```

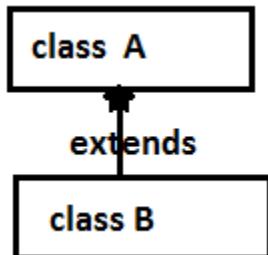
### **Types of inheritance :-**

There are five types of inheritance in java,

1. **Single inheritance**
2. **Multilevel inheritance**
3. **Hierarchical inheritance**
4. **Multiple inheritance**
5. **Hybrid Inheritance**

### **Single inheritance:-**

- One class has 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**.



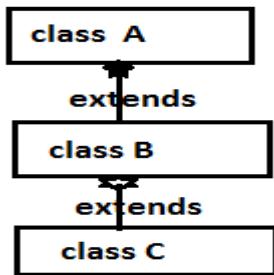
**Class B extends A** ==> class B acquiring properties of A class.

### **Example:-**

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

**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:-**

```

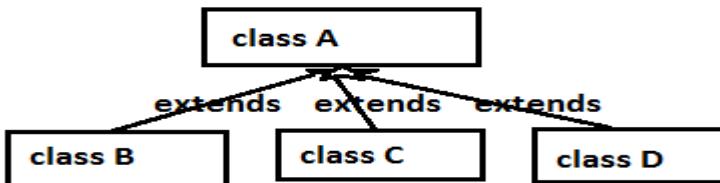
class A
{
    void m1(){System.out.println("m1 method");}
}

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();
    }
}
  
```

**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
{
}

class B extends A
{
}

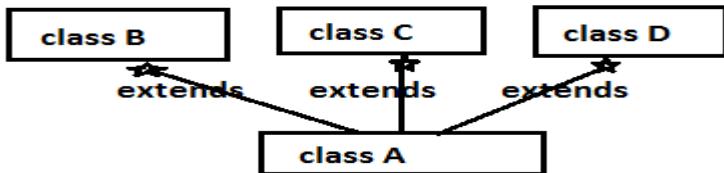
class C extends A
{
}
  
```

**Multiple inheritance:-**

- One sub class is extending more than one super class is called Multiple inheritance and java not supporting multiple inheritance because it is creating ambiguity problems (confusion state).
- Java not supporting multiple inheritances hence in java one class able to extends only one class at a time but it is not possible to extends more than one class.

**Class A extends B**      ==>valid

**Class A extends B ,C**      ==>invalid

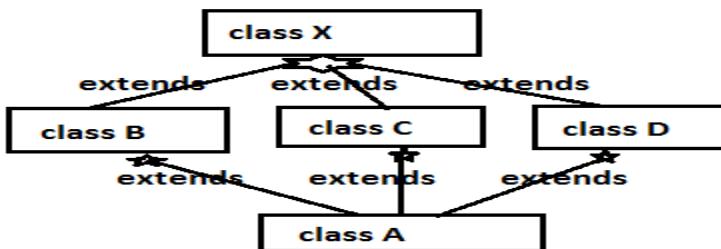
**Example:-**

```

class Parent1
{
    void money() {System.out.println("parent1 money");}
}
class Parent2
{
    void money() {System.out.println("parent2 money");}
}
class Child extends Parent1,Parent2
{
    public static void main(String[] args)
    {
        Child c = new Child();
        c.money();      //ambiguity problems
    }
}
  
```

**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.

**Preventing inheritance:-**

- ✓ You can prevent sub class creation by using final modifier.
- ✓ If a parent class declared as final we can't create sub class for that class.

```
final class Parent
```

```
{ }
```

```
class Child extends Parent
```

```
{ }
```

**compilation error:- cannot inherit from final Parent**

**Instanceof operator:-**

- It is used check the type of the object and it returns Boolean value as a return value.

**Syntax:- reference-variable instanceof class-name;**

- To use the instanceof operator the class name & reference variable must have some relationship either parent to child or child to parent otherwise compiler will generate error message "inconvertible types".
- If the relationship is child to parent it returns **true** & the relationship is parent to child it return **false**.

**Example :-**

```
class Animal{    };
class Dog extends Animal{        };
class Test
{
    public static void main(String[] args)
    {
        Test t = new Test();
        Animal a = new Animal();
        Dog d = new Dog();
        Object o = new Object();
        System.out.println(d instanceof Animal);           //true
        System.out.println(a instanceof Object);           //true
        System.out.println(a instanceof Dog);              //false
        System.out.println(t instanceof Object);           //true
        System.out.println(o instanceof Animal);           //false
        //System.out.println(t instanceof Animal);          compilation error:inconvertible types
    }
}
```

**Association:-**

- Class A uses class B
- When one object wants another object to perform services for it.
- Relationship between teacher and student, number of students associated with one teacher or one student can associate with number of teachers. But there is no ownership and both objects have their own life cycles.

**Example-1:-**

In below example student uses Teacher class object services.

```
class Teacher
{
    void course()
    {
        System.out.println("corejava by ratan");
    }
}
class Student
{
    public static void main(String[] args)
    {
        Teacher t = new Teacher();
        t.course();
    }
};
```

**Aggregation:-**

- Class A has instance of class B is called aggregation.
- Class A can exists without presence of class B . a university can exists without chancellor.
- Take the relationship between teacher and department. A teacher may belongs to multiple departments hence teacher is a part of multiple departments but if we delete department object teacher object will not destroy.

**Example:-****Address.java**

```
class Address
{
    //instance variables
    int dno;
    String state;
    String country;
    Address(int dno,String state,String country) //local variables
    {
        //conversion process
        this.dno=dno;
        this.state= state;
        this.country = country;
    }
};
```

**Heroin.java:**

```
class Heroin
{
    String hname;
    int hage;
    Address addr; //reference of address class [dno,state,country]
    Heroin(String hname,int hage,Address addr)
    {
        //conversion process
        this.hname = hname;
        this.hage = hage;
        this.addr = addr;
    }
    void display()
    {
        System.out.println("*****heroin details*****");
        System.out.println("heroin name-->" +hname);
        System.out.println("heroin age-->" +hage);
        System.out.println("heroin address-->" +addr.country+ " "+addr.state+ " "+addr.hno)
    }
    public static void main(String[] args)
    {
        Address a1 = new Address("india","banglore",111);
        Heroin h1 = new Heroin("anushka",30,a1);
        h1.display();
        Address a2 = new Address("US","california",333);
        Heroin h2 = new Heroin("AJ",40,a2);
        h2.display();
    }
}
```

```

        }
    }
}
```

**The above Project level main method code:-**

```

public static void main(String[] args)
{
    new Heroin("anushka",30,new Address("india","banglore",111)).display();
    new Heroin("AJ",40,new Address("US","california",333)).display();
}
```

**Example:-**

**Test1.java:-**

```

class Test1
{
    int a;
    int b;
    Test1(int a,int b)
    {
        this.a=a;
        this.b=b;
    }
};
```

**Test2.java:-**

```

class Test2
{
    boolean b1;
    boolean b2;
    Test2(boolean b1,boolean b2)
    {
        this.b1=b1;
        this.b2=b2;
    }
};
```

**Test3.java:-**

```

class Test3
{
    char ch1;
    char ch2;
    Test3(char ch1,char ch2)
    {
        this.ch1=ch1;
        this.ch2=ch2;
    }
};
```

**MainTest.java:-**

```

class MainTest
{
    //instance variables
    Test1 t1;
    Test2 t2;
    Test3 t3;
    MainTest(Test1 t1 ,Test2 t2,Test3 t3) //local variables
    {
        //conversion of local-instance
        this.t1 = t1;
        this.t2 = t2;
        this.t3 = t3;
    }
    void display()
    {
        System.out.println("Test1 object values:- → "+t1.a+"---- "+t1.b);
        System.out.println("Test2 object values:- → "+t2.b1+"---- "+t2.b2);
        System.out.println("Test3 object values:- → "+t3.ch1+"---- "+t3.ch2);
    }
    public static void main(String[] args)
    {
        Test1 t = new Test1(10,20);
        Test2 tt = new Test2(true,true);
        Test3 ttt = new Test3('a','b');
        MainTest main = new MainTest(t,tt,ttt);
        main.display();
        // new MainTest(new Test1(10,20),new Test2(true,false),new Test3('a','b'));
    }
};
```

**Composition :-**

- Class A owns class B , it is a strong type of aggregation. There is no meaning of child without parent.
- Order consists of list of items without order no meaning of items. or bank account consists of transaction history without bank account no meaning of transaction history or without student class no meaning of marks class.
- Let's take Example house contains multiple rooms, if we delete house object no meaning of room object hence the room object cannot exists without house object.
- Relationship between question and answer, if there is no meaning of answer without question object hence the answer object cannot exist without question objects.
- Relationship between student and marks, there is no meaning of marks object without student object.

**Example :-****Marks.java**

```
class Marks
{
    int m1,m2,m3;
    Marks(int m1,int m2,int m3)    //local variables
    {
        this.m1=m1;                this.m2=m2;                this.m3=m3;
    }
};
```

**student.java**

```
class Student
{
    Marks mk;      //without student class no meaning of marks is called "composition"
    String sname;
    int sid;
    Student(Marks mk,String sname,int sid) //local variables
    {
        this.mk = mk;                  this.sname = sname;            this.sid = sid;
    }
    void display()
    {
        System.out.println("student name:-->" + sname);
        System.out.println("student id:-->" + sid);
        System.out.println("student marks:-->" + mk.m1 + " --- " + mk.m2 + " -- " + mk.m3);
    }
    public static void main(String[] args)
    {
        Marks m1 = new Marks(10,20,30);
        Student s1 = new Student(m1,"ratan",111);
        s1.display();
        // new Student(new Marks(10,20,30),"ratan",111).display(); project code
        Marks m2 = new Marks(100,200,300);
        Student s2 = new Student(m2,"anu",222);
        s2.display();
        // new Student(new Marks(100,200,300),"anu",222).display(); project code
    }
}
```

**Object delegation:-**

The process of sending request from one object to another object is called object delegation.

**Example :-**

```
class RealPerson      //delegate class
{
    void book(){System.out.println("real java book");}
}

class DummyPerson   //delegator class
{
    RealPerson r = new RealPerson();
    void book( ) {r.book( );} //delegation
}

class Student
{
    public static void main(String[] args)
    {
        //outside world thinking dummy Person doing work but not.
        DummyPerson d = new DummyPerson();
        d.book();
    }
}
```

**Super keyword:-**

“this” keyword is used to represent current class object & “super” keyword is used to represent super class object.

1. Super class variables.
2. Super class methods.
3. Super class constructors.
4. Super class instance blocks.
5. Super class static blocks.

**super class variables calling:-**

```
class Parent
{
    int a=10,b=20;
}

class Child extends Parent
{
    int a=100;
    int b=200;
    void m1(int a,int b)    //local variables
    {
        System.out.println(a+b);           //local variables addition
        System.out.println(this.a+this.b);  //current class variables addition
        System.out.println(super.a+super.b); //super class variables addition
    }
    public static void main(String[] args)
    {
        new Child().m1(1000,2000);
    }
}
```

**super class methods calling:-**

```

class Parent
{
    void m1(){System.out.println("parent m1() method");}
};

class Child extends Parent
{
    void m1()      {System.out.println("child class m1() method");}
    void m3()
    {
        this.m1();
        super.m1();
    }
    public static void main(String[] args)
    {
        new Child().m3();
    }
};

```

To call the super class methods **super** keyword is mandatory but to represent current class methods **this** keyword is optional.

**super class constructors calling:-****Example-1:-**

To call the current class constructors use **this** keyword but to call super class constructor use **super** keyword.

<b>super()</b>	----> <b>super class 0-arg constructor calling</b>
<b>super(10)</b>	----> <b>super class 1-arg constructor calling</b>
<b>super(10,20)</b>	----> <b>super class 2-arg constructor calling</b>
<b>super(10,'a',true)</b>	----> <b>super class 3-arg constructor calling</b>

```

class Parent
{
    Parent() {System.out.println("parent 0-arg constructor");}
};

class Child extends Parent
{
    Child()
    {
        this(10);           //current class 1-arg constructor calling
        System.out.println("Child 0-arg constructor");
    }
    Child(int a)
    {
        super();           //super class 0-arg constructor calling
        System.out.println("child 1-arg constructor-->" + a);
    }
    public static void main(String[] args)
    {
        new Child();
    }
};

```

**Example-2:-**

Inside the constructor super keyword must be first statement otherwise compiler generates error message “**call to super must be first line in constructor**”.

**No compilation error:-**

```
Child()
{
    this(10);           // (this must be first line)
    System.out.println("Child 0-arg constructor");
}
Child(int a)
{
    super();            //(super must be first line)
    System.out.println("child 1-arg constructor--->" + a);
}
```

**Compilation Error:-**

```
Child()
{
    System.out.println("Child 0-arg constructor");
    this(10);           //compilation error
}
Child(int a)
{
    System.out.println("child 1-arg constructor--->" + a);
    super();            //(compilation Error)
}
```

**Example-3:-**

Inside the constructor it is possible to use either this keyword or super keyword but,

- ✓ Two super keywords are not allowed.
- ✓ Two this keywords are not allowed.
- ✓ Both super & this keyword also not allowed.

**Compilation Error:-**

```
Child()
{
    this(10);
    super();
    System.out.println("Child 0-arg constructor");
}
Child()
{
    super(10);
    super();
    System.out.println("Child 0-arg constructor");
}
Child()
{
    this(10);
    this();
    System.out.println("Child 0-arg constructor");
}
```

**Example-4:-**

In below example parent class default constructor is executed that is provided by compiler.

```
class Parent
{
    // default constructor
};

class Child extends Parent
{
    Child()
    {
        super()
        System.out.println("Child 0-arg constructor");
    }

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

D:\>java Child

**Child 0-arg constructor**

**Example-5:-**

1. Inside the constructor whether it is a 0-argument or parameterized if we are not declaring **super** or **this** keyword then compiler generate super keyword at first line of the constructor.
2. The compiler generated super keyword is always 0-arg constructor calling.

```
class Parent
{
    Parent() { System.out.println("parent 0-arg constructor"); }
};

class Child extends Parent
{
    Child()
    {
        //super(); generated by compiler at compilation time
        System.out.println("Child 0-arg constructor");
    }

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

D:\>java Child

**parent 0-arg constructor**  
**Child 0-arg constructor**

**Example-6:-**

In below example child class is calling parent class 0-argument constructor since not there so compiler generate error message.

```
class Parent
{
    Parent(int a) { System.out.println("parent 1-arg cons-->"+a); }
}

class Child extends Parent
{
    Child()
    {
        //super(); generated by compiler
        System.out.println("Child 0-arg constructor");
    }

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

**Example-7:-**

In below compiler generate default constructor and inside that default constructor super keyword is generated.

**Application code before compilation:- ( .java )**

```
class Parent
{
    Parent() {
        System.out.println("parent 0-arg cons");
    }
}

class Child extends Parent
{
    public static void main(String[] args)
    {
        new Child();
    }
}
```

**Application code after compilation:- ( .class )**

```
class Parent
{
    Parent()
    {
        System.out.println("parent 0-arg cons");
    }
}

class Child extends Parent
{
    /* below code is generated by compiler
    Child()
    {
        super();
    } */
    public static void main(String[] args)
    {
        new Child();
    }
}
```

**Example-8 :-**

In below example in Test class 0-argument constructor compiler generate super keyword it execute parent class(Object) default constructor.

```
class Test extends Object
{
    Test()
    {
        //super(); generated by compiler
        System.out.println("Test class constructor");
    }

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

**Example-9:-**

In below example in child class 1-argument & 0 argument constructors compiler generate super keyword hence parent class 0-argument constructor will be executed.

```
class Parent
{
    Parent(){System.out.println("parent 0-arg cons");}
};

class Child extends Parent
{
    Child()
    {
        //super(); generated by compiler
        System.out.println("Child 0-arg constructor");
    }

    Child(int a)
    {
        //super(); generated by compiler
        System.out.println("child 1-arg cons");
    }

    public static void main(String[] args)
    {
        new Child();
        new Child(10);
    }
};
```

**Example-10:-**

```
class GrandParent
{
    int c;
    GrandParent(int c)
    {
        this.c=c;
    }
};

class Parent extends GrandParent
{
    int b;
    Parent(int b,int c)
    {
        super(c);
        this.b=b;
    }
};

class Child extends Parent
{
    int a;
    Child(int a,int b,int c)
    {
        super(b,c);
        this.a=a;
    }

    void disp()
    {
        System.out.println("child class =" +a);
        System.out.println("parent class =" +b);
        System.out.println("grandparent class =" +c);
    }
}

public static void main(String[] args)
{
    new Child(10,20,30).disp();
}
```

**Super class instance blocks:-****Example-1:-**

In parent and child relationship first parent class instance blocks are executed then child class instance blocks are executed.

```
class Parent
{
    {System.out.println("parent instance block");}      //instance block
}
class Child extends Parent
{
    { System.out.println("Child instance block"); } //instance block
    Child()
    {
        System.out.println("chld 0-arg cons");
    }
    public static void main(String[] args)
    {
        new Child();
    }
};
```

**Super class static blocks:-****Example-1:-**

In parent and child relationship first parent class static blocks are executed only one time then child class static blocks are executed only one time because static blocks are executed with respect to .class loading.

Instance block execution depends on object creation & static block execution depends on class loading.

```
class Parent
{
    static{System.out.println("parent static block");}
    {System.out.println("parent instance block");}      //static block
                                                       //instance block
}
class Child extends Parent
{
    static{System.out.println("child static block");}
    {System.out.println("child instance block");}      //static block
                                                       //instance block
    public static void main(String[] args)
    {
        new Child();
        new Child();
    }
};
E:\>java Child
parent static block
child static block
parent instance block
child instance block
parent instance block
child instance block
```

**Example: - object usage in real time projects.****File-1: Gmail.java**

```
class Gmail
{
    void compose() { System.out.println("to perform compose operation"); }
    void inbox() { System.out.println("to perform inbox operation"); }
    void sentItem() { System.out.println("to perform sent Item operation"); }
}
```

**File-2: UserOperations1.java**

**Problem:** in below example we are creating object locally & in every method we are creating object it increases duplication of the code.

```
class UserOperations
{
    void ratan()
    {
        System.out.println("ratan operations:");
        Gmail g = new Gmail();
        g.compose();
    }

    void anu()
    {
        System.out.println("anu operations:");
        Gmail g = new Gmail();
        g.inbox();
    }

    void yadhu()
    {
        System.out.println("yadhu operations:");
        Gmail g = new Gmail();
        g.sentItem();
    }
}
```

**File-2: UserOperations2.java**

**Solution:** in below example we are creating object globally hence every method is able to access this variable.

```
class UserOperations
{
    Gmail g;
    void ratan()
    {
        System.out.println("ratan operations:");
        g= new Gmail();
        g.compose();
    }

    void anu()
    {
        System.out.println("anu operations:");
        g.inbox();
    }

    void yadhu()
    {
        System.out.println("yadhu operations:");
        g.sentItem();
    }
}
```

**File 3: Test.java**

```
class Test
{
    public static void main(String[] args)
    {
        UserOperations o = new UserOperations();
        o.ratan();           o.anu();           o.yadhu();
    }
}
```

**Polymorphism:-**

- The ability to appear in more forms is called polymorphism.
- One thing can exhibits more than one form is called polymorphism.
- One functionality with different actions is called polymorphism.
- One person with different behaviours is called polymorphism.
- Polymorphism is a Greek word poly means many and morphism means forms.

**There are two types of polymorphism in java.**

- 1) Compile time polymorphism / static binding / early binding  
*Example :- method overloading.*
- 2) Runtime polymorphism / dynamic binding / late binding.  
*Example :- method overriding.*

**Compile time polymorphism [Method Overloading]:-**

- 1) If java class allows more than one method with same name but different number of arguments or same number of arguments but different data types those methods are called overloaded methods.
  - a. Same method name but different number of arguments.  
`void m1(int a){ }  
void m1(int a,int b){ }`
  - b. Same method name & same number of arguments but different data types.  
`void m1(int a){ }  
void m1(char ch){ }`
- 2) To achieve overloading concept one java class sufficient.
- 3) It is possible to overload any number of methods in single java class.

**Example:-**

```
class Test
{
    //below three methods are overloaded methods.
    void m1(int i)
    {
        System.out.println("int-argument method");
    }
    void m1(int i,int j)
    {
        System.out.println("int,int argument method=");
    }
    void m1(char ch)
    {
        System.out.println("char-argument method");
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1(10);
        t.m1(10,20);
        t.m1('a');
    }
}
```

**Example:-**

- ✓ In method overloading it is possible to have different data types for overloaded method.
- ✓ While overloading the methods check the signature(methodname+parameters) of the method but not return type.

```
class Test
{
    //below two methods are overloaded methods
    double m1(int a,int b)
    {
        System.out.println("int,int arguments method");
        return 20.5;
    }
    int m1(float f)
    {
        System.out.println("float argument method");
        return 100;
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        double d = t.m1(10,20);
        System.out.println("return type="+d);
        int x = t.m1(10.5f);
        System.out.println("return type="+x);
    }
}
```

**method overloading with type promotion**

byte → short  
 char → Int → long → float → double

**Example:-**

```
class Test
{
    void m1(int a,long b)
    {
        System.out.println("int,long arguments method");
    }
    void m1(float f)
    {
        System.out.println("float argument");
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1(10,20);
        t.m1((byte)10,(short)20);
        t.m1(10);
        t.m1(10L);
    }
}
```

```
E:\>java Test
int,long arguments method
int,long arguments method
float argument
float argument
```

**Example:-**

```

class Test
{
    void sum(int a,int b)
    {
        System.out.println("int arguments method");
        System.out.println(a+b);
    }
    void sum(long a,long b)
    {
        System.out.println("long arguments method");
        System.out.println(a+b);
    }
    void sum(int a,long b)
    {
        System.out.println("int,long arguments method");
        System.out.println(a+b);
    }
    void sum(long a,int b)
    {
        System.out.println("long,int arguments method");
        System.out.println(a+b);
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        t.sum(1,2);
        t.sum(10L,20L);
        t.sum(100,200L);
        t.sum(1000L,2000);
    }
}

```

**Types of overloading:-**

There are three types of overloading in java,

- a. Method overloading } explicitly by the programmer
- b. Constructor overloading }
- c. Operator overloading } implicitly by the JVM('+ addition& concatenation)

**Constructor Overloading:-**

If the class contains more than one constructors with same name but different arguments or same number of arguments with different data types those constructors are called overloaded constructors.

- a. Same constructor name but different number of arguments.

```

Test(int a){ }           //assume Test is java class
Test(int a,int b){ }

```

- b. Same constructor name & same number of arguments but different data types.

```

Test(int a){ }
Test(char ch){ }

```

```

class Test
{
    //overloaded constructors
    Test()
    {
        System.out.println("0-arg constructor");
    }
    Test(int i)
    {
        System.out.println("int argument constructor");
    }
    Test(char ch,int i)
    {
        System.out.println("char,int argument constructor");
    }
    Test(char ch)
    {
        System.out.println("char argument constructor");
    }
    public static void main(String[] args)
    {
        new Test();
        new Test(10);
        new Test('a',100);
        new Test('r');
    }
}

```

**Operator overloading:-**

- ✓ One operator with different behaviors is called Operator overloading .
- ✓ Java is not supporting operator overloading but only one overloaded in java language is ‘+’.
  - If both operands are integer then “+” performs addition.
  - If at least one operand is String then “+” perform concatenation.

**Example:-**

```

class Test
{
    public static void main(String[] args)
    {
        int a=10;
        int b=20;
        System.out.println(a+b);           //30 [addition]
        System.out.println(a+"ratan");    //10Ratan [concatenation]
        System.out.println("ratan"+ "anu" + 2+2+"kids");
        System.out.println(2+2+"kids" + "ratan" + "anu");
    }
}

```

**Runtime polymorphism [Method Overriding]:-**

- ✓ To achieve method overloading one java class sufficient but to achieve method overriding we required two java classes with parent and child relationship.
- ✓ The method implementations already present in parent class,
  - a. If the child class required that implementation then access those implementations.
  - b. If the child class not required, parent class method implementations then override parent class method in child class to provide child specific implementations.
- ✓ The sub class overrides super class method to provide sub class method implementations.
- ✓ In overriding parent class method is called ==> **overridden method**  
Child class method is called ==> **overriding method**

**Example :-**

In below example marry method present in parent class with some implementations but child class overriding marry method to provide child specific implementation is called overriding.

```
class Parent
{
    void property()
    {
        System.out.println("money+land+hhouse");
    }
    void marry() //overridden method
    {
        System.out.println("black girl");
    }
}
class Child extends Parent
{
    void marry() //overriding method
    {
        System.out.println("white girl/red girl");
    }
    public static void main(String[] args)
    {
        Child c=new Child();
        c.property();
        c.marry();
    }
}
```

```
E:\>java Child
money+land+hhouse
white girl/red girl
```

**While overriding methods must follow these rules:-**

- 1) Overridden method signature & overriding method signatures must be same.
- 2) The return types of overridden method & overriding method must be same (at primitive level).
- 3) While overriding it is possible to change return type by using co-variant return types concept.
- 4) Final methods can't override.
- 5) Static method can't override but method hiding possible.
- 6) Private methods can't override.
- 7) While overriding it is not possible to maintain same level permission or increasing order but not decreasing.
- 8) Overriding with exception handling rules.

**Method overriding rule 1:-**

While overriding methods the overridden method signature and overriding method signature must be same.

Method signature nothing but method-name & parameters list.

```
class Parent
{
    void marry(){ }      //overridden method
}
class Child extends Parent
{
    void marry(){ }      //overriding method
};
```

**Method overriding rule-2:-**

While overriding method overridden method return type & overriding method return type must be same at primitive level (byte,int,double,boolean...etc) otherwise compiler will generate error message.

```
class Parent
{
    void marry(){ }      //overridden method
}
class Child extends Parent
{
    int marry(){ }       //overriding method
};
```

**compilation error:-** marry() in Child cannot override marry() in Parent  
return type int is not compatible with void

**Method overriding rule-3:-**

- ✓ Before java 1.5 version it is not possible to override the method by changing return type but from java 5 versions it is possible to override the method by changing return type by using co-variant return type's concept.
- ✓ While overriding methods it is possible to change the return type of overridden method & overriding methods at class level by using co-variant return type concept but not primitive level.
- ✓ The return type of overriding method is must be sub-type of overridden method return type this is called covariant return types.

**Note :while overriding methods it is possible to change return type at class-level but not primitive level.**

```
class Animal{ }
class Dog extends Animal{ }
class Parent
{
    Animal m1() //overridden method
    {
        System.out.println("parent m1");
        return new Animal();
    }
}
class Child extends Parent
{
    Dog m1() //overriding method
    {
        System.out.println("child m1");
        return new Dog();
    }
    public static void main(String[] args)
    {
        Parent p = new Parent();
        Animal a = p.m1();
        Child c = new Child();
        Dog d = c.m1();
    }
};
```

**Method overriding rule-4:-**

- 1) If an overridden method is final it is not possible to override that method in child class.
- 2) Final classes are preventing inheritance concept & final methods are preventing overriding concept.

**Example :-**

```
class Parent
{
    final void marry(){}
} //overridden method

class Child extends Parent
{
    void marry(){}
} //overriding method

Compilation error:- marry() in Child cannot override marry() in Parent
overridden method is final
```

**Example: final variables**

When we declare variable as a final it is not possible to change the value of final variable. If we are trying to change final variable compiler will generate error message.

Final variables are fixed constants it is not possible to perform modifications.

**Case 1:-** in java for the local variables only one modifier is applicable that is **final**.

```
class Test
{
    public static void main(String[] args)
    {
        final int a=10;
        a=a+10; //trying to modify a value it will generate error
        System.out.println(a);
    }
};

compilation error:- cannot assign a value to final variable a
```

**case 2:-** if you declare instance or static variable with final we should initialize that variable with some values by using constructors or instance blocks otherwise compiler will generate error message.

```
class Test
{
    final int a; //instance variable
    final static int b; //static variable
};
```

The above example will generate error message "variable might not have been initialized"

**Example:-**

- ✓ Final class variables are not a final but final class methods are by default final.
- ✓ Final class methods are by default final because for the final class not possible to create subclasses hence it is not possible to override that method.

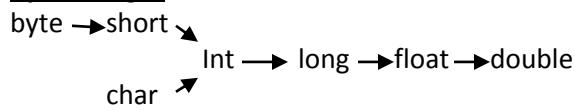
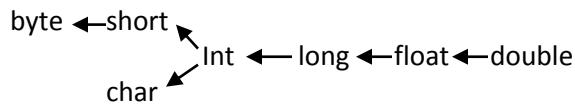
```
final class Test
{
    int a=10;
    void m1()
    {
        System.out.println("m1 method");
        a=a+10;
        System.out.println(a);
    }
    public static void main(String[] args)
    {
        new Test().m1();
    }
};
```

**Type-casting:-**

The process of converting data one type to another type is called type casting.

There are two types of type casting

1. Implicit typecasting /widening/up casting
2. Explicit type-casting (narrowing)/do

**Type casting chart:-****Up-casting :-****down-casting:-**

When we assign higher value to lower data type range then compiler will rise compiler error “possible loss of precision” but whenever we are type casting **higher data type-lower data type** compiler won’t generate error message but we will loss the data.

**Implicit-typecasting:- (widening) or (up casting)**

1. When we assign lower data type value to higher data type that typecasting is called up- casting.
2. When we perform up casting data no data loss.
3. It is also known as up-casting or widening.
4. Compiler is responsible to perform implicit typecasting.

**Explicit type-casting:- (Narrowing) or (down casting)**

1. When we assign a higher data type value to lower data type that type-casting is called down casting.
2. When we perform down casting data will be loss.
3. It is also known as narrowing or down casting.
4. User is responsible to perform explicit typecasting.

**Note :- Parent class reference variable is able to hold child class object but Child class reference variable is unable to hold parent class object.**

```

class Parent
{
}
class Child extends Parent
{
}
Parent p = new Child(); //valid
Child c = new Parent(); //invalid
  
```

**Example :-type casting**

```

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

```

{      public static void main(String[] args)
{          //implicit typecasting (up casting)
    byte b=120;
    int i=b; //#[automatic conversion of byte-int]
    System.out.println(b);
    char ch='a';
    int a=ch;      //#[automatic conversion of char to int]
    System.out.println(a);
    long l1=20;
    float f = l1; //#[automatic conversion of long-float]
    System.out.println(f);
    /*below examples it seems up-casting but compilation error:possible loss of precision
     :conversion not possible
    byte i=100; (1 byte size)
    char ch=i;      (assigned to 2 bytes char)
    System.out.println(ch);
    char ch='a';
    short a=ch;
    System.out.println(a);  compilation error:possible loss of precision
    float f = 10.5f;
    long l = f;
    System.out.println(l);  compilation error:possible loss of precision
    float f=10.5f;
    long l = f;
    System.out.println(l);  compilation error:possible loss of precision (memory
    representation different) */
    //explicit-typecasting (down-casting)
    // converted-type-name var-name = (converted-type-name)conversion-var-type;
    int a1=120;
    byte b1 =(byte)a1;
    System.out.println(b1);
    int a2=130;
    byte b2 =(byte)a2;
    System.out.println(b2);
    float ff=10.5f;
    int x = (int)ff;
    System.out.println(x);
    Parent p = new Child();
    //target-type variable-name=(target-type)source-type;
    Child c1 =(Child)p;
    Parent p = new Child();
    Child c1 = (Child)p;
}
}

```

**Example-2:-**

- In java parent class reference variable is able to hold Child class object but Child class reference variable unable to hold Parent class object.
  - **Parent p = new Child();** ---->valid

- Child c = new Parent(); ---->invalid

```

class Parent
{
    void m1(){System.out.println("parent m1 method");}
}

class Child extends Parent
{
    void m1(){System.out.println("child m1 method");}
    void m2(){System.out.println("child m2 method");}
    public static void main(String[] args)
    {
        //parent class is able to hold child class object
        Parent p1 = new Child(); //creates object of Child class
        p1.m1(); //child m1() will be executed
        //p1.m2(); Compilation error we are unable to call m2() method
        Child c1 =(Child)p1; //type casting parent reference variable to child object.
        c1.m1();
        c1.m2();
    }
}

```

- In above example parent class is able to hold child class object but when you call **p.m1()**; method compiler is checking m1() method in parent class at compilation time. But at runtime child object is created hence Child method will be executed.
- Based on above point decide in above method execution decided at runtime hence it is a runtime polymorphism.
- When you call **p.m2 ()**; compiler is checking m2 () method in parent class since not there so compiler generate error message. Finally it is not possible to call child class m2 () by using parent reference variable even thought child object is created.
- Based on above point we can say by using parent reference it is possible to call only overriding methods (**m1 ()**) of child class but it is not possible to call direct method(**m2()**) of child class.
- To overcome above limitation to call child class method perform typecasting.

**Example:- in java it is possible to override methods in child classes but it is not possible to override variables in child classes.**

```

class Parent
{
    int a=100;
}

class Child extends Parent
{
    int a=1000;
    public static void main(String[] args)
    {
        Parent p = new Child();
        System.out.println("a values is :--->" + p.a); //100
    }
}

```

If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass *hides* the one in the superclass.

```
class Animal
{
    void instanceMethod(){System.out.println("instance method in Animal");}
    static void staticMethod(){System.out.println("static method in Animal");}
}
class Dog extends Animal
{
    void instanceMethod(){System.out.println("instance method in Dog");}//overriding
    static void staticMethod(){System.out.println("static method in Dog");}//hiding
    public static void main(String[ ] args)
    {
        Animal a = new Dog();
        a.instanceMethod();
        a.staticMethod(); // [or] Animal.staticMethod();
    }
}
```

- The version of the overridden instance method that gets invoked is the one in the subclass.
- The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.

The Cat class overrides the instance method in Animal and hides the static method in Animal.

The main method in this class creates an instance of Cat and invokes testClassMethod() on the class and testInstanceMethod() on the instance.

#### **Example :- importance of converting parent class reference variable into child class object**

```
//let assume predefined class
class ActionForm
{
    void xxx(){}
    void yyy(){}
}
class LoginForm extends ActionForm //assume to create LoginForm our class must extends ActionForm
{
    void m1(){System.out.println("LoginForm m1 method");}//method of LoginForm class
    void m2(){System.out.println("LoginForm m2 method");}//method of LoginForm class

    public static void main(String[] args)
    {
        //assume server(Tomcat,glassfish...) is creating object of LoginForm
        ActionForm af = new LoginForm(); //creates object of LoginForm class
        //af.m1();      af.m2(); //by using af it is not possible to call m1() & m2()

        LoginForm lf = (LoginForm)af;//type casting
        lf.m1();
        lf.m2();
    }
}
```

**Example:- method overriding vs. Hierarchical inheritance**

```

class Heroin
{
    int rating( ) { return 0 ; }
};

class Anushka extends Heroin
{
    int rating(){return 1;};
};

class Nazriya extends Heroin
{
    int rating(){return 5;};
};

class Kf extends Heroin
{
    int rating(){return 2;};
};

class Test
{
    public static void main(String[] args)
    {
        /*Heroin h,h1,h2,h3;
        h = new Heroin();
        h1 = new Anushka();
        h2 = new Nazriya();
        h3 = new Kf();*/
        Heroin h = new Heroin();
        Heroin h1 = new Anushka();
        Heroin h2 = new Nazriya();
        Heroin h3 = new Kf();
        System.out.println("Heroin rating     :--->" + h.rating());
        System.out.println("Anushka rating   :--->" + h1.rating());
        System.out.println("Nazsriya rating :--->" + h2.rating());
        System.out.println("Kf rating         :--->" + h3.rating());
    }
};

```

**In above example when you call rating() method compilation time compiler is checking method in parent class(Heroin) but runtime Child class object are created hence child class methods are executed.**

**Example :-**

```

class Animal
{void eat(){System.out.println("animal eat");}
};

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

class Cat extends Animal
{    void eat(){System.out.println("cat eat");}
};

```

```

class Test
{
    public static void main(String[] args)
    {
        Animal a1,a2;
        a1=new Dog();           //creates object of Dog class
        a1.eat();               //compiletime:Animal runtime : Dog
        a2=new Cat();           //creates object of Cat class
        a2.eat();               //compiletime:Animal runtime : Cat
    }
};

```

**Example:-method overriding vs. multilevel inheritance.**

```

class Person
{
    void eat(){System.out.println("4-idly");}
};

class Ratan extends Person
{
    void eat(){System.out.println("10-idly");}
};

class RatanKid extends Ratan
{
    void eat(){System.out.println("2-idly");}
};

class Test
{
    public static void main(String[] args)
    {
        Person p = new Ratan();
        p.eat();           //compiletime: Person runtime:Ratan
        Ratan r = new RatanKid();
        r.eat();           //compiletime: Ratan runtime:RatanKid
        Person p1 = new RatanKid();
        p1.eat();          //compiletime: Person runtime:RatanKid
    }
};

```

**Example :- overriding vs method hiding**

- static method cannot be overridden because static method bounded with class where as instance methods are bounded with object.
- In java it is possible to override only instance methods but not static methods.
- The below example seems to be overriding but it is method **hiding concept**.

```

class Parent
{
    static void m1(){System.out.println("parent m1()");}
};

class Child extends Parent
{
    static void m1(){System.out.println("child m1()");}
    public static void main(String[] args)
    {
        Parent p = new Child();
        p.m1(); //output : parent m1()
    }
};

```

**toString():-**

- **toString()** method present in Object and it is printing String representation of Object.
- **toString()** method return type is String object it means **toString()** method is returning String object.
- **Note :- whenever you are printing reference variable internally **toString()** method is called.**

```
class Object
{
    public String toString()
    {
        return getClass().getName() + '@' + Integer.toHexString(hashCode());
    }
};
```

**Example:-**

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

**Example :- Employee class overriding **toString()** method**

```
class Employee
{
    //instance variables
    String ename;
    int eid;
    double esal;
    Employee(String ename,int eid,double esal)//local variables
    {
        //conversion of local variables to instance variables
        this.ename = ename;           this.eid = eid;           this.esal = esal;
    }
    public String toString()
    {
        return ename+" "+eid+" "+esal;
    }
    public static void main(String[] args)
    {
        Employee e1 = new Employee("ratan",111,60000);
        System.out.println(e1);
        System.out.println(e1.toString());
    }
};
```

employee class is not overriding **toString()**

D:\morn11>java Employee

Employee@530daa

employee class is overriding **toString()**

D:\morn11>java Employee

Ratan 111 60000

Ratan 111 60000

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

**Case 1:-**

- if a class is declared as final, then we cannot inherit that class it means we cannot create child class for that final class.

**Example :-**

```
final class Parent //parent is final class child class creation not possible
{
}
class Child extends Parent //compilation error
{
}
```

**Example :-** Every method present inside a final class is always final but every variable present inside the final class not be final variable.

```
final class Test
{
    int a=10; //not a final variable
    void m1() //final method
    {
        System.out.println("m1 method is final");
        a=a+100;
        System.out.println(a); //110
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1();
    }
}
```

**Case 2:-** If a method declared as a final we can not override that method in child class.

**Example :-**

```
class Parent
{
    final void marry(){}
};
class Child extends Parent
{
    void marry(){}
};
```

**Compilation Error:-** marry() in Child cannot override marry() in Parent  
overridden method is final

**Case 3:-** If a variable declared as a final we can not reassign that variable if we are trying to reassign compiler generate error message.

For the local variables only one modifier is applicable that is final.

**Example:-**

```
class Test
{
    public static void main(String[] args)
    {
        final int a=100 ;//local variables
        a = a+100; // [compilation error because trying to reassignment]
        System.out.println(a);
    }
};
```

**Compilation Error :-** cannot assign a value to final variable a

**Example :-** it is not possible to make default values as a final values.

final variables always needs initialization, if you don't initialize compiler will generate compilation error.

```
class Test
{
    final static int a;
    public static void main(String[] args)
    {
        System.out.println("static variable value="+Test.a);
    }
}
```

**Compilation error:**

variable a might not have been initialized

## Garbage Collector

- Garbage collector is destroying the useless object and it is a part of the JVM.
- There are three ways to make eligible objects to garbage collector

**Example-1 :- Whenever we are assigning null constants to our objects then objects are eligible for GC(garbage collector)**

```
class Test
{
    public static void main(String[] args)
    {
        Test t1=new Test();
        Test t2=new Test();
        System.out.println(t1);
        System.out.println(t2);
        ///////////////
        t1=null;      //t1 object is eligible for Garbage collector
        t2=null;      //t2 object is eligible for Garbage Collector
        System.out.println(t1);
        System.out.println(t2);
    }
};
```

**Example-2 :- Whenever we reassign the reference variable the objects are automatically eligible for garbage collector.**

```
class Test
{
    public static void main(String[] args)
    {
        Test t1=new Test();
        Test t2=new Test();
        System.out.println(t1);
        System.out.println(t2);
        t1=t2;          //reassign reference variable then one object is destroyed.
        System.out.println(t1);
        System.out.println(t2);
    }
};
```

**Example -3:- Whenever we are creating objects inside the methods one method is completed the objects are eligible for garbage collector.**

```
class Test
{
    void m1()
    {
        Test t1=new Test();
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1();
        System.gc();
    }
};
```

**Example :-**

- To call the garbage collector explicitly use `gc()` method it is a static method system class.
- Just before destroying object garbage collector will call `finalize()` method.

```
class Test extends Object
{
    public void finalize()
    {
        System.out.println("object destroyed");
    }
    public static void main(String[] args)
    {
        Test t1 = new Test();
        Test t2 = new Test();
        System.out.println(t1.toString());
        System.out.println(t2.toString());
        ::::::://using object
        t1=null;
        t2=null;
        System.gc();
    }
};
```

**Note :-**

*if any exceptions raised in `finalize()` method those exceptions are ignored & object is destroyed.*

**Abstraction:-**

There are two types of methods in java

- a. Normal methods
- b. Abstract methods

**Normal methods:- (component method/concrete method)**

Normal method is a method which contains method declaration as well as method implementation.

**Example:-**

```
void m1() --->method declaration
{
    body; --->method implementation
}
```

**Abstract methods:-**

- 1) The method which is having only method declaration but not method implementations such type of methods are called abstract Methods.
- 2) In java every abstract method must end with “ ; ”.

**Example :-**      **abstract void m1(); ----→method declaration**

Based on above representation of methods the classes are divided into two types

- 1) Normal classes.
- 2) Abstract classes.

**Normal classes:-**

Normal class is a ordinary java class it contains only normal methods if we are trying to declare at least one abstract method that class will become abstract class.

**Example:-**

```
class Test //normal class
{
    void m1() { body ; } //normal method
    void m2() { body ; } //normal method
    void m3() { body ; } //normal method
};
```

**Abstract class:-**

Abstract class is a java class which contains at least one abstract method(wrong definition).

**If any abstract method inside the class that class must be abstract.**

**Example 1:-**

```
class Test //abstract class
{
    void m1( ){ } //normal method
    void m2( ){ } //normal method
    void m3(); //abstract method
};
```

**Example-2:-**

```
class Test //abstract class
{
    abstract void m1(); //abstract method
    abstract void m2(); //abstract method
    abstract void m3(); //abstract method
};
```

**Abstract modifier:-**

- Abstract modifier is applicable for methods and classes but not for variables.
- To represent particular class is abstract class and particular method is abstract method to the compiler use abstract modifier.
- The abstract class contains declaration of methods it says abstract class partially implement class hence for partially implemented classes object creation is not possible. If we are trying to create object of abstract class compiler generate error message “class is abstract con not be instantiated”

**Example -1:-**

- ❖ **Abstract classes are partially implemented classes hence object creation is not possible.**
- ❖ **For the abstract classes object creation not possible, if you are trying to create object compiler will generate error message.**

```
abstract class Test          //abstract class
{
    abstract void m1();      //abstract method
    abstract void m2();      //abstract method
    abstract void m3();      //abstract method
    void m4(){System.out.println("m4 method");} //normal method
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m4();
    }
};
```

Compilation error:- **Test is abstract; cannot be instantiated**

**Test t = new Test();**

**Example-2 :-**

- Abstract class contains abstract methods for that abstract methods provide the implementation in child classes.
- Provide the implementations is nothing but override the methods in child classes.
- The abstract class contains declarations but for that declarations implementation is present in child classes.

```
abstract class Test
{
    abstract void m1();
    abstract void m2();
    abstract void m3();
    void m4(){System.out.println("m4 method");}
};

class Test1 extends Test
{
    void m1(){System.out.println("m1 method");}
    void m2(){System.out.println("m2 method");}
    void m3(){System.out.println("m3 method");}
}

public static void main(String[] args)
{
    Test1 t = new Test1();
    t.m1();           t.m2();           t.m3();           t.m4();

    Test t1 = new Test1(); //abstract class reference variable Child class object
    t1.m1();         //compile : Test runtime : Test1
    t1.m2();         //compile : Test runtime : Test1
    t1.m3();         //compile : Test runtime : Test1
    t1.m4();         //compile : Test runtime : Test1
}
```

**Example -3 :-**

- Abstract class contains abstract methods for that abstract methods provide the implementation in child classes.
- if the child class is unable to provide the implementation of all parent class abstract methods at that situation declare that class with abstract modifier then take one more child class to complete the implementation of remaining abstract methods.
- It is possible to declare multiple child classes but at final complete the implementation of all methods.

```
abstract class Test
{
    abstract void m1();
    abstract void m2();
    abstract void m3();
    void m4(){System.out.println("m4 method");}
};

abstract class Test1 extends Test
{
    void m1(){System.out.println("m1 method");}
};

abstract class Test2 extends Test1
{
    void m2(){System.out.println("m2 method");}
};

class Test3 extends Test2
{
    void m3(){System.out.println("m3 method");}
    public static void main(String[] args)
    {
        Test3 t = new Test3();
        t.m1();
        t.m2();
        t.m3();
        t.m4();
    }
};
```

**Example-5 :-**

for the abstract methods it is possible to provide any return type(void, int, char, Boolean.....etc)

```
class Emp{};
abstract class Test1
{
    abstract int m1(char ch);
    abstract boolean m2(int a);
    abstract Emp m3();
};

abstract class Test2 extends Test1
{
    int m1(char ch)
    {
        System.out.println("char value is:-"+ch);
        return 100;
    }
};
```

```

class Test3 extends Test2
{
    boolean m2(int a)
    {
        System.out.println("int value is:-"+a);
        return true;
    }
    Emp m3()
    {
        System.out.println("m3 method");
        return new Emp();
    }
    public static void main(String[] args)
    {
        Test3 t=new Test3();
        int a=t.m1('a');
        System.out.println("m1() return value is:-"+a);
        boolean b=t.m2(111);
        System.out.println("m2() return value is:-"+b);
        Emp e = t.m3();
        System.out.println("m3() return value is:-"+e);
    }
}

```

**Example-6:-**

**It is possible to override non-abstract as a abstract method in child class.**

```

abstract class Test
{
    abstract void m1();                                //m1() abstract method
    void m2(){System.out.println("m2 method");}          //m2() normal method
};

abstract class Test1 extends Test
{
    void m1(){System.out.println("m1 method");}          //m1() normal method
    abstract void m2();                                //m2() abstract method
};

class FinalClass extends Test1
{
    void m2(){System.out.println("FinalClass m2() method");}
    public static void main(String[] args)
    {
        FinalClass f = new FinalClass();
        f.m1();
        f.m2();
    }
}

```

**Example:- inside the abstract class it is possible to declare main method.**

```

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

```

**Example-8:-**

- Constructor is used to create object (wrong definition).
- Constructor is executed during object creation to initialize values to instance variables.
- Constructors are used to write the functionality that is executed during object creation.
- There are multiple ways to create object in java but if we are creating object by using "new" then only constructor is executed.

**Note :- in below example abstract class constructor is executed but object is not created.**

**Inside the abstract class it is possible to declare the constructor.**

```
abstract class Test
{
    Test()
    {
        System.out.println("abstract class con");
    }
}
class Test1 extends Test
{
    Test1()
    {
        super();
        System.out.println("normal class con");
    }
    public static void main(String[] args)
    {
        new Test1();
    }
}
D:\>java Test1
abstrac calss con
normal class con
```

**case 1:- [abstract method to normal method]**

```
abstract class Test
{
    abstract void m1();
}
class Test1 extends Test
{
    void m1(){System.out.println("m1 method");}
}
```

**case 2:-[normal method to abstract method]**

```
class Test
{
    void m1(){System.out.println("m1 method");}
}
abstract class Test1 extends Test
{
    abstract void m1();
}
```

**Example-9:- the abstract class allows zero number of abstract method.**

**Definition of abstract class:-**

**Abstract class may contain abstract methods or may not contains abstract methods but object creation is not possible. The below example contains zero number of abstract methods.**

**Ex:- HttpServlet (doesn't contains abstract methods still it is abstract object creation not possible )**

**abstract class Test**

```
{    void cm()      {      System.out.println("ratan");      }
    void pm()      {      System.out.println("anushka");      }
    public static void main(String[] args)
    {        Test t = new Test();
        t.cm(); t.pm();
    }
};
```

*Test.java:6: Test is abstract; cannot be instantiated*

**Abstraction definition :-**

- The process highlighting the set of services and hiding the internal implementation is called abstraction.
- Bank ATM Screens Hiding the internal implementation and highlighting set of services like , money transfer, mobile registration,...etc).
- Syllabus copy of institute just highlighting the contents of java but implementation there in classed rooms .
- We are achieving abstraction concept by using Abstract classes & Interfaces.

**Encapsulation:-**

- ✓ **The process of binding the data and code as a single unit is called encapsulation.**  
Or
- ✓ **The process of hiding the implementation details to user is called encapsulation.** And we are achieving this concept by declaring variables as a private modifier because it is possible to access private members within the class only.

**Example:-**

If the variables are declared as a private it is possible to access those variables only within the class but it is possible to set(update) the data by using setter methods and it is possible to get(read) the data by using getter methods.

The data of the private field can be accessed only by using public setter & getter method. In this way we are hiding implementation to other classes. The setter and getter methods are user defined methods.

**Syntax:-**

```
setXXX()      where xxx=property name
getXXX()      where xxx=property name
```

The setter method return type is always void & getter method return type is always property return type.

**File-1:-**

```
class Encapsulation
{
    private int sid;
    private int sname;
    public void setSid(int x)
    {
        this.sid=sid;
    }
    public void setSname(String sname)
    {
        this.sname=sname;
    }
    public int getSid()
    {
        return sid;
    }
    public String getSname()
    {
        return sname;
    }
};
```

**To access encapsulated use following code:-**

```
class Test
{
    public static void main(String[] args)
    {
        Encapsulation e=new Encapsulation();
        e.setSid(100);
        e.setSname("ratan");
        System.out.println(e.getSid());
        System.out.println(e.getSname());
    }
};
```

**Main Method:-**      **Public static void main(String[] args)**

**Public** → To provide access permission to the jvm declare main with public.

**Static** ---> To provide direct access permission to the jvm declare main is static(with out creation of object able to access main method)

**Void** → don't return any values to the JVM.

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

**String**       $\rightarrow$  it is possible to take any type of argument.

[ ] → represent possible to take any number of arguments.

### ***Modifications on main():-***

- 1) Modifiers order is not important it means it is possible to take **public static** or **static public**.

**Example :-** `public static void main(String[] args)`  
`static public void main(String[] args)`

- 2) the following declarations are valid

the following declarations are valid

**Example:-** static public void main(String[] args)

**Example:** static public void main(String[] args){  
static public void main(String []args,{  
static public void main(String args[ ]) {

- 3) instead of args it is possible to take any variable name (a b c etc)

**Example:-** static public void main(String ratan)

```
static public void main(String... args) {
```

```
static public void main(String... args)
```

- 4) for 1.5 version instead of String[] args it is possible to take String... args(only three dots represent variable argument.)

Example: static public void main(String args)

- E) the applicable modifiers on main method

the applicable modifiers on main method.

- public, **h**, static, **a**, final, **d**, strictfp, **e**, synchronized

**a. public b. static c. final d. strictfp e. synchronized**

In above five modifiers public and static mandatory remaining three modifiers optional  
Example : `public static final String synchronized void main(String args) { }`

Which of the following declarations are valid?

- Which of the following declarations are valid:-

  1. public static void main(String... a) --->valid
  2. final strictfp static void mian(String[] Sravya) --->invalid
  3. static public void mian(String a[])) --->valid
  4. final strictfp public static main(String[] rattaiah) --->invalid
  5. final strictfp synchronized public static void main(String... pandu) --->valid

### ***Strictfp modifier:-***

- a. Strictfp is a modifier applicable for classes and methods.
  - b. If a method is declared as strictfp all floating point calculations in that method will follow IEEE754 standard. So that we will get platform independent results.
  - c. If a class is declared as strictfp then every method in that class will follow IEEE754 standard so we will get platform independent results.

**Ex:-** `strictfp class Test { //methods/// }`  --->all methods fallows IEEE754  
`strictfp void m1() { }`  ---> m1() method fallows IEEE754

**Native modifier:-**

- Native is the modifier applicable only for methods.
- Native method is used to represent particular method implementations there in non-java code (other languages like C,CPP) .
- Native methods are also known as "foreign methods".

**Examples:-**

```
public final int getPriority();
public static native java.lang.Thread currentThread();
```

**Example:-**

```
class Test
{
    final strictfp synchronized static public void main(String...ratan)
    {
        System.out.println("hello ratan sir");
    }
};
```

**Example:-main method VS inheritance****Case-1**

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

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

D:\>java Child  
Child class

**Case-2**

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

class Child extends Parent
{
};
```

D:\>java Child  
Parent class

**Example:-main method VS overloading**

In java it is possible to overload main method but JVM is always calling String[] main method.

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("String[] main method");
        main(100);
    }

    public static void main(int a)
    {
        main('r');
        System.out.println("int main method");
    }

    public static void main(char ch)
    {
        System.out.println("char main method");
    }
};
```

E:\>java Test  
String[] main method  
char main method  
int main method

**note:- it is not possible to override main method because main is a static method.**

**Command Line Arguments:-**

- ✓ The arguments which are passed from command prompt to application at runtime are called command line arguments.
- ✓ The command line argument separator is space.
- ✓ In single command line argument it is possible to provide space by declaring that command line argument in double quotes.

**Example-1 :-**

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

D:\>java Test corejava ratan
corejava
ratan
D:\>java Test core java ratan
core
java
D:\>java Test "core java" ratan
core java
ratan
```

**Var-arg method:-(1.5 version)**

It allows the methods to take any number of parameters to particular method.

**Syntax:-** **Void m1(int... a){-----} (only 3 dots)**

The above m1() is allows any number of parameters.(0 or 1 or 2 or 3.....n)

**Example:-**

```
class Test
{
    void m1(int... a)
    {
        System.out.println("Ratan");
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1();           //int var-arg method executed
        t.m1(10);         //int var-arg method executed
        t.m1(10,20);     //int var-arg method executed
        t.m1(10,20,30); //int var-arg method executed
    }
}
```

**Example:- var-arg VS normal arguments**

For java methods it is possible to provide normal arguments along with variable arguments.

```
class Test
{
    void m1(char ch,int... a)
    {
        System.out.println(ch);
        for (int a1:a)
        {
            System.out.println(a1);
        }
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1('a');
        t.m1('b',10);
        t.m1('c',10,20);
        t.m1('d',10,20,30,40);
    }
}
```

**Note :- inside the method it is possible to declare only one variable-argument and that must be last argument otherwise the compiler will generate compilation error.**

void m1(int... a)	--->valid
void m2(int... a,char ch)	--->invalid
void m3(int... a,boolean... b)	--->invalid
void m4(double d,int... a)	--->valid
void m5(char ch ,double d,int... a)	--->valid
void m6(char ch ,int... a,boolean... b)	--->invalid

**Example:- normal-arguments vs variable-argument**

If the call contains both var-arg method & normal argument method then it prints normal argument value.

```
class Test
{
    void m1(int... a)
    {
        System.out.println("variable argument="+a);
    }
    void m1(int a)
    {
        System.out.println("normal argument="+a);
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1(10);
    }
}
E:\>java Test
normal argument=10
```

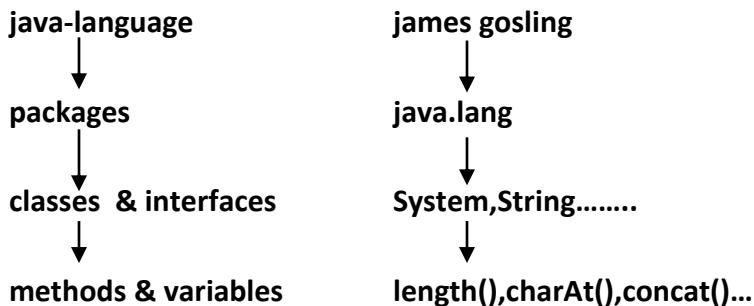
**Example :- var-arg method vs overloading**

```
class Test
{
    void m1(int... a)
    {
        for (int a1 : a)
        {
            System.out.println(a1);
        }
    }
    void m1(String... str)
    {
        for (String str1 : str)
        {
            System.out.println(str1);
        }
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1(10,20,30);           //int var-arg method calling
        t.m1("ratan","Sravya");   //String var-arg calling
        t.m1();//var-arg method vs ambiguity [compilation error ambiguous]
    }
}
```

## Packages

### **java-language:-**

in java James Gosling is maintained predefined support in the form of packages and these packages contains classes & interfaces, and these classes and interfaces contains predefined methods & variables.



### **java source code:-**

- java is a open source software we are able to download it free of cost and we are able to see the source code of the java.
- The source code location **C:\Program Files\Java\jdk1.7.0\_75\src (zip file)** extract the zip file.
- ❖ Java contains 14 predefined packages but the default package in java if **java.lang**. package.

Java.lang	java.beans	java.text	java.sql
Java.io	java.net	java.nio	java.math
Java.util	java.applet	java.rmi	
Java.awt	java.awt	java.security	

**Note : The default package in java is java.lang package.**

**Note : package is nothing but physical folder structure.**

### **Types of packages:-**

There are two types of packages in java

- 1) Predefined packages.
- 2) User defined packages.

### **Predefined packages:**

The predefined packages are introduced by James Gosling and these packages contains predefined classes & interfaces and these class & interfaces contains predefined variables and methods.

*Example:- java.lang, java.io ,java.util.....etc*

### **User defined packages:-**

- ❖ The packages which are defined by user, and these packages contains user defined classes and interfaces.

- ❖ Declare the package by using **package** keyword.

syntax :      **package package-name;**  
example :      **package com.sravya;**

- ❖ Inside the source file it is possible to declare only one package statement and that statement must be first statement of the source file.

**Example-1:valid**

```
package com.sravya;
import java.io.*;
import java.lang.*;
```

**Example-2:Invalid**

```
import java.io.*;
package com.sravya;
import java.io.*;
```

**Example-3:Invalid**

```
import java.io.*;
import java.lang.*;
package com.sravya;
```

**Example-4:Invalid**

```
package com.sravya;
package com.tecs;
```

**some predefined package and it's classes & interfaces:-**

**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.

```
java
```

```
|-----→lang
    |--> String(class)
    |--> StringBuffer(class)
    |--> Object(class)
    |--> Runnable(interface)
    |--> Cloneable(interface)
```

Note:- the default package in the java programming is java.lang package.

**Java.io package:**-The classes which are used to perform the input output operations that are present in the java.io packages.

```
java
```

```
|-----→io
    |--> FileInputStream(class)
    |--> FileOutputStream(class)
    |--> FileReader(class)
    |--> FileWriter(class)
    |--> Serializable(interface)
```

**Java.net package:**-The classes which are required for connection establishment in the network that classes are present in the java.net package.

```
java
```

```
|-----→net
    |--> Socket(class)
    |--> ServerSocket(class)
    |--> URL(class)
    |--> SocketOption(interface)
```

**Package name coding conventions :-(not mandatory but we have to follow)**

- 1) The package name must reflect with organization domain name(**reverse of domain name**).

Domain name:- www.tcs.com

Package name:- Package com.tcs;

- 2) Package name must reflect with project name.

Project name :- bank

package :- Package com.tcs.bank;

- 3) The project name must reflect with project module name.

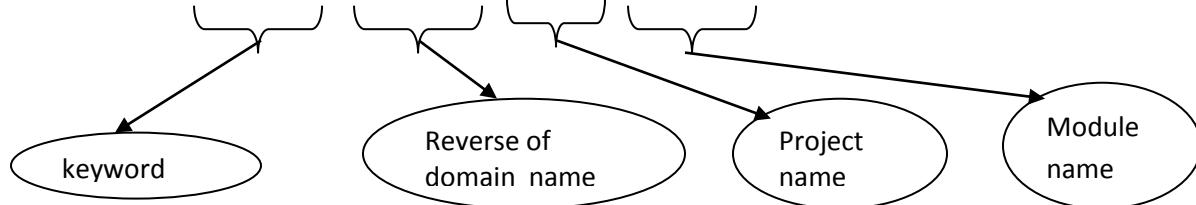
Domain name:- www.tcs.com

Project name:- bank

Module name:- deposit

package name:- Package com.tcs.bank.deposit;

### **Package com.tcs.bank.deposit;**

**Advantages of packages:-**

company name : tcs

project name : bank

**module-1 Deposit**

```

com
|-->tcs
  |-->bank
    |-->deposit
      |-->.class files
  
```

**module-2 withdraw**

```

com
|-->tcs
  |-->bank
    |-->withdraw
      |-->.class files
  
```

**Module-3 moneytranfer**

```

com
|-->tcs
  |-->bank
    |-->moneytranfer
      |-->.class files
  
```

**module-4 accountinfo**

```

com
|-->tcs
  |-->bank
    |-->accountinfo
      |-->.class files
  
```

- 1) It improves parallel development of the project.
- 2) Project maintenance will become easy.
- 3) It improves sharability of the project.
- 4) It improves readability.
- 5) It improves understandability.

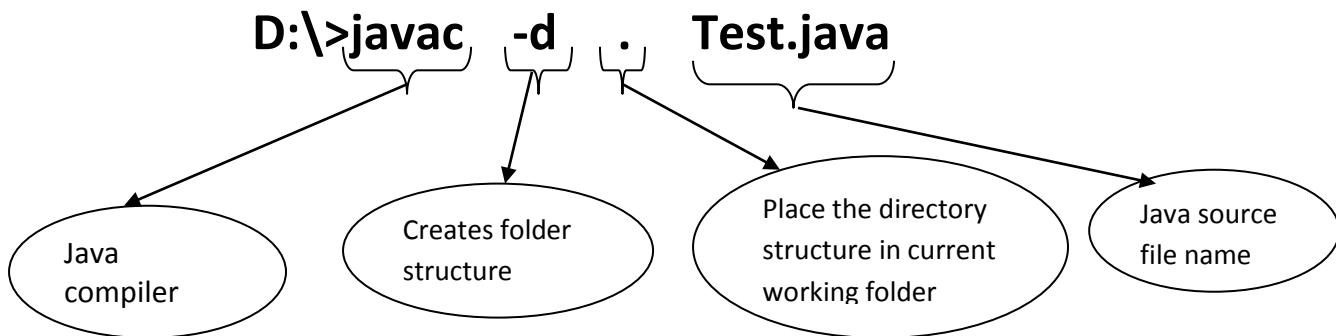
**Note :- In real time the project is divided into number of modules that each and every module is nothing but package statement.**

**Example-1:-****Step-1: write the application with package statement.**

```
package com.sravya.java.corejava;
class Test
{
    public static void main(String[] args)
    {
        System.out.println("package first example");
    }
}
class A
{
}
class B
{
}
interface It
{}
```

**Step-2: compilation process**

If the source file contains the package statement then compile that application by using following command.

**Step-3:- folder Structure.**

```
com
|-->sravya
    |-->java
        |-->corejava
            |-->Test.class
            |-->A.class
            |-->B.class
            |-->It.class
```

**Step-4:-execution process.**

Execute the .class file by using fully qualified name(**class name with complete package structure**)

```
java com.sravya.java.corejava.Test
output : package first example
```

**Example-2:-****Error-1 :-**

- ❖ If it is a predefined package or user defined package Whenever we are using other package classes then must import that package by using import statement.
- ❖ If the application required two classes (System, String) then We are able to import the classes in two ways
  - **Importing all classes.**  
Import java.lang.\*;
  - **Importing application required classes**  
Import java.lang.System;  
Import java.lang.String;

In above two approaches second approach is recommended because it is importing application required classes.

**Error-2:-**

- ❖ Whenever we are using other package classes then that classes must be public otherwise compiler generate error message.

**Error: class is not public we are unable to access outside package.**

**Public modifier:-**

- ✓ Public modifier is applicable for variables, methods, classes.
- ✓ All packages are able to access public members.

**Default modifier:-**

- It is applicable for variables, methods, classes.
- We are able to access default members only within the package and it is not possible to access outside package .
- Default access is also known as package level access.
- The default modifier in java is default.

**Error-3:-**

- ❖ Whenever we are using other package class member that members also must be public.

**Note : When we declare class as public the corresponding members are not public, if we want access public class members that members also must be public.**

**File-1: Sravya.java**

```
package com.sravya.states.info;
public class Sravya
{
    public void ts(){System.out.println("jai telengana");}
    public void ap(){System.out.println("jai andhra");}
    public void others(){System.out.println("jai jai others");}
}
```

**File-2: Tcs.java**

```
package com.tcs.states.requiredinfo;
import com.sravya.states.info.*;
class Tcs
{    public static void main(String[] args)
    {        Sravya s = new Sravya();
        s.ts();           s.ap();           s.others();
    }
}
```

E:\>javac -d . Sravya.java

**compilation of Sravya**

E:\>javac -d . Tcs.java

**compilation of Tcs**

E:\>java com.tcs.states.requiredinfo.Tcs  
jai telengana  
jai andhra  
jai jai others

**execution of Tcs**

**Private modifier:-**

- ✓ private modifier applicable for methods and variables.
- ✓ We are able to access private members only within the class and it is not possible to access even in child classes.

```
class Parent
{
    private int a=10;
}
class Child extends Parent
{
    void m1()
    {
        System.out.println(a); //a variables is private Child class unable to access
    }
    public static void main(String[] arghs)
    {
        Child c = new Child();
        c.m1();
    }
};
```

**error: a has private access in Parent**

**Note :- the most accessible modifier in java is public & most restricted modifier in java is private.**

**Protected modifier:-**

- ✓ Protected modifier is applicable for variables,methods.
- ✓ We are able to access protected members within the package and it is possible to access outside packages also but only indirect child classes & it is not possible to call even in indirect child classes.
- ✓ But in 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.

**A.java:-**

```
package app1;
public class A
{
    protected int fee=1000;
};
```

**B.java:-**

```
package app2;
import app1.*;
public class B extends A
{
    public static void main(String[] args)
    {
        B b = new B();
        System.out.println(b.fee);
    }
};
```

**Summary of variables:-**

<u>modifier</u>	<u>Private</u>	<u>no-modifier</u>	<u>protected</u>	<u>public</u>
<b>Same class</b>	yes	yes	yes	yes
<b>Same package sub class</b>	no	yes	yes	yes
<b>Same package non sub class</b>	no	yes	yes	yes
<b>Different package sub class</b>	no	no	yes	yes
<b>Different package non sub class</b>	no	no	no	yes

**Example:- method overriding rule-7**

- ✓ In java while overriding it is possible to maintain same level(**default-default**) permission or increasing order(**default-public**) but it is not possible to decrease(**public-default**) the permission.
- ✓ In java if we are trying to decrease the permission compiler will generate error message "attempting to assign weaker access privileges"

<u>Parent-class method</u>	<u>child-class method</u>	
<b>Default</b>	default (same level) protected , public (increasing level ) Private (decreasing level)	-->valid --> valid --> invalid
<b>Public</b>	public (same level) Default,private,protected(decreasing)	--> valid -->invalid
<b>Protected</b>	protected(same level) Public(increasing permission) Default,private (decreasing level)	--> valid -->valid -->invalid

**Case 1:- same level [default-default]**

```
class Parent
{
    void m1(){System.out.println("m1 method");}
}
class Child extends Parent
{
    void m1(){System.out.println("m1 method");}
}
```

**Case 2:- increasing permission [protected-public]**

```
class Parent
{
    protected void m1(){System.out.println("m1 method");}
```

```

};

class Child extends Parent
{
    public void m1(){System.out.println("m1 method");}
}

```

**Case3 :- decreasing permission [public-protected]**

```

class Parent
{
    public void m1(){System.out.println("m1 method");}
}

class Child extends Parent
{
    protected void m1(){System.out.println("m1 method");}
}

```

**Example :-****Test.java:-**

```

package app1;
public class Test
{
    public void m1() { System.out.println("app1.Test class m1()"); }
}

```

**X.java:-**

```

package app1.corejava;
public class X
{
    public void m1() { System.out.println("app1.core.X class m1()"); }
}

```

**Y.java:-**

```

package app1.corejava.advjava;
public class Y
{
    public void m1() { System.out.println("app1.corejava.advjava.Y class m1()"); }
}

```

**Z.java:-**

```

package app1.corejava.advjava.structs;
public class Z
{
    public void m1() { System.out.println("app1.corejava.advjava.structs.Z class m1()"); }
}

```

**Ratan.java:-**

```

import app1.Test;
import app1.corejava.X;
import app1.corejava.advjava.Y;
import app1.corejava.advjava.structs.Z;
class Ratan
{
    public static void main(String[] args)
    {
        Test t = new Test(); t.m1();
        X x = new X(); x.m1();
        Y y = new Y(); y.m1();
        Z z = new Z(); z.m1();
    }
}

```

**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 of a particular class directly to the application without using class name.
- a. **import static java.lang.System.\*;**

The above line is used to call all the static members of System class directly into application without using class name.

**Ex:-without static import**

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

**Ex :- with static import**

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

**Example:-**

```
package com.dss.java.corejava;
public class Sravya
{
    public static int fee=1000;
    public static void course()
    {
        System.out.println("core java");
    }
    public static void duration()
    {
        System.out.println("1-month");
    }
    public static void trainer()
    {
        System.out.println("ratan");
    }
};
```

**without static import**

```
package com.tcs.course.coursedetails;
import com.dss.java.corejava.*;
class Tcs
{
    public static void main(String[] args)
    {
        System.out.println(Sravya.fee);
        Sravya.course();
        Sravya.duration();
        Sravya.trainer();
    }
}
```

**with static import**

```
package com.tcs.course.coursedetails;
import static com.dss.java.corejava.Sravya.*;
class Tcs
{
    public static void main(String[] args)
    {
        System.out.println(fee);
        course();
        duration();
        trainer();
    }
}
```

**Example:-**

When you import main package we are able to access only package classes, it is not possible to access sub package classes, if we want sub package classes must import sub packages also.

```
com
|-->sravya
    |--->A.class
    |--->B.class
    |--->C.class
    |--->ratan
        |--->D.class
```

In above example when we import **com.sravya.\*** it is possible to access only three classes(A,B,C) but it is not possible to access sub package classes (**ratan package D class**) if we want sub package classes must import sub package(**import com.sravya.ratan.\***).

**File-1: A.java**

```
package jav.corejava;
public class A
{
    public void m1()
    {System.out.println("core java World!");
    }
}
```

**Package structure:-**

```
jav
|-->corejava
    |--->A.class
```

**File-2: B.java:**

```
package jav.corejava.advjava;
public class B
{
    public void m1()
    {System.out.println("Adv java World!");
    }
}
```

```
jav
|-->corejava
    |--->A.class
    |--->advjava
        |--->B.class
```

**File-3: C.java:-**

```
package jav.corejava.advjava.structs;
public class C
{
    public void m1()
    {System.out.println("Structs World!");
    }
}
```

```
jav
|-->corejava
    |--->A.class
    |--->advjava
        |--->B.class
        |--->structs
            |--->C.class
```

**File-4:- MainTest.java****Package structure :-**

```
import jav.corejava.A;
import jav.corejava.advjava.B;
import jav.corejava.advjava.structs.C;
class MainTest
{
    public static void main(String[] args)
    {
        new A().m1();
        new B().m1();
        new C().m1();
    }
}
```

**Example :-**

in java it is not possible to use predefined package names as a user defined packages. If we are trying to use predefined package names as a user defined packages at runtime JVM will generate securityException.

```
package java.lang;
class Test
{
    public static void main(String[] args)
    {
        System.out.println("Ratan World!");
    }
}
class A
{
};

D:\DP>javac -d . Test.java
D:\DP>java java.lang.Test
Exception in thread "main" java.lang.SecurityException: Prohibited package name: java.lang
```

**Applicable modifiers on constructors:-**

- 1) Public
- 2) Private
- 3) Protected
- 4) default

**Private constructors:-**

```
class Parent
{
    private Parent(){ } //private constructor
}
class Child extends Parent
{
    Child()
    {super();} //we are calling parent class private constructor it is not possible
};
D:\>javac Test.java
Test.java:6: Parent() has private access in Parent
```

**we are achieving singleton class creation by using private constructors in java:-**

- when we declare constructor with private modifier we can't create object outside of the class.
- Singleton class allows to create only one object of particular class and we are achieving singleton class creation by using private constructors.
- In some scenarios it is appropriate to have exactly one instance of class like,
  - Window manager
  - File systems
  - Project manager
  - Admin
 These type of objects are called singleton objects.

**file-1:-**

```
package com.dss.st;
class Test
```

```
{     public static Test t=null;
    private Test(){}
    public static Test getInstance()
    {
        if (t==null)
        {
            t = new Test();
        }
        return t;
    }
    public void disp()
    {
        System.out.println("this is ratan singleton class");
    }
};
```

**File-2:-**

```
Package com.dss;
Import com.dss.st.Test;
class Test1
{
    public static void main(String[] args)
    {
        //Test t = new Test(); compilation error Test() has private access in Test
        Test t1 = Test.getInstance();
        Test t2 = Test.getInstance();
        System.out.println(t1.hashCode());//31168322
        System.out.println(t2.hashCode());//31168322
    }
};
```

## Interfaces

1. Interface is also one of the type of class it contains only abstract methods. And Interfaces not alternative for abstract class it is extension for abstract classes.
2. The interface allows to declare only abstract methods and these methods are by default public & abstract if we are declaring or not.
3. The interface is highlighting set of functionalities but implementations are hiding.
4. For the interfaces also compiler will generates .class files after compilation.
5. 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**

```
interface It1
{
    *****
}
```

- ✓ if you don't know anything about implementation just we have the requirement specification then declare that requirements by using interface.
- ✓ If u know the implementation but not completely then we should go for abstract classes.
- ✓ if you know the implementation completely then we should go for concrete classes.

### 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.

### Example-1 :-

- Interface constrains abstract methods and by default these methods are "public abstract".
- Interface contains abstract method for these methods provide the implementation in the implementation classes.
- Implementation class is nothing but the class which implements particular interface.
- While providing implementation of interface methods that implementation methods must be public methods otherwise compiler generate error message "attempting to assign weaker access privileges".

```
interface it1
{
    Void m1();      //abstract method by default [public abstract]
    Void m2();      //abstract method by default [public abstract]
    Void m3();      //abstract method by default [public abstract]
}
Class Test implements it1           //Test is implementation class of It1 interface
{
    Public void m1()            //implementation method must be public
    {
        System.out.println("m1-method implementation ");
    }
    Public void m2()
}
```

```

    {
        System.out.println("m2-method implementation");
    }
Public void m3()
{
    System.out.println("m3 -method implementation");
}
Public static void main(String[] args)
{
    Test t=new Test();
    t.m1();
    t.m2();
    t.m3();
}
}

```

**Example :-**

```

interface It1          //abstract
{
    void m1();         //public abstract
    void m2();
    void m3();
}
class Test implements It1
{
    public void m1(){System.out.println("m1 method");}
    public void m2(){System.out.println("m2 method");}
    public void m3(){System.out.println("m3 method");}

    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
        t.m2();
        t.m3();

        It1 i = new Test();
        i.m1(); //compile : It1 runtime : Test
        i.m2(); //compile : It1 runtime : Test
        i.m3(); //compile : It1 runtime : Test
    }
};

```

**Example-2:-**

- Interface contains abstract method for these methods provide the implementation in the implementation class.
- If the implementation class is unable to provide the implementation of all abstract methods then declare implementation class with abstract modifier & complete the remaining abstract method implementation in next created child classes.
- If the child class also unable to provide implementation then declare the child class with abstract modifier & take one more child class to complete the implementations.
- In java it is possible to take any number of child classes but at final complete the implementation of all abstract methods.

```

interface It1
{   void m1();      //public abstract
    void m2();
    void m3();
}
abstract class Test implements It1
{   public void m1(){System.out.println("m1 method");}
};
abstract class Test1 extends Test
{   public void m2(){System.out.println("m2 method");}
};
class Test2 extends Test1
{   public void m3(){System.out.println("m3 method");}
    public static void main(String[] args)
    {       Test2 t = new Test2();
            t.m1();
            t.m2();
            t.m3();
    }
};

```

**Example:-** The interface reference variables is able to hold child class objects.

```

interface It1           // interface declaration
{   void m1();      //abstract method by default [public abstract]
    void m2();      //abstract method by default [public abstract]
    void m3();      //abstract method by default [public abstract]
}
//Test1 is abstract class contains 2 abstract methods m2() m3() hence object creation not possible
abstract class Test1 implements It1
{   public void m1()
{   System.out.println("m1 method");
};

//Test2 is abstract class contains 1 abstract method m3() hence object creation not possible
abstract class Test2 extends Test1
{   public void m2()
{   System.out.println("m2 method");
};

//Test3 is normal class because it contains only normal methods hence object creation possible
class Test3 extends Test2
{   public void m3()
{   System.out.println("m3 method");
    }
    public static void main(String[] args)
    {       It1 t = new Test3();
            t.m1();      t.m2();      t.m3();
            Test1 t1 = new Test3();
            t1.m1();      t1.m2();      t1.m3();
    }
};

```

```
Test2 t2 = new Test3();
t2.m1();      t2.m2();      t2.m3();
}

interface It1
{
    void m1();
}
interface It2
{
    void m2();
}
interface It3 extends It1,It2
{
    void m3();
}
class Test implements It1
{
    1 method
};
class Test implements It1,It2
{
    2 methods
};
class Test implements It1,It2,It3
{
    3 methods
};
```

**Difference between abstract classes & interfaces:-****Abstract class**

- 1) The purpose of abstract class is to specify default functionality of an object and let its sub classes explicitly implement that functionality. It stands it is providing abstraction layer that must be extended and implemented by the corresponding sub classes.
- 2) An abstract class is a class that declared with **abstract** modifier.

Ex: ***abstract class A***

```
{     abstract void m1(); }
```

- 3) The abstract allows declaring both abstract & concrete methods.
- 4) Abstract class methods must declare with abstract modifier.
- 5) If the abstract class contains abstract methods then write the implementations in child classes.
- 6) In child class the implementation methods need not be public it means while overriding it is possible to declare any valid modifier.
- 7) The abstract class is able to provide implementations of interface methods.
- 8) One java class is able to extends only one abstract class at a time.
- 9) Inside abstract class it is possible to declare main method & constructors.
- 10) It is not possible to instantiate abstract class.
- 11) For the abstract classes compiler will generate **.class** files.
- 12) The variables of abstract class need not be **public static final**.

**Interface**

1. It is providing complete abstraction layer and it contains only declarations of the project then write the implementations in implementation classes.
2. Declare the interface by using **interface** keyword.
- Ex:- ***interface It1***  

```
{     void m1(); }
```
3. The interface allows declaring only abstract methods.
4. Interface methods are by default **public abstract**.
5. The interface contains abstract methods write the implementations in implementation classes.
6. In implementation class the implementation methods must be public.
7. The interface is unable to provide implementation of abstract class methods.
8. One java class is able to implements multiple interfaces at a time.
9. Inside interface it is not possible to declare methods and constructors.
10. It is not possible to instantiate interfaces.
11. For the interfaces compiler will generate **.class** files.
12. The variables declared in interface by default **public static final**.

**Example-1:-**

- Inside the interfaces it is possible to declare variables and methods.
- By default interface methods are **public abstract** and by default interface variables are **public static final**.
- The final variables are replaced with their values at compilation time only.

**Application before compilations:- (.java file)**

```
interface It1           //interface declaration
{
    int a=10;          //interface variables
    void m1();         //interface method
}
class Test implements It1
{
    public void m1()
    {
        System.out.println("m1 method");
        System.out.println(a);
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
    }
};
```

**Application after compilation:- (.class file)**

```
interface It1
{
    public abstract void m1(); // compiler generate public abstract
    public static final int a = 10; // public static final generated by compiler
}
class Test implements It1
{
    public void m1()
    {
        System.out.println("m1 method");
        System.out.println(10); // a is final variable hence it replaced at compilation time only
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
    }
};
```

**Message.java:-**

```
package com.sravya.declarations;
public interface Message
{
    void morn();
    void even();
    void gn();
}
```

**Helper.java:-**

```
package com.sravya.helper;
import com.sravya.declarations.Message;
public abstract class Helper implements Message
{
    public void gn(){System.out.println("good night from helper class");}
}
```

**TestClient1.java:-**

```
package com.sravya.client;
import com.sravya.declarations.Message;
class TestClient1 implements Message
{
    public void morn(){System.out.println("good morning");}
    public void even(){System.out.println("good evening");}
    public void gn(){System.out.println("good night");}
    public static void main(String[] args)
    {
        TestClient1 t = new TestClient1();
        t.morn();
        t.even();
        t.gn();
    }
}
```

**TestClient2.java:-**

```
package com.sravya.client;
import com.sravya.helper.Helper;
class TestClient2 extends Helper
{
    public void morn(){System.out.println("good morning");}
    public void even(){System.out.println("good evening");}
    public static void main(String[] args)
    {
        TestClient2 t = new TestClient2();
        t.morn();
        t.even();
        t.gn();
    }
}
D:\>javac -d . Message.java
D:\>javac -d . Helper.java
D:\>javac -d . TestClient1.java
D:\>javac -d . TestClient2.java
```

```
D:\>java com.sravya.client.TestClient1  
good morning  
good evening  
good 9t
```

```
D:\>java com.sravya.client.TestClient2  
good morning  
good evening  
good night from helper class
```

Real time usage of packages:-

**Message.java:-**

```
package com.dss.declarations;  
public interface Message  
{  
    void msg1();  
    void msg2();  
}
```

**BusinessLogic.java:-**

```
package com.dss.businesslogics;  
import com.dss.declarations.Message;  
public class BusinessLogic implements Message  
{  
    public void msg1(){System.out.println("i like you");}  
    public void msg2(){System.out.println("i miss you");}  
}
```

**TestClient.java:-**

```
package com.dss.client;  
import com.dss.businesslogics.BusinessLogic;  
class TestClient  
{  
    public static void main(String[] args)  
    {  
        BusinessLogic b = new BusinessLogic();  
        b.msg1();  
        b.msg2();  
        Message b1 = new BusinessLogic();  
        b1.msg1();  
        b1.msg2();  
    }  
}
```

**Interfaces vs. inheritance :-****Example :-**

```

interface it1      //it contains 2 methods m1() & m2()
{
    public abstract void m1();
    public abstract void m2();
}

interface it2 extends it1      // it contains 4 methods m1() & m2() & m3() & m4()
{
    public abstract void m3();
    public abstract void m4();
}

interface it3 extends it2      // it contains 6 methods m1() & m2() & m3() & m4() & m5() & m6
{
    public abstract void m5();
    public abstract void m6();
}

interface it4 extends it3      // it contains 7 methods m1() & m2() & m3() & m4() & m5() & m6 & m7()
{
    public abstract void m7();
}

```

**Case 1:-**

```

class Test implements it1
{
    provide the implementation of 2 methods      m1() & m2()
};

```

**Case 2:-**

```

class Test implements it2
{
    provide the implementation of 4 methods      m1() & m2() & m3() & m4()
};

```

**Case 3:-**

```

class Test implements it3
{
    provide the implementation of 6 methods      m1() & m2() & m3() & m4() & m5() & m6()
};

```

**Case 4:-**

```

class Test implements it4
{
    provide the implementation of 7 methods m1() & m2() & m3() & m4() & m5() & m6() & m7()
};

```

**Case 6:-**

```

class Test implements it1,it2 //one class is able to implements multiple interfaces
{
    provide the implementation of 4 methods      m1() & m2() & m3() & m4()
};

```

**Case 7:-**

```

class Test implements it1,it3
{
    provide the implementation of 6 methods      m1() & m2() & m3() & m4() & m5() & m6()
};

```

**Case 8:-**

```

class Test implements it2,it3
{
    provide the implementation of 6 methods      m1() & m2() & m3() & m4() & m5() & m6()
};

```

**Case 9:-**

```

class Test implements it2,it4
{
    provide the implementation of 7 methods      m1() & m2() & m3() & m4() & m5() & m6 & m7()
};

```

```
};
```

**Case 10:-**

```
class Test implements it1,it2,it3
{
    provide the implementation of 6 methods      m1() & m2() & m3() & m4() & m5() & m6
};
```

**Case 11:-**

```
class Test implements it1,it2,it3,it4
{
    provide the implementation of 7 methods      m1() & m2() & m3() & m4() & m5() & m6 & m7()
};
```

**Nested interfaces:-**

**Example :- declaring interface inside the class is called nested interface.**

```
class A
{
    interface it1 //nested interface declared in A class
    {
        void add();
    }
};

class Test implements A.it1
{
    public void add()
    {
        System.out.println("add method");
    }

    public static void main(String[] args)
    {
        Test t=new Test();    t.add();
    }
};
```

**Example :- it is possible to declare interfaces inside abstract class also.**

```
abstract class A
{
    abstract void m1();

    interface it1 //nested interface declared in A class
    {
        void m2();
    }
};

class Test implements A.it1
{
    public void m1()
    {
        System.out.println("m1 method");
    }

    public void m2()
    {
        System.out.println("m2 method");
    }

    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1(); t.m2();
    }
};
```

**Ex:- declaring interface inside the another interface is called nested interface.**

```
interface it2
{
    void m1();

    interface it1
    {
        void m2();
    }
};

class Test2 implements it2.it1
```

```

{     public void m1()
    {         System.out.println("m1 method");      }
    public void m2()
    {         System.out.println("m2 method");      }
    public static void main(String[] args)
    {         Test2 t=new Test2();
        t.m1(); t.m2();
    }
};


```

**Marker interface :-**

- An interface that has no members (methods and variables) is known as marker interface or tagged interface or ability interface.
- In java whenever our class is implementing marker interface our class is getting some capabilities that are power of marker interface. We will discuss marker interfaces in detail in later classes.

Ex:- **serializable , Cloneable , RandomAccess...etc**

**Note: - user defined empty interfaces are not a marker interfaces only, predefined empty interfaces are marker interfaces.**

**Cloneable:-**

- 1) The process of creating exactly duplicate object is called cloning.
- 2) We can create a duplicate object only for the cloneable classes .
- 3) We can create cloned object by using clone()
- 4) The main purpose of the cloning is to maintain backup.

```

class Test implements Cloneable
{
    int a=10,b=20;
    public static void main(String[] args) throws CloneNotSupportedException
    {
        Test t1 = new Test();//creates object of Test class
        Test t2 = (Test)t1.clone();//duplicate object of Test class
        System.out.println(t1.a);
        System.out.println(t1.b);
        t1.b=555;
        t1.a=444;
        System.out.println(t1.a);
        t1.b=333;
        System.out.println(t1.a);
        System.out.println(t1.b);
        //if we want initial values use duplicate object
        System.out.println(t2.a);//10
        System.out.println(t2.b);//20
    }
}

```

**Possibility of extends & implements keywords:-**

**class extends class**  
**interface extends interface**  
**class implements interface**

<b>class A extends B</b>	<b>---&gt;valid</b>
<b>class A extends B,C</b>	<b>---&gt;Invalid</b>
<b>class A extends A</b>	<b>---&gt;Invalid</b>
<b>class A implements It</b>	<b>---&gt;valid</b>
<b>class A implements It1,It2</b>	<b>---&gt;valid</b>
<b>class A extends It</b>	<b>---&gt;Invalid</b>
<b>interface It1 extends It2</b>	<b>---&gt;valid</b>
<b>interface It1 extends It2,It3</b>	<b>---&gt;valid</b>
<b>interface It1 extends A</b>	<b>---&gt;invalid</b>
<b>interface It1 implements It2</b>	<b>---&gt;invalid</b>
<b>interface It1 extends It1</b>	<b>---&gt;invalid</b>
<b>class A extends B implements It1,It2</b>	<b>---&gt;valid [extends first]</b>
<b>class A implements It1,It2 extends B</b>	<b>---&gt;Invalid</b>

**Adaptor class:-**

It is an intermediate class between the interface and user defined class. And it contains empty implementation of interface methods.

**Example:-**

```

interface It          // interface
{
    void m1();
    void m2();
    .....
    void m100();
}
class X implements It //adaptor class
{
    public void m1(){}
    public void m2(){}
    .....
    public void m100(){}
};
//user defined class implementing interface
class Test implements It
{
    must provide 100 methods impl
};
//user defined class extending Adaptor class(X)
class Test extends X
{
override required methods because already X contains empty implementations
};

```

**Adaptor class realtime usage:-**

**Message.java:-** **interface**  
 package com.dss.declarations;  
 public interface Message  
 { void morn(); // by default public abstract  
 void eve();  
 void night();  
 }

**HelperAdaptor.java:-** **Adaptor class**

package com.dss.helper;  
 import com.dss.declarations.Message;  
 public class HelperAdaptor implements Message  
 { public void night(){}
 public void eve(){}
 public void morn(){}
 }

**GoodStudent.java:-**

package com.dss.bl;  
 import com.dss.declarations.Message;  
 public class GoodStudent implements Message  
 { public void morn(){System.out.println("good morning ratan");}
 public void eve(){System.out.println("good evening ratan");}
 public void night(){System.out.println("good nightratan");}
 }

**TestClient.java:-**

package com.dss.client;  
 import com.dss.bl.GoodStudent;  
 import com.dss.bl.Student;  
 class TestClient  
 { public static void main(String[] args)
 { GoodStudent s = new GoodStudent();
 s.eve();s.morn();s.night();

Student s1 = new Student();
 s1.morn();

}

**com**

**/-->dss**

- /--->declarations**
- / | -->Message.class**
- /-->helper**
- / | --->HelperAdaptor.class**
- /-->bl**
- / | -->GoodStudent.class**
- / | -->Student.class**
- /-->client**
- /-->TestClient.class**

### ***String manipulations***

- 1) **Java.lang.String**
- 2) **Java.lang.StringBuffer**
- 3) **Java.lang.StringBuilder**
- 4) **Java.util.StringTokenizer**

#### **Java.lang.String:-**

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

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

        String str1=new String("ratan");
        System.out.println(str1);

        char[] ch={'r','a','t','a','n'};
        String str3=new String(ch);
        System.out.println(str3);

        char[] ch1={'a','r','a','t','a','n','a'};
        String str4=new String(ch1,1,5);
        System.out.println(str4);

        byte[] b={65,66,67,68,69,70};
        String str5=new String(b);
        System.out.println(str5);

        byte[] b1={65,66,67,68,69,70};
        String str6=new String(b1,2,4);
        System.out.println(str6);
    }
}
```

#### **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.*

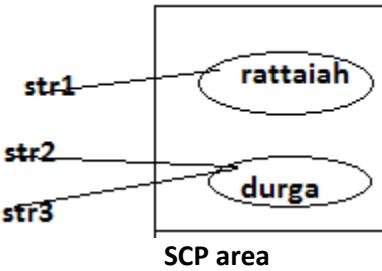
- 1) *Without using new operator*      **String str="ratan";**
- 2) *By using new operator*      **String str = new String("ratan");**

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

**StringBuffer sb = new StringBuffer("sravyainfotech");**

**Creating a string object without using new operator :-**

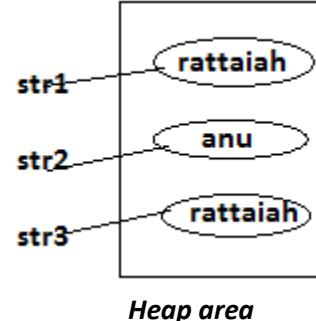
- When we create String object without using new operator the objects are created in SCP (String constant pool) area.
- ```
String str1="rattaiah";
String str2="Sravya";
String str3="Sravya";
```



- When we create object without using new operator 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.

**Creating a string object by using new operator**

- Whenever we are creating String object by using new operator the object created in heap area.
- ```
String str1=new String("rattaiah");
String str2 = new String("anu");
String str3 = new String("rattaiah");
```



- 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 = "ratan";
        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("ratansoft");
        System.out.println(sb);
    }
}
```

**== operator :-**

- ❖ It is comparing reference type and it returns Boolean value as a return value.
- ❖ If two reference variables are pointing to same object then it returns true otherwise false.

**Example:-**

```
class Test
{
    public static void main(String[] args)
    {
        Test t1 = new Test();
        Test t2 = new Test();
        Test t3 = t1;
        System.out.println(t1==t2);      //false
        System.out.println(t1==t3);      //true

        String str1="ratan";
        String str2="ratan";
        System.out.println(str1==str2); //true

        String s1 = new String("anu");
        String s2 = new String("anu");
        System.out.println(s1==s2);      //false

        StringBuffer sb1 = new StringBuffer("sravya");
        StringBuffer sb2 = new StringBuffer("sravya");
        System.out.println(sb1==sb2);   //flase
    }
}
```

**Case : `toString()`**

- `toString()` method present in object class it returns a string representation of object(**classname@hashcode**).
- `String` is child class of `Object` and it is overriding `toString()` to return content of the `String` object.
- `StringBuffer` is child class of `Object` and it is overriding `toString()` to return content of the `StringBuffer` object.

**Note :- whenever we are printing any type of reference variable in java internally it is calling `toString()` method .**

```
class Object
{
    public java.lang.String toString()
    {
        return getClass().getName() + '@' + Integer.toHexString(hashCode());
    }
}
```

```
class String extends Object
{
    //overriding method
    public java.lang.String toString()
    {
        return "content of String";
    }
};

class StringBuffer extends Object
{
    //overriding method
    public java.lang.String toString()
    {
        return "content of String";
    }
};
```

**Example:-**

```
class Test
{
    public static void main(String[] args)
    {
        //object class toString() executed
        Test t = new Test();
        System.out.println(t);
        System.out.println(t.toString());

        //String class toString() executed
        String str="ratan";
        System.out.println(str);
        System.out.println(str.toString());

        //StringBuffer class toString() executed
        StringBuffer sb = new StringBuffer("anu");
        System.out.println(sb);
        System.out.println(sb.toString());
    }
};
```

In above example when we call `toString()` method on `Test` class reference type then first it will check `toString()` in `Test` class since not available it will execute `Object` class `toString()`.

D:\>java Test

Test@530daa      Test@530daa    ratan   ratan    anu    anu

**Case 3:- immutability vs mutability**

*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); //ratan

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

**Concat() :-**

➤ Concat() method is combining two String objects and it is returning new String object.

**public java.lang.String concat(java.lang.String);**

**Example :-**

```
class Test
{
    public static void main(String[] args)
    {
        String str="ratan";
        String str1 = str.concat("soft"); //concat() method return String object.
        System.out.println(str);
        System.out.println(str1);
    }
}
```

**Creation of user defined immutability class:-**

- ✓ Once we create the object, it is not allowing modifications that behavior is called immutability.
- ✓ Java contains number of immutable classes like String, wrapper classes(Integer, Long, Float...etc).
- ✓ We can achieve immutability nature of the class by using following properties,

**1) Declare the fields final & private**

Once we declare the fields private it is not possible to access outside class & if we declare final modifications are not allowed.

**2) Declare the class with final**

Once we declare class with final it is not possible to create child classes it prevents method overriding in sub-classes.

**3) Set the data by using parameterized constructor**

Set the data to properties by using constructor but don't use setter methods because setter method meant for data change in object.

**File-1:Emp.java**

```
class Emp
{
    private final String ename;
    Emp(String ename)
    {
        this.ename=ename;
    }
    public String getName()
    {
        return ename;
    }
};
```

**File 2:Test.java**

```
class Test
{
    public static void main(String[] args)
    {
        Emp e = new Emp("ratan");
        System.out.println(e.getName());
    }
}
```

In above example Emp class immutable class because,

- Emp class properties are final we are unable to change the value after creating object.
- Emp class if final we are unable create sub-class & it is not possible to override the methods.
- We are using constructor it assign the values only once.
- There is no setter method hence we have no option to change the value of variable.

**Case 4:-****Internal implementation equals() method:-**

- equals( ) method present in object used for reference comparison & return Boolean value.
  - If two reference variables are pointing to same object returns true otherwise false.
- String is child class of object and it is overriding equals( ) methods used for content comparison.
  - If two objects content is same then returns true otherwise false.
- StringBuffer class is child class of object and it is not overriding equals() method hence it is using parent class(Object) equals() method used for reference comparison.
  - If two reference variables are pointing to same object returns true otherwise false.

```
class Object
{
    public boolean equals(java.lang.Object)
    {
        // reference comparison;
    }
}
class String extends Object
{
    //String class is overriding equals() method
    public boolean equals(java.lang.Object);
    {
        //content comparison;
    }
}
class StringBuffer extends Object
{
    //not overriding hence it is using parent class equals() method
    //reference comparison;
}
```

**Example :-**

```

class Test
{
    Test(String str) { }
    public static void main(String[] args)
    {
        Test t1 = new Test("ratan");
        Test t2 = new Test("ratan");
        //Object class equals() method executed (reference comparison)
        System.out.println(t1.equals(t2));

        String str1 = new String("Sravya");
        String str2 = new String("Sravya");
        //String class equals() method executed (content comparison)
        System.out.println(str1.equals(str2));

        StringBuffer sb1 = new StringBuffer("anu");
        StringBuffer sb2 = new StringBuffer("anu");
        //StringBuffer class equals() executed (reference comparison)
        System.out.println(sb1.equals(sb2));
    }
}

```

**== operator vs equals() :-**

- In above example we are completed equals() method.
- == operator used to check reference variables & returns boolean ,if two reference variables are pointing to same object returns true otherwise false.

```

class Test
{
    Test(String str){}
    public static void main(String[] args)
    {
        Test t1 = new Test("ratan");
        Test t2 = new Test("ratan");
        System.out.println(t1==t2);      //reference comparison  false
        System.out.println(t1.equals(t2)); //reference comparison  false

        String str1="anu";
        String str2="anu";
        System.out.println(str1==str2); //reference comparison  true
        System.out.println(str1.equals(str2)); //content comparison  true

        String str3 = new String("Sravya");
        String str4 = new String("Sravya");
        System.out.println(str3==str4);      //reference comparison  false
        System.out.println(str3.equals(str4)); //content comparison  true

        StringBuffer sb1 = new StringBuffer("students");
        StringBuffer sb2 = new StringBuffer("students");
        System.out.println(sb1==sb2);      //reference comparison  false
        System.out.println(sb1.equals(sb2)); //reference comparison  false
    }
}

```

**Example :- String identity vs String equality**

```

class Test
{
    public static void main(String[] args)
    {
        String str1 = "hello";
        String str2 = "hello";
        String str3= new String("hello");
        //identity checking
        System.out.println(str1==str2);          //true
        System.out.println(str1==str3);          //false
        System.out.println(str1==str3);          //false
        //equality checking
        System.out.println(str1.equals(str2));    //true
        System.out.println(str1.equals(str3));    //true
        System.out.println(str2.equals(str3));    //true
    }
}

```

**Java.lang.String class methods:-****1) CompareTo() & compareTolgnoreCase():-**

- 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
  - if the two strings are equal then it return zero.
  - If the first string first character Unicode value is bigger than second string first character Unicode value then it return +ve value.
  - If the first string first character Unicode value is smaller than second string first character Unicode value then it return -ve value.

**`compareTo()` method comparing two string with case sensitive.**

**`compareTolgnoreCase()` method comparing two strings character by character by ignoring case.**

```

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
        System.out.println("ratan".compareToCase("RATAN")); //0
    }
}

```

**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`

**cahrAt(int) & split() & trim():-**

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

`public char charAt(int);`

**Split(String):-** By using split() method we are dividing string into number of tokens.

`public java.lang.String[] split(java.lang.String);`

**trim():-** trim() is used to remove the trail and leading spaces this method always used for memory saver.

`public java.lang.String trim();`

```
class Test
{
    public static void main(String[] args)
    {
        //cahrAt() 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);
        //split() method
        String s="hi rattaiah how r u";
        String[] str1=s.split(" ");
        for(String str2 : str1)
        {
            System.out.println(str2);
        }
        //trim()
        String ss="      ratan      ";
        System.out.println(ss.length());//7
        System.out.println(ss.trim());//ratan
        System.out.println(ss.trim().length());//5
    }
}
```

**replace() & toUpperCase() & toLowerCase():-**

`public java.lang.String replace(Stirng str, String str1);-`

`public java.lang.String replace(char, char);`

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

`public java.lang.String toLowerCase();`

`public java.lang.String toUpperCase();`

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 how 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());
    }
}

```

**endsWith() & startsWith() & substring():-**

- **endsWith()** is used to find out if the string is ending with particular character/string or not.
- **startsWith()** used to find out the particular String starting with particular character/string or not.  
`public boolean startsWith(java.lang.String);`  
`public boolean endsWith(java.lang.String);`
- **substring()** used to find substring in main String.  
`public java.lang.String substring(int); int = starting index`  
`public java.lang.String substring(int, int); int=starting index to int =ending index`  
while printing substring() it includes starting index & it excludes ending index.

**Example:-**

```

class Test
{
    public static void main(String[] args)
    {
        String str="rattaiah how r u";
        System.out.println(str.endsWith("u"));           //true
        System.out.println(str.endsWith("how"));         //false
        System.out.println(str.startsWith("d"));         //false
        System.out.println(str.startsWith("r"));         //true

        String s="ratan how r u";
        System.out.println(s.substring(2));               //tan how r u
        System.out.println(s.substring(1,7));             //atan h
        System.out.println("ratansoft".substring(2,5));  //tan
    }
}

```

**StringBuffer class methods:-****reverse() & delete() & deleteCharAt():-**

```
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():-**

```
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);
    }
}
```

**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);
    }
}
```

**Java.lang.StringBuilder:-**

- 1) Introduced in jdk1.5 version.
- 2) StringBuilder is identical to StringBuffer except for one important difference.
- 3) Every method present in the StringBuilder is not Synchronized means that is not thread safe.
- 4) multiple threads are allow to operate on StringBuilder methods hence the performance of the application is increased.

 **StringTokenizer:-**

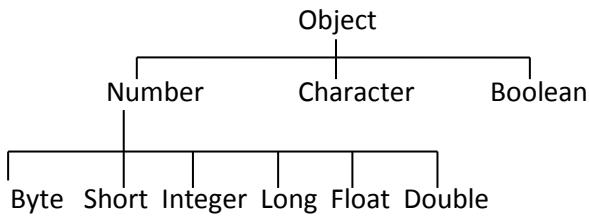
```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        String str="hi ratan w r u wt bout anushka";
        StringTokenizer st = new StringTokenizer(str);//split the string with by default (space symbol)
        while (st.hasMoreElements())
        {
            System.out.println(st.nextElement());
        }

        //used our string to split give String
        String str1 = "hi,rata,mf,sdfsdf,ara";
        StringTokenizer st1 = new StringTokenizer("hi,rata,mf,sdfsdf,ara", ",");
        while (st1.hasMoreElements())
        {
            System.out.println(st1.nextElement());
        }
    }
}
```

## 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');
  
```

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

**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**).

### 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.

**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
    }
}

```

**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);
    }
}
```

**XxxValue():-** it is used to convert wrapper object into corresponding primitive value.

**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);
    }
}
```

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

**Example :-**

```
class Test
{
    public static void main(String[] args)
    {
        String str1="100";
        String str2="100";
        System.out.println(str1+str2);

        //parseXXX() converion of String to primitive type
        int a1 = Integer.parseInt(str1);
        float a2 = Float.parseFloat(str2);
        System.out.println(a1+a2);
    }
}
```

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**.

#### **Example:-**

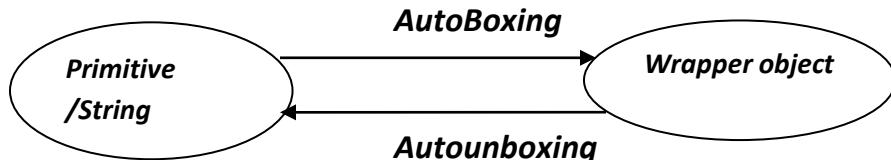
```

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);
    }
}

```

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



**Factory method:-**

- ❖ One java class method returns same class object or different class object is called factory method.
- ❖ There are three types of factory methods in java.
  - **Instance factory method.**
  - **Static factory method.**
  - **Pattern factory method.**
- ❖ The factory is called by using class name is called static factory method.
- ❖ The factory is called by using reference variable is called instance factory method.
- ❖ One java class method is returning different class object is called pattern factory method.

**Example:-**

```

class Test
{
    public static void main(String[] args)
    {
        //static factory method
        Integer i = Integer.valueOf(100);
        System.out.println(i);

        Runtime r = Runtime.getRuntime();
        System.out.println(r);

        //instance factory method
        String str="ratan";
        String str1 = str.concat("soft");
        System.out.println(str1);

        String s1="sravyainfotech";
        String s2 = s1.substring(0,6);
        System.out.println(s2);

        //pattern factory method
        Integer a1 = Integer.valueOf(100);
        String ss = a1.toString();
        System.out.println(ss);

        StringBuffer sb = new StringBuffer("ratan");
        String sss = sb.toString();
        System.out.println(sss);
    }
}

```

### **Java .io package**

- ✓ Java.io package contains classes to perform input and output operations.
- ✓ By using java.io package we are performing file handling java.
- ✓

#### **Stream:-**

*Stream is nothing but sequence of data, and it is called as stream means stream of water continuous flow.*

*There are three streams are automatically created*

- System.in standard input stream
- System.out standard output stream
- System.error standard error stream

```
import java.io.*;
class Test
{
    public static void main(String[] args) throws IOException
    {
        System.out.println("output stream");
        System.err.println("error stream");

        int a = System.in.read();
        System.out.println((char)a);
    }
}
```

#### **I/O Streams:-**

- Byte Streams handle I/O of raw binary data.
- Character Streams handle I/O of character data, automatically handling translation to and from the local character set.
- Buffered Streams optimize input and output by reducing the number of calls to the native API.
- Scanning and Formatting allows a program to read and write formatted text.
- I/O from the Command Line describes the Standard Streams and the Console object.
- Data Streams handle binary I/O of primitive data type and String values.
- Object Streams handle binary I/O of objects.

**I/O Streams:-**

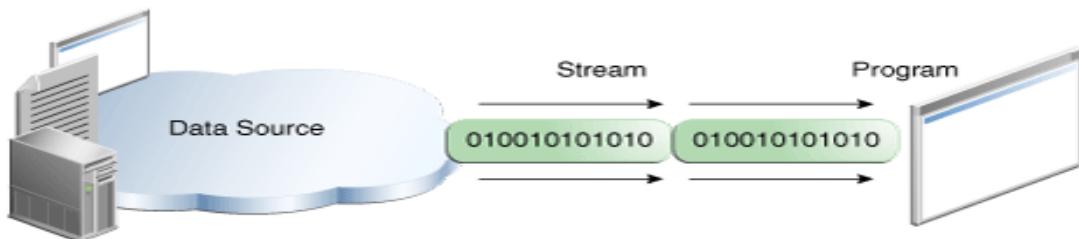
An I/O stream represents input source or an output destination. A stream represent many kind of source and destination like disk files & devices & memory arrays.

Streams support different kind of data

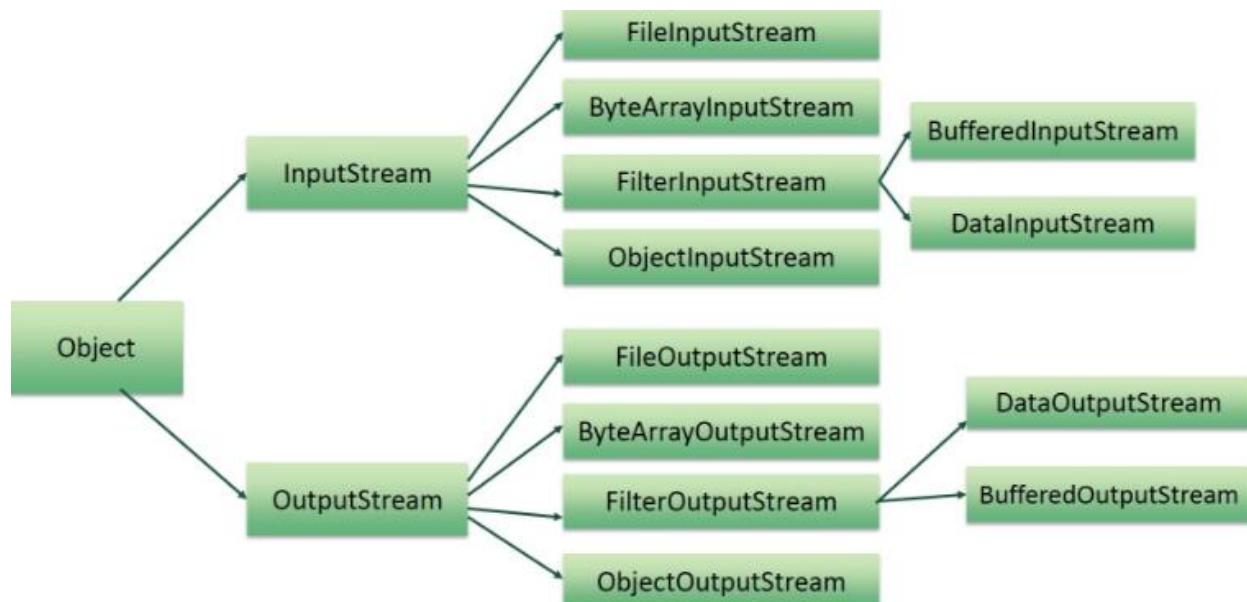
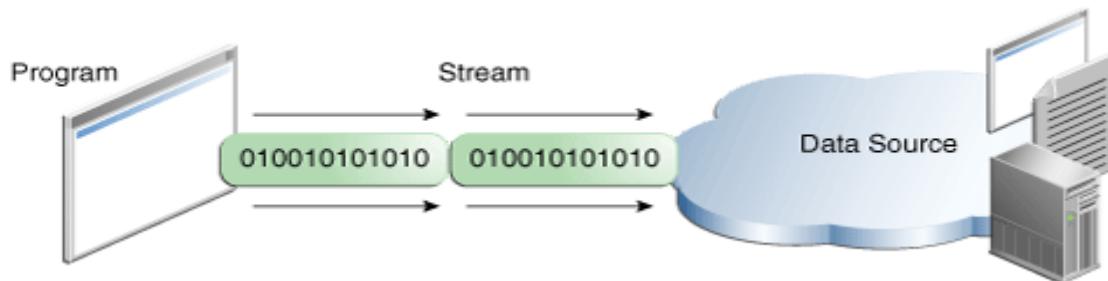
- Simple bytes.
- Primitive data types
- Localized characters
- Objects ....etc

Stream is a communication channel between source and destination & A stream is a sequence of data.

**Input stream:-** Program uses Input stream to read the data from a source one item at a time.



**Output stream:-** Program uses output stream to write the data to a destination one item at a time.



**Example :- creation of physical File**

```
import java.io.*;
class Test
{
    public static void main(String[] args) throws IOException
    {
        File f = new File("anu.txt");
        boolean b = f.createNewFile();
        if (b)
        {
            System.out.println("File is created successfully");
        }
        else
        {
            System.out.println("File is already existed in location");
        }
    }
}
```

**Example : creation of directory**

```
iimport java.io.*;
class Test
{
    public static void main(String[] args) throws IOException
    {
        //creation of File
        File f = new File("anu.txt");
        System.out.println(f.exists());
        f.createNewFile();
        System.out.println(f.exists());

        //creation of directory
        File f1 = new File("durga");
        System.out.println(f1.exists());
        f1.mkdir();
        System.out.println(f1.exists());

        //creation of file inside the directory (directory must present)
        File f2 = new File("durga","durga.txt");
        f2.createNewFile();
    }
}
```

**Byte streams:-**

Program uses byte stream to perform input & output of byte data. All byte stream classes developed based on *InputStream* & *OutputStream*.

Program uses byte stream to perform input and output of 8-bit bytes.

To demonstrate how the byte stream works file I/O provided two main classes

- ✓ *FileInputStream*
  - It is used to read the data from source one item at a time.
  - To read the data from source use *read()* method of *FileInputStream* class.  
**public int read() throws java.io.IOException;**  
*read()* method returns first character Unicode value in the form of integer value.
  
- ✓ *FileOutputStream*
  - It is used to write the data to destination one item at a time.
  - To write the data to destination use *write()* method of *FileOutputStream* class.  
**public void write(int unicode) throws java.io.IOException;**  
*write()* method is taking Unicode value of the character as a parameter.

**Steps to design application:-**

Step 1 :- create the channel.

Step 2:- read the data from source file.

Step 3:- store the data in some variable temporarily.

Step 4:- check the stream is ended or data (data flow completed or not).

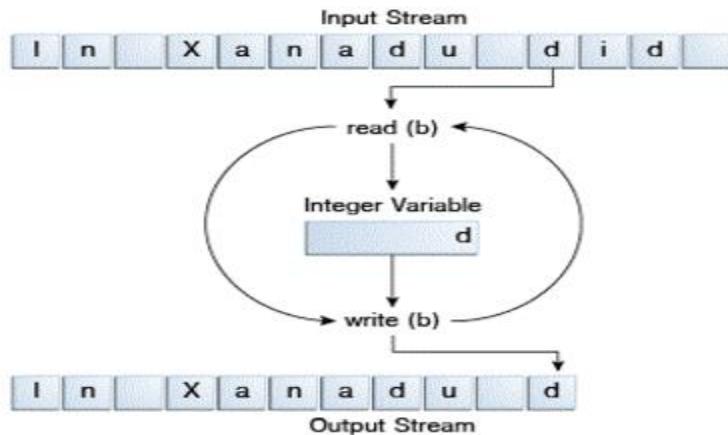
Step 5:- write the data to destination.

Step 6:- close the streams.

**Example :-**

```
import java.io.*;
class Test
{
    public static void main(String[] args) throws FileNotFoundException, IOException
    {
        //Byte oriented channel creation
        FileInputStream fis = new FileInputStream("abc.txt");
        FileOutputStream fos = new FileOutputStream("xyz.txt");
        int c;
        while((c=fis.read())!=-1)
        {
            System.out.print((char)c);
            fos.write(c);
        }
        System.out.println("read() & write operations are completed");
        //stream closing operations
        fis.close();
        fos.close();
    }
}
```

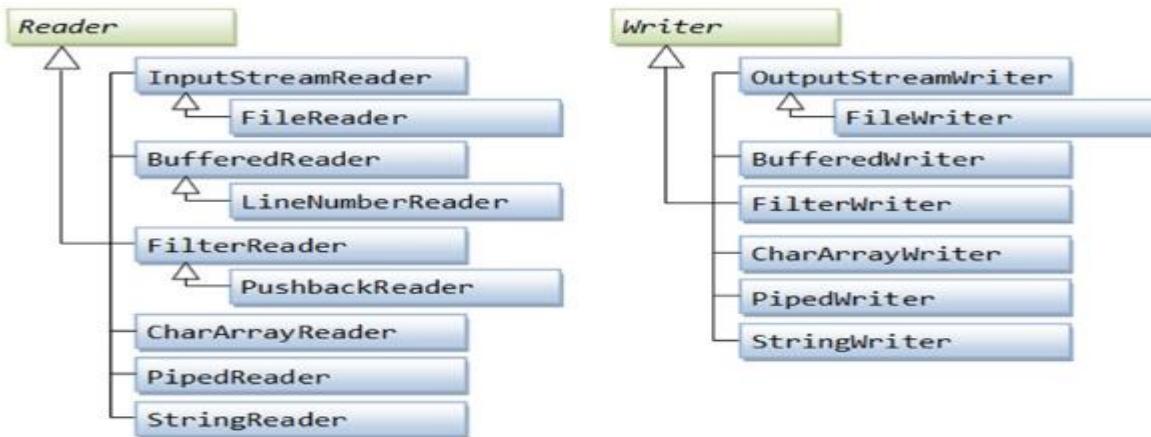
While working with streams we will get two exceptions mainly `FileNotFoundException`, `IOException` & these two exceptions are checked exceptions so must handle these exception by using try-catch blocks or throws keyword.



Simple byte stream input and output.

#### Limitation of byte streams:-

- ❖ It reads the data only one byte at a time hence it takes more time to copy.
- ❖ Must close the streams always.
- ❖ These streams always hitting hard disk to retrieve the data it reduces the performance.



#### Character streams:-

Program uses character stream to perform input & output of character data. All character stream classes developed based on Reader & Writer classes.

To demonstrate how the character stream works file I/O provided two main classes

- ✓ `FileReader`
  - It is used to read the data from source one item at a time.
  - To read the data from source use `read()` method of `InputStream` class.

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

`read()` method returns first character Unicode value in the form of integer value.

- ✓ `FileWriter`
  - It is used to write the data to destination one item at a time.
  - To write the data to destination use `write()` method of `OutputStream` class.

**public void write(int unicode) throws java.io.IOException;**  
write() method is taking Unicode value of the character as a parameter.

**Example :-**

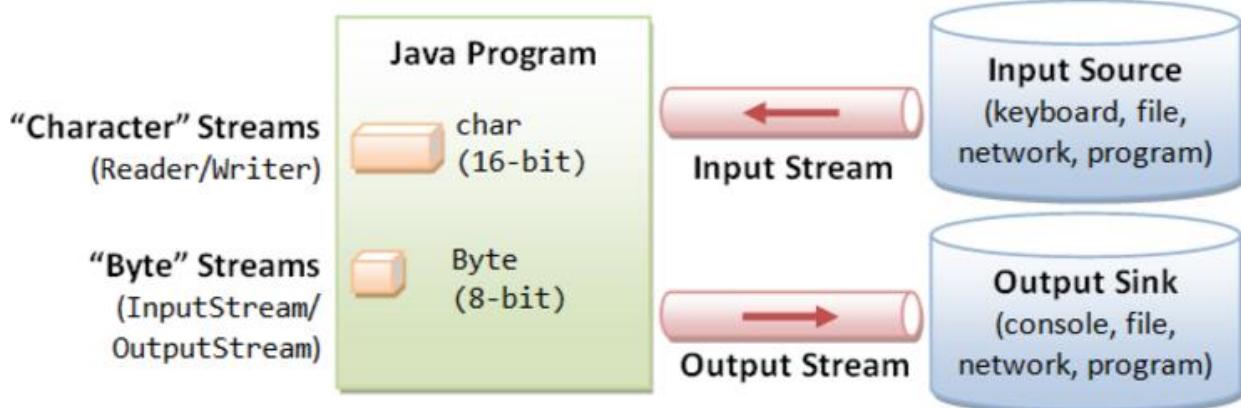
```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
public class Test
{ public static void main(String[] args) throws IOException
    { FileReader inputStream = null;
    FileWriter outputStream = null;
    try {
        inputStream = new FileReader("abc.txt");
        outputStream = new FileWriter("characteroutput.txt");

        int c;
        while ((c = inputStream.read()) != -1)
            { outputStream.write(c);
            }
        }

        finally
        { if (inputStream != null)
            { inputStream.close();
            }
        if (outputStream != null)
            { outputStream.close();
            }
        }
    }
}
```

*Note : In CopyCharacters, the int variable holds a character value in its last 16 bits; in CopyBytes, the int variable holds a byte value in its last 8 bits.*

*CopyCharacters is very similar to CopyBytes. The most important difference is that CopyCharacters uses FileReader and FileWriter for input and output in place of FileInputStream and FileOutputStream. Notice that both CopyBytes and CopyCharacters use an int variable to read to and write from. However, in CopyCharacters, the int variable holds a character value in its last 16 bits; in CopyBytes, the int variable holds a byte value in its last 8 bits.*



**CharArrayWriter:-** It is used to write the data to multiple files & this class implements Appendable interface.

In below example we are adding same data to four files.

```
import java.io.*;
class Test
{
    public static void main(String[] args) throws IOException
    {
        CharArrayWriter ch = new CharArrayWriter();
        FileReader fr = new FileReader("abc.txt");
        int a;
        while((a=fr.read())!=-1)
        {
            ch.write(a);
        }
        FileWriter fw1 = new FileWriter("a.txt");
        FileWriter fw2 = new FileWriter("b.txt");
        FileWriter fw3 = new FileWriter("c.txt");
        FileWriter fw4 = new FileWriter("d.txt");
        ch.writeTo(fw1);
        ch.writeTo(fw2);
        ch.writeTo(fw3);
        ch.writeTo(fw4);

        fw1.close();
        fw2.close();
        fw3.close();
        fw4.close();
        fr.close();
        System.out.println("operations are completed");
    }
}
```

**Line oriented I/O:-**

In above two streams(byte & character) it is possible to read only one item at time it increases number of read & write operations hence the performance is decreased.

To overcome above limitation to improve performance of the application instead of reading the data item by item, read the data line-by-line format to improve the performance.

To perform line oriented operations use two classes.

✓ **BufferedReader**

- It is used to read the data from source file in line by line format.
- To read the data use `readLine()` method of `BufferedReader` class .

**`public java.lang.String readLine() throws java.io.IOException;`**

The `readLine()` method returns first line of the text file in the form of `String`.

✓ **PrintWriter**

- It is used to write the data to destination file in line by line format.
- To write the data to file use `println()` method of `PrintWriter` class.

**`public void println(java.lang.String);`**

The above method used to write the data to destination file.

**Example :-**

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines {
    public static void main(String[] args) throws IOException {
        BufferedReader inputStream = null;
        PrintWriter outputStream = null;
        try {
            inputStream = new BufferedReader(new FileReader("ratan.txt"));
            outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));
            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

**Buffered Streams:-**

- ✓ In previous examples we are using un-buffered I/O .This means each read and write request is handled directly by the underlying OS.
  - ✓ In normal streams each request directly triggers disk access it is relatively expensive & performance is degraded.
- To overcome above limitations use buffered streams.
- Bufferd input stream read the data from buffered memory and it interacting with hard disk only when buffered memory is empty.
  - Buffered output stream write the data to buffer memory.

There are four buffered stream classes.

**Buffered byte streams,**

1. *BufferedInputStream*
2. *BufferedOutputStream*

A program can convert an un buffered stream into buffered streams.

```
new BufferedInputStream(new FileInputStream("xanadu.txt"));
new BufferedOutputStream(new FileOutputStream("byteoutput.txt"));
```

**Buffered Character streams,**

3. *BufferedReader*
4. *BufferedWriter*

A program can convert an un buffered stream into buffered streams.

```
new BufferedReader(new FileReader("ratan.txt"));
new BufferedWriter(new FileWriter("characteroutput.txt"));
```

**Example:-**

```
import java.io.*;
class Test
{
    public static void main(String[] args)
    {
        BufferedReader br;
        BufferedWriter bw;
        try{
            br=new BufferedReader(new FileReader("Test1.java"));
            bw=new BufferedWriter(new FileWriter("States.java"));
            String str;
            while ((str=br.readLine())!=null)
            {
                bw.write(str);
            }
            br.close();
            bw.close();
        }
        catch(Exception e)
        {
            System.out.println("getting Exception");
        }
    }
}
```

**Example:-**

```

import java.io.*;
class Test
{
    public static void main(String[] args)
    {
        BufferedInputStream bis;
        BufferedOutputStream bos;
        try{
            bis=new BufferedInputStream(new FileInputStream("abc.txt"));
            bos=new BufferedOutputStream(new FileOutputStream("xyz.txt"));
            int str;
            while ((str=bis.read())!=-1)
            {
                bos.write(str);
            }
            bis.close();
            bos.close();
        }
        catch(Exception e)
        {
            System.out.println(e);
            System.out.println("getting Exception");
        }
    }
}

```

**I/O from the Command Line by using Console class:-**

- ✓ The console class used to get the data from console and it is introduced in 1.5 version present in java.io package.
- ✓ It provides the methods to read text data and password.

```

import java.io.*;
class Test
{
    public static void main(String[] args)
    {
        Console c = System.console();

        System.out.println("enter your name:");
        String name = c.readLine();
        System.out.println("welcome to="+name);

        System.out.println("enter password");
        char[] ch = c.readPassword();
        System.out.println("password is="+ch);
        String pwd = String.valueOf(ch);
        System.out.println("password is="+pwd);
    }
};

```

**Serialization:-**

The process of saving an object to a file (or) the process of sending an object across the network is called serialization.

But strictly speaking the process of converting the object from java supported form to the network supported form of file supported form.

To do the serialization we required following classes

1. FileOutputStream
2. ObjectOutputStream

**Deserialization:-**

The process of reading the object from file supported form or network supported form to the java supported form is called deserialization.

We can achieve the deserialization by using following classes.

1. FileInputStream
2. ObjectInputStream

```
import java.io.*;
class Emp implements Serializable
{
    int eid;
    String ename;
    Emp(int eid,String ename)
    {this.eid=eid;
     this.ename=ename;
    }
    public static void main(String[] args)throws Exception
    {
Emp e = new Emp(111,"ratan");
```

**//serialization [write the object to file]**

```
FileOutputStream fos = new FileOutputStream("xxxx.txt");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(e);
System.out.println("serialization completed");
```

**//deserialization [read object form text file]**

```
FileInputStream fis = new FileInputStream("xxxx.txt");
ObjectInputStream ois = new ObjectInputStream(fis);
Emp e1 = (Emp)ois.readObject(); //returns Object
System.out.println(e1.eid+"----"+e1.ename);
System.out.println("de serialization completed");
}
```

**Transient Modifiers :-**

Transient modifier is the modifier applicable for only variables and we can't apply for methods and classes.

- At the time of serialization, if we don't want to save the values of a particular variable to meet security constraints then we should go for transient modifier.
- At the time of serialization JVM ignores the original value of transient variable and default value will be serialized

```
import java.io.*;
class Emp implements Serializable
{
    transient int eid;
    transient String ename;
}
0---null
```

## Exception Handling

### **Introduction:-**

- ❖ **Dictionary meaning of the exception is abnormal termination.**
- ❖ **An exception is an event that occurs during execution of the program that disturbs normal flow of the program instructions.**
- ❖ **If the application contains exception then the program terminated abnormally then the rest of the application is not executed.**

To overcome above limitation in order to execute the rest of the application & to get normal termination of the application must handle the exception.

There are two ways to handle the exceptions in java.

- 1) By using try-catch block.
- 2) By using throws keyword.

### **Reasons for exceptions:-**

- ✓ opening a non-existing file.
- ✓ Network connection problems.
- ✓ Values are out of range values
- ✓ End user input mistakes.....etc

### **Exception Handling:-**

- ✓ **The main objective of exception handling is to get normal termination of the application in order to execute rest of the application code.**
- ✓ **Exception handling means just we are providing alternate code to continue the execution of remaining code & to get normal termination of the application.**

Every Exception is a predefined class present in different packages.

<code>java.lang.ArithmaticException</code>	<code>java.lang</code>
<code>java.io.IOException</code>	<code>java.io</code>
<code>java.sql.SQLException</code>	<code>java.sql</code>
<code>javax.servlet.ServletException</code>	<code>javax.servlet</code>

### **Types of Exceptions:-**

As per the sun micro systems standards The Exceptions are divided into three types

- 1) Checked Exception
- 2) Unchecked Exception
- 3) Error

### **Checked Exception:-**

- ✓ The Exceptions which are checked by the compiler at the time of compilation are called Checked Exceptions.  
Examples:- `IOException,SQLException,InterruptedException,ClassNotFoundException.....etc`
- ✓ If the application contains checked Exception the compiler is able to check it and it will give intimation to developer regarding Exception in the form of compilation error.
- ✓ Handle the checked Exception in two ways
  - By using try-catch block.
  - By using throws keyword.

In our application whenever we are using exceptional methods the code is not compiled because these methods throws checked exception hence must handle the exception by using try-catch or throws keywords. And no need to remember the methods just use the method compile it then compiler is saying exception information handle it.

**Note:** - If application contains checked Exception then compile time just compiler is displays exception information for handling **but the exception raised at runtime** if the required resources are not available.

#### Checked Exception scenarios:-

##### 1) **java.lang.InterruptedException**

When we used **Thread.sleep(2000)**; your thread is entered into sleeping mode then other threads are able to interrupt the program is terminated abnormally & rest of the application is not executed.

To overcome above problem compile time compiler is checking that exception & displaying exception information in the form of compilation error.

Based on compiler generated error message write the try-catch blocks or throws , if runtime any exception raised the try-catch or throws keyword executed program is terminated normally.

##### 2) **Java.io.FileNotFoundException**

If we are trying to read the file from local disk but at runtime if the file is not available program is terminated abnormally rest of the application is not executed.

To overcome above problem compile time compiler is checking that exception & displaying exception information in the form of compilation error.

Based on compiler generated error message write the try-catch blocks or throws , if runtime any exception raised the try-catch or throws keyword executed program is terminated normally.

##### 3) **Java.sql.SQLException**

If we are trying to connect to data base but at runtime data base is not available program is terminated abnormally rest of the application is not executed.

**Note:** In above scenarios compile time compiler is display just exception information but exception raised at runtime.

#### Example:-

```
import java.io.*;
class Test
{
    public static void main(String[] args)
    {
        File f = new File("abc.txt");
        FileReader fr = new FileReader(f);
    }
}
```

If you are trying to compile the above compilation the compiler will show the compilation error.

G:\>javac Test.java

**Test.java:5: error: unreported exception FileNotFoundException; must be caught or declared to be thrown**

**FileReader fr = new FileReader(f);**

**Unchecked Exception:-**

- ❖ The exceptions which are not checked by the compiler at the time of compilation are called unchecked Exception.  
ArithmException, ArrayIndexOutOfBoundsException, NumberFormatException....etc
- ❖ If the application contains un-checked Exception code is compiled but at runtime JVM (Default Exception handler) display exception message then program terminated abnormally.
- ❖ To overcome runtime problem must handle the exception in two ways.
  - By using try-catch blocks.
  - By using throws keyword.

**Example :-**

```
class Test
{
    public static void main(String[] args)
    {
        int[] a ={10,20,30};
        System.out.println(a[6]);
    }
}
G:\>java Test
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 6
```

**Note-1:-** Whether it is a checked Exception or unchecked exception exceptions are raised at runtime but not compile time.

**Note 2:-** In java whether it is a checked Exception or unchecked Exception must handle the Exception by using try-catch blocks or throws keyword to get normal termination of application & to execute rest of the application.

**Exception vs Error:-**

The exception are occurred due to several reasons

- a. Developer mistakes
- b. End-user input mistakes.
- c. Resource is not available
- d. Networking problems.

✓ Exceptions are caused to several reasons like developer mistakes, end user input mistakes, network problems. But error is caused due to lack of system resources.

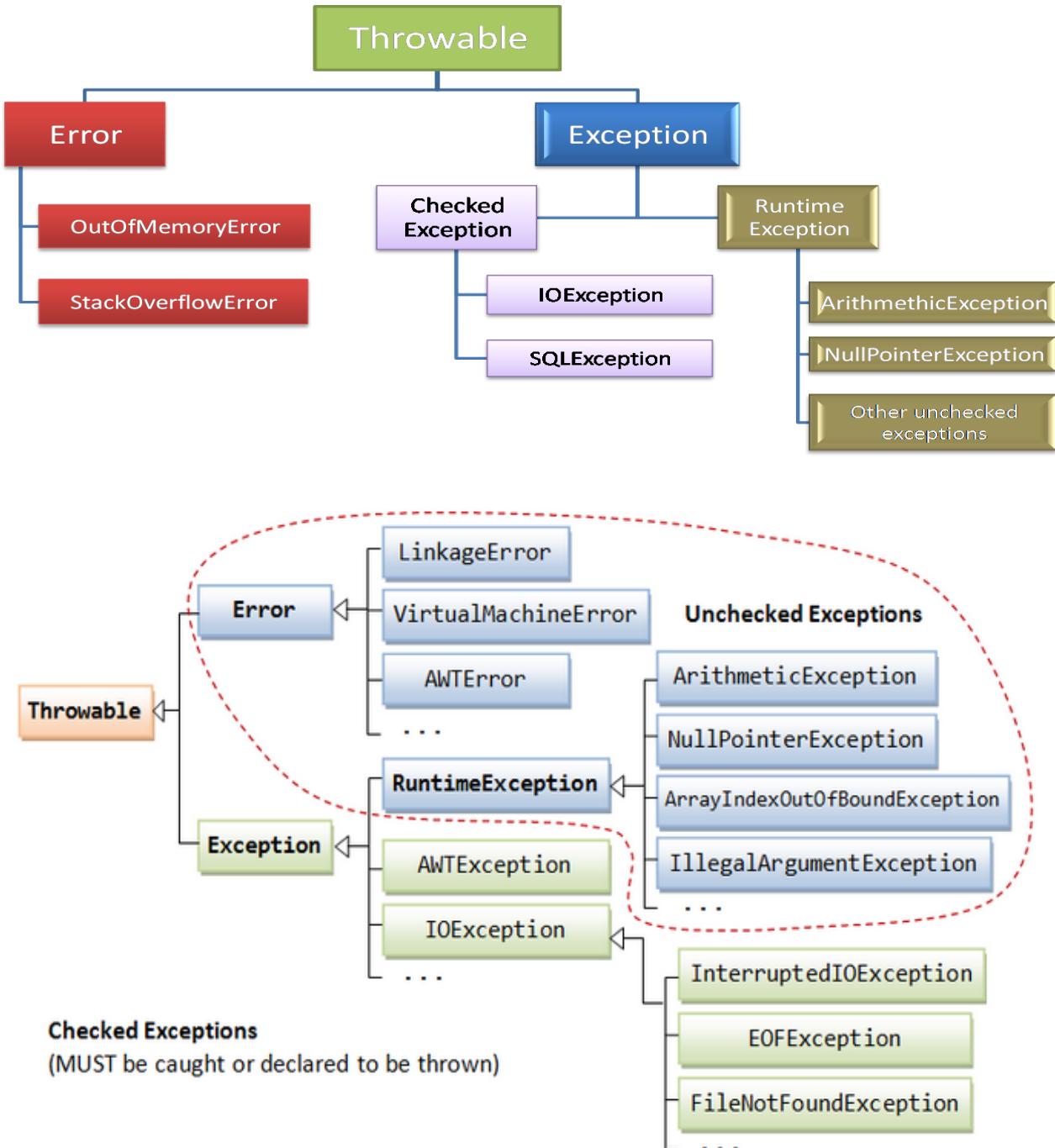
Example: - StackOverflowError, OutOfMemoryError, AssertionEr.....etc

✓ It is possible to handle the exceptions by using try-catch blocks or throws keyword but it is not possible to handle the errors.

✓ Error is an un-checked type exception.

**Example:-**

```
class Test
{
    public static void main(String[] args)
    {
        Test[] t = new Test[100000000];
    }
}
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

Exception Handling Tree Structure:-

- ✓ In above tree Structure `RuntimeException` its child classes & `Error` its child classes are `Unchecked` remaining all exceptions are `checked Exceptions`.
- ✓ The root class of exception handling is **Throwable** class.

**Fully Checked vs partially checked:-**

- ✓ The root class & all its child class are checked then that root class is called **fully checked exception**.  
Example :- IOException,SQLException....etc
- ✓ The root class contains some child classes are checked exceptions & some child classes are un-checked exception then that root class is called **partially checked exception**.  
Example :- Exception , Throwable..etc

**Exception handling key words:-**

- 1) try
- 2) catch
- 3) finally
- 4) throw
- 5) throws

There are two ways to handle the exceptions in java.

- 1) By using try-catch block.
- 2) By using throws keyword.

**Exception handling by using Try –catch blocks:-**

**Syntax:-** try

```
{   exceptional code;
}
catch (ExceptionName reference_variable)
{   Code to run if an exception is raised (alternate code);
}
```

**Example-1 :-****Application without try-catch blocks**

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("ratan");
        System.out.println(10/0);
        System.out.println("rest of the application");
    }
}
```

E:\>java Test

ratan

Exception in Thread “main” java.lang.ArithmaticException: / by zero

Handled by JVM

type of the Exception

description

In above example exception raised program is terminated abnormally & rest of the application is not executed

Whenever the exception raised the default exception handler is responsible to handle the exception & it is component of the JVM.

**Application with try-catch blocks:-**

Whenever the exception is raised in the try block JVM won't terminate the program immediately it will search corresponding catch block.

- If the catch block is matched then that block will be executed & rest of the application executed & program is terminated normally.
- If the catch block is not matched program is terminated abnormally.

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("ratan");
        try
        {
            System.out.println(10/0);
        }
        catch (ArithmaticException ae)
        {
            System.out.println(10/2);
        }
        System.out.println("rest of the application");
    }
}
E:\>java Test
ratan
5
rest of the application
```

**In above example we are handling exception by using try-catch block hence the program is terminated normally & rest of the application is executed.**

**Example-2 :-**

Whenever the exception is raised in the try block JVM won't terminate the program immediately it will search corresponding catch block.

- If the catch block is matched then that block will be executed & rest of the application executed & program is terminated normally.
- If the catch block is not matched program is terminated abnormally.

In below example catch block is not matched hence program is terminated abnormally.

```
class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("sravya");
            System.out.println(10/0);
        }
        catch(NullPointerException e)
        {
            System.out.println(10/2);
        }
        System.out.println("rest of the app");
    }
}
E:\sravya>java Test
sravya
Exception in thread "main" java.lang.ArithmaticException: / by zero
```

**Example 3:-** If there is no exception in try block the corresponding catch blocks are not checked.

```
class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("sravya");
            System.out.println("anu");
        }
        catch(NullPointerException e)
        {
            System.out.println(10/2);
        }
        System.out.println("rest of the app");
    }
}
```

E:\sravya>java Test

```
sravya
anu
rest of the app
```

**Example 4:-** In Exception handling independent try blocks declaration are not allowed must declare

**try-catch or try-finally or try-catch-finally.**

```
class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("sravya");
            System.out.println("anu");
        }
        System.out.println("rest of the app");
    }
}
```

E:\sravya>javac Test.java

Test.java:4: 'try' without 'catch' or 'finally'

**Example 5:-**

✓ In between try-catch blocks it is not possible to declare any statements, if we are declaring statements compiler will generate error message.

✓ In exception handling must declare try with immediate catch block.

```
class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println(10/0);
        }
        System.out.println("anu");
        catch(ArithmeticException e)
        {
            System.out.println(10/2);
        }
        System.out.println("rest of the app");
    }
}
```

**Example 6:-**

- ✓ If the exception raised in try block jvm will search corresponding catch block but if the exception raised other than try-catch blocks it is always abnormal termination.
- ✓ If the exception raised in other than try block it is always abnormal termination.
- ✓ In below example exception raised in catch block hence program is terminated abnormally.

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

**Example 7:-**

- ✓ If the exception raised in try block the remaining code of try block is not executed.
- ✓ Once the control is out of the try block the control never entered into try block once again.
- ✓ Don't take normal code inside try block because no guarantee all statements in try-block will be executed or not.

```
class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println(10/0);
            System.out.println("sravya");
            System.out.println("ratan");
        }
        catch(ArithmetricException e)
        {
            System.out.println(10/2);
        }
        System.out.println("rest of the app");
    }
}
E:\sravya>java Test
5
rest of the app
```

**Example 8:-** The way of handling the exception is varied from exception to the exception hence it is recommended to provide try with multiple number of catch blocks.

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Scanner s=new Scanner(System.in);      //Scanner object used to take dynamic input
        System.out.println("provide the division value");
        int n=s.nextInt();
        try
        {
            System.out.println(10/n);
            System.out.println("ratan".charAt(10));
        }
        catch (ArithmaticException ae)
        {
            System.out.println("Ratan soft");
        }
        catch (StringIndexOutOfBoundsException se)
        {
            System.out.println("durga soft");
        }
        System.out.println("rest of the code");
    }
}
```

**Output:-** provide the division value: 5  
Write the output

**Output:-** provide the division value: 0  
Write the output

**Example 9:-** By using **Exception** class catch block it is possible to hold any type of exceptions.

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Scanner s=new Scanner(System.in);      //Scanner object used to take dynamic input
        System.out.println("provide the division value");
        int n=s.nextInt();
        try
        {
            System.out.println(10/n);
            System.out.println("ratan".charAt(10));
        }
        catch (Exception ae)
        {
            System.out.println("Ratansoft");
        }
        System.out.println("rest of the code");
    }
}
```

**Output:-** provide the division value: 5  
Write the output

**Output:-** provide the division value: 0  
Write the output

**Example 10:-** When we declare multiple catch blocks then the catch block order must be **child-parent** but if we are declaring parent to child compiler will generate error message.

**No compilation error (catch block order child to parent type)**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Scanner s=new Scanner(System.in);
        System.out.println("provide the division val");
        int n=s.nextInt();
        try
        {
            System.out.println(10/n);
            String str=null;
            System.out.println(str.length());
        }
        //catch block order is child to parent
        catch (ArithmaticException ae)
        {
            System.out.println("Exception"+ae);
        }
        catch (Exception ne)
        {
            System.out.println("Exception"+ne);
        }
        System.out.println("rest of the code");
    }
}
```

**Compilation error (catch block order is parent to child)**

```
//catch block order is parent to child
catch (Exception ae)
{
    System.out.println("Exception"+ae);
}
catch (ArithmaticException ne)
{
    System.out.println("Exception"+ne);
}
```

G:\>javac Test.java

Test.java:16: error: exception ArithmaticException has already been caught

**Example 11:-** There are three methods to print Exception information

- `toString()`
- `getMessage()`
- `printStackTrace()`

```
class Test
{
    void m1()
    {
        m2();
    }
    void m2()
    {
        m3();
    }
    void m3()
    {
        try{ System.out.println(10/0); }
        catch(ArithmaticException ae)
        {
            System.out.println(ae.toString());
            System.out.println(ae.getMessage());
            ae.printStackTrace();
        }
    }
    public static void main(String[] args)
    {
        Test1 t = new Test1();
        t.m1();
    }
};
D:\DP>java Test
java.lang.ArithmaticException: / by zero          //toString() method output
/by zero                                         //getMessage() method output
java.lang.ArithmaticException: / by zero          //printStackTrace() method
                                                at Test1.m3(Test1.java:8)
                                                at Test1.m2(Test1.java:5)
                                                at Test1.m1(Test1.java:3)
                                                at Test1.main(Test1.java:17)
```

**Example 12:-** Internally JVM uses `printStackTrace()` method to print exception information.

```
class Test
{
    void m3()      {      System.out.println(10/0);      }
    void m2()      {      m3();      }
    void m1()      {      m2();      }
    public static void main(String[] args)
    {
        new Test().m1();
    }
};
E:\>java Test
Exception in thread "main" java.lang.ArithmaticException: / by zero
                                                at Test.m3(Test.java:3)
                                                at Test.m2(Test.java:6)
                                                at Test.m1(Test.java:9)
                                                at Test.main(Test.java:12)
```

**Example 13:-**

- ✓ It is possible to combine more than one exception in single catch block.
- ✓ This is introduced In 1.7 version the code is compiled & executed in 1.7 & above version.
- ✓

```
catch(ArithmaticException | StringIndexOutOfBoundsException a) .
catch(NumberFormatException | NullPointerException | StringIndexOutOfBoundsException a)
```

```
import java.util.Scanner;
import java.io.*;
public class Test
{ public static void main(String[] args)
{ Scanner s = new Scanner(System.in);
    System.out.println("enter a number");
    int n = s.nextInt();
    try {
        System.out.println(10/n);
        System.out.println("ratan".charAt(13));
    }
    catch(ArithmaticException | ClassCastException a)
    { System.out.println("exception info="+a);
    }
    catch(NumberFormatException | NullPointerException | StringIndexOutOfBoundsException a)
    { System.out.println("exception info="+a);
    }
    System.out.println("Rest of the application");
}
}
```

**Output:- provide the division value: 5**  
**Write the output**

**Output:- provide the division value: 0**  
**Write the output**

**Example14:- multiple exception in single catch for checked Exception.**

```
import java.io.*;
public class Test
{ public static void main(String[] args)
{
    try {
        FileInputStream f = new FileInputStream("abc.txt");
        Thread.sleep(2000);
    }
    catch(FileNotFoundException | InterruptedException a)
    { System.out.println("exception info="+a);
    }
    System.out.println("Rest of the application");
}
}
```

**Example 15:** Above example Observation

If you are declared exceptions in catch block but these exceptions are not raised in try block then it raise an error message.

```
import java.io.*;
public class Test
{ public static void main(String[] args)
{
    try
    {
        FileInputStream f = new FileInputStream("abc.txt");
        Thread.sleep(2000);
    }
    catch(FileNotFoundException | ClassCastException a)
    {
        System.out.println("exception info="+a);
    }
    catch(InterruptedException | ClassNotFoundException a)
    {
        System.out.println("exception info="+a);
    }
}
}
```

G:\>javac Test.java

error: exception ClassNotFoundException is never thrown in body of corresponding try statement  
 catch(InterruptedException | ClassNotFoundException a)

**Example 16: Exception propagation**

If the exception raised in top of the stack method but if you are not handled it drops down to the stack previous method, if you are not catch it drop down until end of the stack(up to main method) this is called exception propagation.

**Note :** only the unchecked Exceptions are propagated but not checked.

```
class Test
{    void m3()
    {
        System.out.println(10/0);
    }
    void m2()
    {
        m3();
    }
    void m1()
    {
        try{
            m2();
        }
        catch(ArithmeticException ae)
        {
            System.out.println("Arithmetic Exception propagation..... ");
        }
    }
    public static void main(String[] args)
    {
        new Test().m1();
    }
}
```

**Propagating unchecked Exception:-(Automatically)**

```

1 package propogation;
2
3
4 /** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
5 public class ExceptionTest {
6    public static void main(String[] args)
7    {
8        method1();
9        System.out.println("after calling m()");
10    }
11
12    static void method1()
13    {
14        method2();
15    }
16
17    static void method2()
18    {
19        method3();
20    }
21
22    static void method3()
23    {
24        throw new NullPointerException();
25    }
26
27
28 }

/*OUTPUT

Exception in thread "main" java.lang.NullPointerException
at propogation.ExceptionTest.method3(ExceptionTest.java:21)
at propogation.ExceptionTest.method2(ExceptionTest.java:17)
at propogation.ExceptionTest.method1(ExceptionTest.java:13)
at propogation.ExceptionTest.main(ExceptionTest.java:8)
*/

```

step 1  
step 2  
step 3  
step 4 (exception propagated)  
step 5 (exception propagated)  
step 6 (exception propagated)

**Propagating checked Exception by using throws keyword:-**

```

1 package propogation;
2 import java.io.FileNotFoundException;
3
4 /** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
5 public class ExceptionTest {
6    public static void main(String[] args)
7    {
8        throws FileNotFoundException {
9            method1();
10           System.out.println("after calling m()");
11       }
12       static void method1() throws FileNotFoundException{
13           method2();
14       }
15
16       static void method2() throws FileNotFoundException{
17           method3();
18       }
19
20       static void method3() throws FileNotFoundException{
21           throw new FileNotFoundException();
22       }
23
24   }
25
26
27 }

/*OUTPUT

Exception in thread "main" java.io.FileNotFoundException
at propogation.ExceptionTest.method3(ExceptionTest.java:21)
at propogation.ExceptionTest.method2(ExceptionTest.java:17)
at propogation.ExceptionTest.method1(ExceptionTest.java:13)
at propogation.ExceptionTest.main(ExceptionTest.java:8)
*/

```

step 1  
step 2  
step 3  
step 4 (propagate exception)  
step 5 (propagate exception)  
step 6 (propagate exception)

**Example 17:-**

Generally finally block is used to close the resources but some time we forgot to close the resources then we will get problems.

You can use a finally block to ensure that a resource is closed regardless of whether the try statement completes normally or abruptly. The following example uses a finally block instead of a try-with-resources statement:

```
BufferedReader br = new BufferedReader(new FileReader(path));
try {    return br.readLine();
}finally {
    if (br != null) br.close();
}
```

To overcome above problem use try-with resources when we declare the resource by using try, when the try-catch block completed resource is automatically closed.

```
try (BufferedReader br = new BufferedReader(new FileReader(path))) {
    return br.readLine();
}
```

BufferedReader will be closed regardless of whether the try statement completes normally or abnormally.

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Scanner s=null;
        try
        {
            s = new Scanner(System.in);
            System.out.println("enter id");
            int a = s.nextInt();
            System.out.println("input value="+a);
        }
        catch (InputMismatchException ae)
        {
            System.out.println("entered input wrong ..... ");
        }
        finally
        {
            s.close();
            System.out.println("scanner is closed");
        }
        //after closing scanner object
        System.out.println("enter second id");
        int b = s.nextInt();
        System.out.println("input value="+b);
    }
}
G:\>java Test
enter id
123
input value=123
scanner is closed
enter second id
Exception in thread "main" java.lang.IllegalStateException: Scanner closed
```

**Example 18:-**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        try(Scanner s = new Scanner(System.in))
        {
            System.out.println("enter id");
            int a = s.nextInt();
            System.out.println("input value="+a);
        }
        catch (InputMismatchException ae)
        {
            System.out.println("entered input wrong ..... ");
        }
    }
}

```

G:\>java Test

```

enter id
123
input value=123

```

**example 19:-** while declaring try with multiple resource every resource separated with semicolon.

```

import java.util.*;
import java.io.*;
class Test
{
    public static void main(String[] args)
    {
        try(Scanner s = new Scanner(System.in);FileInputStream fis = new FileInputStream("abc.txt"))
        {
            System.out.println("enter id");
            int a = s.nextInt();
            System.out.println("input value="+a);
        }
        catch (Exception e)
        {
            System.out.println("entered input wrong ..... ");
        }
    }
}

```

**Example 20:-**

```

import java.io.*;
class Test
{
    void m1(ArithmaticException e)
    {
        System.out.println("m1 method code="+e);
    }
    void m1(Exception ee)
    {
        System.out.println("m2 method code="+ee);
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1(new ArithmaticException());
        t.m1(new IOException());
    }
}

```

**Possibilities of try-catch:-****Case-1**

```
try
{
}
catch ()
```

**Case-3**

```
try
{
}
catch ()
{
}
catch ()
{
}
```

**Case-5**

```
try
{
}
catch ()
{
    try
{
}
catch ()
{
}
}
```

**Case-2**

```
try
{
}
catch ()
```

**Case-4**

```
try
{
    try
{
}
catch ()
{
}
catch ()
{
}
```

**Case-6**

```
try
{
    try
{
}
catch ()
{
    try
{
}
catch ()
{
}
}
```

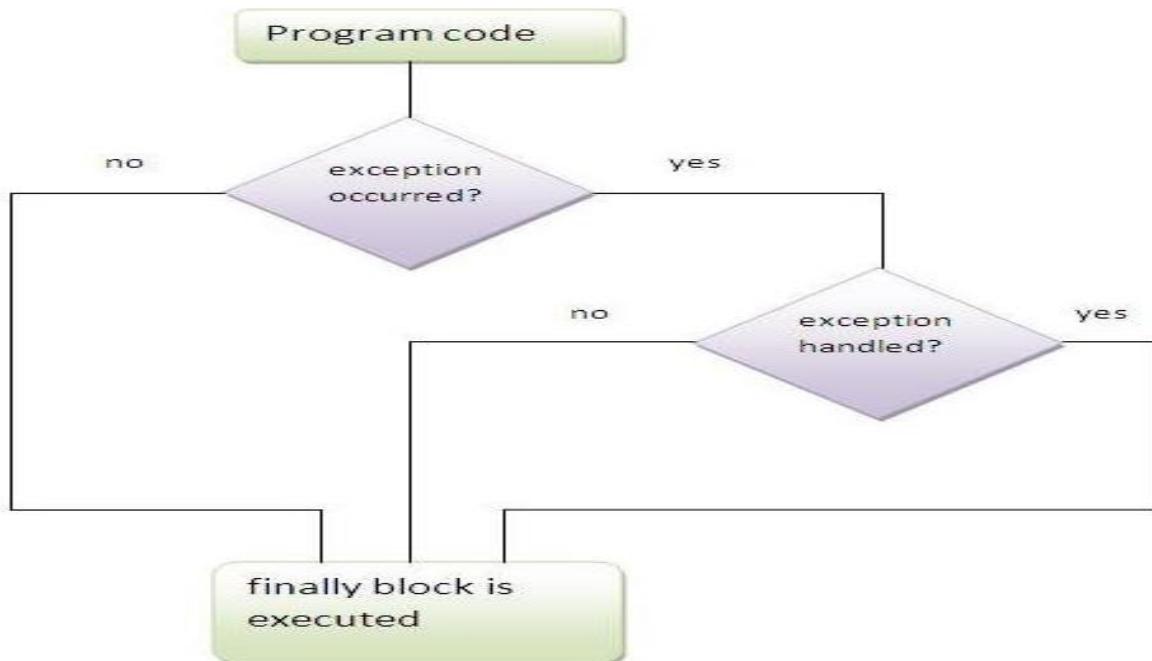
**Finally block:-**

- 1) Finally block code is always executed irrespective of try and catch block code.
- 2) It is used to provide clean-up code
  - a. connection closing. **Connection.close();**
  - b. streams closing. **Scanner.close();**
  - c. Object destruction . **Test t = new Test(); t=null;**
- 3) It is not possible to write finally alone.
 

a. try-catch-finally	---> valid
b. try-catch	---> valid
c. catch-finally	---> invalid
d. try-catch-catch-finally	---> valid
e. try-finally	---> valid
f. catch-catch-finally	---> invalid
g. Try	---> invalid
h. Catch	---> invalid
i. Finally	---> invalid

**Finally block Syntax:-**

```
try
{   risky code;
}
catch (Exception obj)
{   code to be run if the exception raised (handling code);
}
finally
{   Clean-up code;(database connection closing , streams closing.....etc)
}
```



**All possibilities of finally block execution :-****Case 1:-**

```

try
{
    System.out.println("try");
}
catch (ArithmaticException ae)
{
    System.out.println("catch");
}
finally
{
    System.out.println("finally");
}

```

**Output:-**

Try  
finally

**case 2:-****case 2:-**

```

try
{
    System.out.println(10/0);
}
catch (ArithmaticException ae)
{
    System.out.println("catch");
}
finally
{
    System.out.println("finally");
}

```

**Output:-**

catch  
finally

**case 3:-**

```

try
{
    System.out.println(10/0);
}
catch (NullPointerException ae)
{
    System.out.println("catch");
}
finally
{
    System.out.println("finally");
}

```

**Output:-**

finally  
Exception in thread "main"  
java.lang.ArithmaticException: / by zero  
at Test.main(Test.java:4)

**case 5:-**

```

try
{
    System.out.println("try");
}
catch(ArithmaticException ae)
{
    System.out.println("catch");
}
finally
{
    System.out.println(10/0);
}
System.out.println("rest of the code");

```

**D:\>java Test**

try  
Exception in thread "main"  
java.lang.ArithmaticException: / by zero

**case 4:-**

```

try
{
    System.out.println(10/0);
}
catch (ArithmaticException ae)
{
    System.out.println(10/0);
}
finally
{
    System.out.println("finally");
}

```

**D:\morn11>java Test**

finally  
Exception in thread "main"  
java.lang.ArithmaticException: / by zero  
at Test.main(Test.java:7)  
at Test.main(Test.java:15)

**case 6:-it is possible to provide try-finally.**

```

try
{
    System.out.println("try");
}
finally
{
    System.out.println("finally");
}
System.out.println("rest of the code");

```

**D:\>java Test**

try  
finally  
rest of the code

**Example:- final vs return statement**

If the return statement present in try or catch block but finally block executed then only return statement will be executed.

```
class Test
{
    int m1()
    {
        try
        {
            System.out.println("hi try");
            return 10;
        }
        finally
        {
            System.out.println("hi finally");
        }
    }
    public static void main(String[] args)
    {
        int a = new Test().m1();
        System.out.println("return value="+a);
    }
}
G:\>java Test
hi try
hi finally
return value=10
```

**Example :- if the try & catch & finally contains return statement then finally block return statement value is printed.**

```
class Test
{
    int m1()
    {
        try
        {
            return 10;
        }
        catch(Exception e)
        {
            return 20;
        }
        finally
        {
            return 30;
        }
    }
    public static void main(String[] args)
    {
        int a = new Test().m1();
        System.out.println("return value="+a);
    }
}
G:\>java Test
return value=30
```

**Example :-** when the exception raised in try,catch finally but default exception handler is able to handle only one exception at a time that is most recently raised.

```
class Test
{
    void m1()
    {
        try
        {
            System.out.println(10/0);
        }
        catch(Exception e)
        {
            System.out.println(10/0);
        }
        finally
        {
            System.out.println("ratan".charAt(20));
        }
    }
    public static void main(String[] args)
    {
        new Test().m1();
    }
}
```

G:\>java Test

Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 20

**Example:-in two cases finally block won't be executed**

**Case 1:-** whenever the control is entered into try block then only finally block will be executed otherwise it is not executed.

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println(10/0);
        Try { System.out.println("ratan"); }
        finally { System.out.println("finally block"); }
        System.out.println("rest of the code");
    }
};
```

D:\>java Test

Exception in thread "main" java.lang.ArithmaticException: / by zero

**Case 2:-** In your program when we used System.exit(0) the JVM will be shutdown hence the rest of the code won't be executed .

```
class Test
{
    public static void main(String[] args)
    {
        try{ System.out.println("ratan");
              System.exit(0); }
        finally
        {
            System.out.println("finally block"); }
        System.out.println("rest of the code");
    }
};
```

D:\>java Test

Ratan

**Example :-**

```

statement 1
statement 2
try
{
    statement 3
    statement 4
    try
    {
        statement 5
        statement 6
    }
    catch ()
    {
        statement 7
        statement 8
    }
}
catch ()
{
    statement 9
    statement 10
    try
    {
        statement 11
        statement 12
    }
    catch ()
    {
        statement 13
        statement 14
    }
}
Finally{
statement 15
statement 16
}
Statement -17
Statement -18

```

- Case 7:- If the exception is raised in the statement 5 and the corresponding catch block is matched but while executing catch block exception raised in statement-7, the outer catch block is matched while executing outer catch exception raised in statement-11, the inner catch block is matched but while executing inner catch the exception raised in statement-13.
- 1,2,3,4,8,9,12,13,14,15 normal termination.**
- Case 8:- If the exception is raised in the statement 6 and the corresponding catch block is matched but while executing catch block exception raised in statement-8, the outer catch block is matched while executing outer catch exception raised in statement-12, the inner catch block is matched but while executing inner catch the exception raised in statement-14.
- 1,2,3,4,8,9,12,13,14,15 normal termination.**
- Case 9:- If the exception raised in statement 15.
- 1,2,3,4,5 abnormal termination.**
- Case 10:- if the Exception raised in statement 18.

**Case1:** No Exception in the above example.

**1, 2, 3, 4, 5, 14, 15 Normal Termination**

**Case 2:-** if the exception is raised in statement 2.

**1 , Abnormal Termination**

**Case 3:-** if the exception is raised in the statement 3 the corresponding catch block Is not matched.

**1,2,15,16 Abnormal termination**

**Case 4:-** if the exception is raise in the statement-4 the corresponding catch block is matched.

**1,2,15,16 Abnormal termination**

**Case 5:-** If the exception is raised in the statement 5 and corresponding catch block is matched.

**1,2,3,4,8,9,10,11,14,15 normal termination**

**Case 6:-** If the exception is raised in the statement 6 and corresponding catch block is not matched but outer catch block is matched.

**1,2,3,4,8,9,10,11,14,15 normal termination**

**Throws keyword:-**

There are two approaches to handle the exceptions in java

- By using try-catch blocks.
- By using throws keyword.

**Handling exception by using Try-catch**

- Try-catch blocks are used to write the exception handling code.
- By using try-catch blocks it is possible to handle multiple exceptions by using multiple catch blocks.
- We can write the try-catch blocks at method implementation level.
- We can provide the try-catch blocks at method & constructor & blocks level.

**Handling Exception by using throws keyword**

Throws keyword is used to delegate the responsibilities of exception handling to caller method.

By using throws it is possible to handle multiple exceptions because one method is able to throw multiple exceptions at time.  
We can write the throws keyword at method declaration level.  
We can provide the throws keyword only at method & constructor level but not block level.

**Example 1:-**

- ✓ in below example exception raised in studentDetails() method but it delegating responsibilities of exception handling to hod() method by using throws keyword.
- ✓ But hod() method delegating responsibilities of exception handling to principal() method by using throws now principal handing this exception by using try-catch blocks.

```
class Test
{
    void studentDetails() throws InterruptedException
    {
        System.out.println("suneel babu is sleeping");
        Thread.sleep(3000);
        System.out.println("do not disturb sir.....");
    }
    void hod()throws InterruptedException
    {
        studentDetails();
    }
    void principal()
    {
        try{
            hod();
        catch(InterruptedException ie)
        {
            ie.printStackTrace();
        }
    }
    void officeBoy()
    {
        principal();
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.officeBoy();
    }
}
```

**Example 2:-**

- ✓ In below example method-by-method using throws keyword to delegate responsibilities of exception handling to caller method.
- ✓ At final main() method uses throws keyword to delegate the responsibilities of exception handling to JVM.

```
class Test
{
    void studentDetails() throws InterruptedException
    {
        System.out.println("suneel babu is sleeping");
        Thread.sleep(3000);
        System.out.println("do not disturb sir..... ");
    }
    void hod()throws InterruptedException
    {
        studentDetails();
    }
    void principal()throws InterruptedException
    {
        hod();
    }
    void officeBoy()throws InterruptedException
    {
        principal();
    }
    public static void main(String[] args) throws InterruptedException
    {
        Test t = new Test();
        t.officeBoy();
    }
}
```

**Example 3:-**

```
import java.io.*;
class Test
{
    void m2()throws FileNotFoundException,InterruptedException
    {
        FileInputStream fis = new FileInputStream("abc.txt");
        Thread.sleep(2000);
        System.out.println("Exceptions are handled");
    }
    void m1()
    {
        try{
            m2();
        catch(FileNotFoundException f) { f.printStackTrace(); }
        catch(InterruptedException ie) { ie.printStackTrace(); }
        }
    public static void main(String[] args) throws InterruptedException
    {
        Test t = new Test();
        t.m1();
    }
}
```

**Example 4:-**

```

import java.io.*;
class Test
{
    void m2()throws Exception           //FileNotFoundException,InterruptedException
    {
        FileInputStream fis = new FileInputStream("abc.txt");
        Thread.sleep(2000);
        System.out.println("Exceptions are handled");
    }
    void m1()throws Exception
    {
        m2();
    }
    public static void main(String[] args) throws Exception
    {
        Test t = new Test();
        t.m1();
    }
}

```

**Exception Handling with Method overriding in java:-****Example 1:-**

If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.

**Case 1:-**

```

import java.io.*;
class Parent
{
    void m1()
    {
        System.out.println("Parent m1()");
    }
}
class Child extends Parent
{
    void m1()throws IOException
    {
        System.out.println("child m1()");
    }
}

```

G:\>javac Test.java

Test.java:8: error: m1() in Child cannot override m1() in Parent  
overridden method does not throw IOException

**case 2:-**

```

class Parent
{
    void m1()
    {
        System.out.println("Parent m1()");
    }
}
class Child extends Parent
{
    void m1()throws ArithmeticException
    {
        System.out.println("child m1()");
    }
}

```

**Example :2** If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

**Case 1: Overriding & Overridden method same type of exception.**

```
class Parent
{
    void m1()throws ArithmeticException
    {
        System.out.println("Parent m1()");
    }
}

class Child extends Parent
{
    void m1()throws ArithmeticException
    {
        System.out.println("child m1()");
    }
}
```

**Case 2: Overriden method contains Exception but overriding no exception**

```
class Parent
{
    void m1()throws ArithmeticException
    {
        System.out.println("Parent m1()");
    }
}

class Child extends Parent
{
    void m1()
    {
        System.out.println("child m1()");
    }
}
```

**Case 3: overriding method is super class & overridden method is sub class**

```
class Parent
{
    void m1()throws Exception
    {
        System.out.println("Parent m1()");
    }
}

class Child extends Parent
{
    void m1()throws ArithmeticException
    {
        System.out.println("child m1()");
    }
}
```

**Case 4: overridden method Is sub-type & overriding method is parent-type not allowed generate Compilation error.**

```
class Parent
{
    void m1()throws ArithmeticException
    {
        System.out.println("Parent m1()");
    }
}

class Child extends Parent
{
    void m1()throws Exception
    {
        System.out.println("child m1()");
    }
}
```

error: m1() in Child cannot override m1() in Parent overridden method does not throw Exception.

**Throw keyword:-**

- ✓ It is used to handover user created Exception object to JVM.
- ✓ It is used to throw exception explicitly.
- ✓ By using throw keyword it is possible throw predefined Exceptions & custom exception but it is always recommended to throw custom exceptions.

*Note: - throw keyword is used to handover user created exception object to JVM whether it is predefined exception class or user defined exception class but it is always recommended throw custom exception.*

***Example:- throw statement throw an predefined exception.***

Step 1:- create the Exception object explicitly by the developer by using new keyword.

```
new ArithmeticException("ratan not eligible");
```

Step 2:- handover (throw) user created Exception object to jvm by using throw keyword.

```
throw new ArithmeticException("ratan not eligible");
```

***Example:-***

```
import java.util.*;
class Test
{
    static void validate(int age)
    {
        if(age<18)
        {
            throw new ArithmeticException("not eligible for vote");
        }
        else
        {
            System.out.println("welcome to the voting");
        }
    }
    public static void main(String[] args)
    {
        Scanner s=new Scanner(System.in);
        System.out.println("please enter your age ");
        validate(s.nextInt());
        System.out.println("rest of the code");
    }
}
```

E:\>java Test  
please enter your age  
45  
Check the output

E:\>java Test  
please enter your age  
10  
Check the output

***Example: - throw statement throw a user defined exception.***

To achieve this mechanism first we must know how to create user defined exception then we are able to use this throw keyword.

There are two types of exceptions present in the java language

- 1) Predefined Exceptions.

ArithmeticException, IOException, NullPointerException.....etc

- 2) User defined Exceptions.(created by user)

InvalidAgeException, MyException...etc

**Customization of exception handling :-( creation of predefined exceptions)**

There are two types of user defined exceptions

1. User defined checked exception.
  - a. Default constructor approach.
  - b. Parameterized constructor approach.
2. User defined un-checked Exception.
  - a. Default constructor approach.
  - b. Parameterized constructor approach.

Note: - while declaring user defined exceptions: the naming conventions are every exception suffix must be the word *Exception*.

**Creation of user defined checked Exception by using default constructor approach:-****Step-1:- create the user defined checked Exception**

Normal java class will become Exception class whenever we are extends *Exception class*.

**InvalidAgeException.java:-**

```
package com.tcs.userexceptions;
public class InvalidAgeExcepiton extends Exception
{
    //default constructor
};
```

**Step-2:- use the user created Exception in our project.****Test.java**

```
package com.tcs.project;
import com.tcs.userexceptions.InvalidAgeExcepiton;
import java.util.Scanner;
class Test
{
    static void status(int age)throws InvalidAgeExcepiton
    {
        if (age>25)
            {System.out.println("eligible for mrg");
        }
        else
            {      throw new InvalidAgeExcepiton(); //default constructor executed
        }
    }
    public static void main(String[] args)throws InvalidAgeExcepiton
    {
        Scanner s = new Scanner(System.in);
        System.out.println("enter u r age");//23
        int age = s.nextInt();
        Test.status(age);
    }
}
```

D:\morn11>java com.tcs.project.Test  
Enter u r age & check the output

Example :-Creation of user defined checked exception by using parameterized constructor approach.

**step-1:- create the user defined checked exception class.**

Normal java class will become checked exception class when we extends Exception class.

**InvalidAgeException.java**

```
package com.tcs.userexceptions;
public class InvalidAgeExcepiton extends Exception
{
    public InvalidAgeExcepiton(String str)
    {
        super(str); //super constructor calling in order to print your information
    }
}
```

**Step-2:- use user created Exception in our project.**

**Test.java**

```
package com.tcs.project;
import com.tcs.userexceptions.InvalidAgeExcepiton;
import java.util.Scanner;
class Test
{
    static void status(int age)throws InvalidAgeExcepiton
    {
        if (age>25)
        {
            System.out.println("eligible for mrg");
        }
        else
        {
            //using user created Exception
            throw new InvalidAgeExcepiton("not eligible try after some time");
        }
    }
    public static void main(String[] args)throws InvalidAgeExcepiton
    {
        Scanner s = new Scanner(System.in);
        System.out.println("enter u r age");
        int age = s.nextInt();
        Test.status(age);
    }
}
```

D:\morn11>javac -d . InvalidAgeExcepiton.java

D:\morn11>javac -d . Test.java

D:\morn11>java com.tcs.project.Test

enter u r age

28

eligible for mrg

D:\morn11>java com.tcs.project.Test

enter u r age

20

*Exception in thread "main" com.tcs.userexceptions.InvalidAgeExcepiton: not eligible try after some time*

*at com.tcs.project.Test.status(Test.java:11)*

*at com.tcs.project.Test.main(Test.java:18)*

**Example:-creation of user defined un-checked exception by using default constructor approach****Step-1:- create user defined un-checked exception.**

Normal java class will become un-checked exception class when we extends RuntimeException class.

**InvalidAgeException.java**

```
package com.tcs.userexceptions;
public class InvalidAgeExcepiton extends RuntimeException
{
    //default constructor
}
```

**Step-2:- use user created Exception in your project.****Test.java**

```
package com.tcs.project;
import com.tcs.userexceptions.InvalidAgeExcepiton;
import java.util.Scanner;
class Test
{
    static void status(int age)
    {
        if (age>25)
            {System.out.println("eligible for mrg");
        }
        else
        {
            //using user created Exception
            throw new InvalidAgeExcepiton();
        }
    }
    public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in);
        System.out.println("enter u r age");//23
        int age = s.nextInt();
        Test.status(age);
    }
}
```

**Example: - creation of user defined un-checked exception by using parameterized constructor approach****Step 1:- create user defined un-checked exception classs.**

Normal java class will become un-checked exception class when we extends RuntimeException class.

**InvalidAgeException.java**

```
package com.tcs.userexceptions;
public class InvalidAgeExcepiton extends RuntimeException
{
    public InvalidAgeExcepiton(String str)
    {
        super(str);
    }
};
```

**Step2:- use user created exception object in your project.**

**Test.java**

```
package com.tcs.project;
import com.tcs.userexceptions.InvalidAgeException;
import java.util.Scanner;
class Test
{
    static void status(int age)
    {
        if (age>25)
        {
            System.out.println("eligible for mrg");
        }
        else
        {
            //using user created Exception
            throw new InvalidAgeException("not eligible for mrg");
        }
    }
    public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in);
        System.out.println("enter u r age");
        int age = s.nextInt();
        Test.status(age);
    }
}
```

**Differences between checked Exception & unchecked Exception:-**

**User checked Exception**

1. Our normal java class will become checked Exception class when extends Exception class.

```
class InvalidAgeException extends Exception
{
    //logics here
}
```

2. Must handle the checked Exceptions by using try-catch block or throws keyword.

**User un-checked Exception**

1. Our normal java class will become checked Exception class when extends Exception class.

```
class InvalidAgeException extends RuntimeException
{
    //logics here
}
```

2. Handling unchecked Exceptions is optional but it is recommended.

**Different types of exceptions:-**

**ArrayIndexOutOfBoundsException:-**

```
int[] a={10,20,30};
System.out.println(a[4]);//ArrayIndexOutOfBoundsException
```

**NumberFormatException:-**

```
String str1="abc";
int b=Integer.parseInt(str1);
System.out.println(b); //NumberFormatException
```

**NullPointerException:-**

```
String str1=null;
System.out.println(str1.length());//NullPointerException
```

**ArithmaticException:-**

```
int b=10/0;
System.out.println(b);//ArithmaticException
```

**IllegalArgumentException:-**

```
Thread priority range is 1-10
1 --->low priority    10 --->high priority
Thread t=new Thread();
t.setPriority(11); //IllegalArgumentException
```

**IllegalThreadStateException:-**

```
Thread t=new Thread();
t.start();
t.start(); //IllegalThreadStateException
```

**StringIndexOutOfBoundsException:-**

```
String str="rattaiah";
System.out.println(str.charAt(13)); //StringIndexOutOfBoundsException
```

**NegativeArraySizeException:-**

```
int[] a=new int[-9];
System.out.println(a.length()); //NegativeArraySizeException
```

**InputMismatchException:-**

```
Scanner s=new Scanner(System.in);
System.out.println("enter first number");
int a=s.nextInt();
```

D:\>**java Test**

**enter first number**

**ratan**

**Exception in thread "main" java.util.InputMismatchException**

**ClassCastException:-**

```
class Test
{
    public static void main(String[] args)
    {
        String s = new String("ratan");
        Object o = (Object)s;

        Object oo = new Object();
        String str = (String)oo;           // java.lang.ClassCastException
    }
}
```

**java.lang.NoClassDefFoundError vs java.lang.ClassNotFoundException:-**

```

class Test1
{
    void m1()
    {
        System.out.println("Test1 class m1()");
    }
}
class Test
{
    public static void main(String[] args) throws ClassNotFoundException
    {
        Test1 t = new Test1();
        t.m1();
        Class.forName("Emp");
    }
}

```

**Observation-1:-** In Test class we are hard coding Test1 object but in target location Test1.class file is not available it will generate **java.lang.NoClassDefFoundError**.

**Observation-2:-**

In java to load .class file dynamically at runtime we are using forName() method but if runtime the class is not available it generate **java.lang.ClassNotFoundException**.

**Different types of Errors:-****StackOverflowError:-**

```

class Test
{
    void m1()
    {
        m2();
        System.out.println("this is Rattaiah");
    }
    void m2()
    {
        m1();
        System.out.println("from Savyasoft");
    }
    public static void main(String[] args)
    {
        new Test().m1();
    }
}

```

**OutOfMemoryError:-**

```

class Test
{
    public static void main(String[] args)
    {
        int[] a=new int[100000000]; //OutOfMemoryError
    }
}

```

**ExceptionInInitializerError:-**

```

class Test
{
    static int a=10/0;
    public static void main(String[] args) { }
}

```

Exception in thread "main" java.lang.ExceptionInInitializerError  
Caused by: java.lang.ArithmaticException: / by zero

**Different types of Exceptions in java:-**

<b>Checked Exception</b>	<b>Description</b>
ClassNotFoundException	If the loaded class is not available
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	If the requested method is not available.
<b>UncheckedException</b>	
ArithmaticException	Arithmatic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.(out of range)
InputMismatchException	If we are giving input is not matched for storing input.
ClassCastException	If the conversion is Invalid.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.

## Multi Threading

### **Information about multithreading:-**

- 1) The earlier days the computer's memory is occupied only one program after completion of one program it is possible to execute another program is called uni programming.
- 2) Whenever one program execution is completed then only second program execution will be started such type of execution is called co operative 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 co operative execution

**To overcome above problem a new programming style will be introduced is called multiprogramming.**

- 1) Multiprogramming means executing the more than one program at a time.
- 2) All these programs are controlled by the CPU scheduler.
- 3) CPU scheduler will allocate a particular time period for each and every program.
- 4) Executing several programs simultaneously is called multiprogramming.
- 5) In multiprogramming a program can be entered in different states.
  - a. Ready state.
  - b. Running state.
  - C. Waiting state.
- 6) Multiprogramming mainly focuses on the number of programs.

### **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.
- 5) Executing more than one thread a time is called multithreading.

**Information about main Thread:-**

When a java program started one Thread is running immediately that thread is called main thread of your program.

1. It is used to create a new Thread(child Thread).
2. It must be the last thread to finish the execution because it perform various actions.

It is possible to get the current thread reference by using `currentThread()` method it is a static public method present in Thread class.

```
class CurrentThreadDemo
{
    public static void main(String[] arhgs)
    {
        Thread t=Thread.currentThread();
        System.out.println("current Thread-->" +t);
        //change the name of the thread
        t.setName("ratan");
        System.out.println("after name changed--> " +t);
    }
};
```

**Single threaded model:-**

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
        System.out.println("hi rattaiah");
        System.out.println("hello Sravyasoft");
    }
}
```

In the above program only one thread is available is called main thread to know the name of the thread we have to execute the fallowing code.

**The main important application areas of the multithreading are**

1. Developing video games
2. Implementing multimedia graphics.
3. Developing animations

**A thread can be created in two ways:-**

- 1) By extending Thread class.
- 2) By implementing `java.lang.Runnable` interface

**First approach to create thread extending Thread class:-**

**Step 1:-** Our normal java class will become Thread class whenever we are extending predefined Thread class.

```
class MyThread extends Thread
{
}
```

**Step 2:- override the run() method to write the business logic of the Thread( run() method present in Thread class).**

```
class MyThread extends Thread
{
    public void run()
    {
        System.out.println("business logic of the thread");
        System.out.println("body of the thread");
    }
}
```

**Step 2:- Create userdefined Thread class object.**

```
MyThread t=new MyThread();
```

**Step 3:- Start the Thread by using start() method of Thread class.**

```
t.start();
```

**Example :-**

```
class MyThread extends Thread //defining a Thread
{
    //business logic of user defined Thread
    public void run()
    {
        for (int i=0;i<10;i++)
        {
            System.out.println("userdefined Thread");
        }
    }
}
class ThreadDemo
{
    public static void main(String[] args) //main thread started
    {
        MyThread t=new MyThread(); //MyThread is created
        t.start(); //MyThread execution started
        //business logic of main Thread
        for (int i=0;i<10;i++)
        {
            System.out.println("Main Thread");
        }
    }
}
```

**Flow of execution:-**

- 1) Whenever we are calling t.start() method then JVM will search start() method in the MyThread class since not available so JVM will execute parent class(**Thread**) start() method.

***Thread class start() method responsibilities***

- a. User defined thread is registered into Thread Scheduler then only decide new Thread is created.
- b. The Thread class start() automatically calls run() to execute logics of userdefined Thread.

**Thread Scheduler:-**

- ✓ Thread scheduler is a part of the JVM. It decides thread execution.
- ✓ Thread scheduler is a mental patient we are unable to predict exact behavior of Thread Scheduler it is JVM vendor dependent.
- ✓ Thread Scheduler mainly uses two algorithms to decide Thread execution.
  - 1) Preemptive algorithm.
  - 2) Time slicing algorithm.
- ✓ We can't expect exact behavior of the thread scheduler it is JVM vendor dependent. So we can't say expect output of the multithreaded examples we can say the possible outputs.

**Preemptive scheduling:-**

In this highest priority task is executed first after this task enters into waiting state or dead state then only another higher priority task come to existence.

**Time Slicing Scheduling:-**

A task is executed predefined slice of time and then return pool of ready tasks. The scheduler determines which task is executed based on the priority and other factors.

**Example :-is it possible to start a thread twice : no**

```
class MyThread extends Thread
{
    public static void main(String[] args)//main thread started
    {
        MyThread t=new MyThread(); //MyThread is created
        t.start();
        t.start();
    }
}
D:\DP>java MyThread
Exception in thread "main" java.lang.IllegalThreadStateException
```

**Life cycle stages are:-**

- 1) New
- 2) Ready
- 3) Running state
- 4) Blocked / waiting / non-running mode
- 5) Dead state

**New :-**      MyThread t=new MyThread();

**Ready :-**      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:-** If the business logic of the project is completed means run() over thread goes dead state.

**Second approach to create thread implementing Runnable interface:-**

**Step 1:-** our normal java class will become Thread class whenever we are implementing Runnable interface.

*class MyClass extends Runnable*

```
{
};
```

**Step2: override run method to write logic of Thread.**

```
class MyClass extends Runnable
{
    public void run()
    {
        System.out.println("Rattaiah from SravyaInfotech");
        System.out.println("body of the thread");
    }
}
```

**Step 3:- Creating a object.**

```
MyClass obj=new MyClass();
```

**Step 4:- Creates a Thread class object.**

After new Thread is created it is not started running until we are calling start() method.  
So whenever we are calling start method that start() method call run() method then the new Thread execution started.

```
Thread t=new Thread(obj);
t.start();
```

**creation of Thread implementing Runnable interface :-**

```
class MyThread implements Runnable
{
    public void run()
    {
        //business logic of user defined Thread
        for (int i=0;i<10;i++)
        {
            System.out.println("userdefined Thread");
        }
    }
};

class ThreadDemo
{
    public static void main(String[] args)      //main thread started
    {
        MyThread r=new MyThread(); //MyThread is created
        Thread t=new Thread(r);
        t.start();           //MyThread execution started
        //business logic of main Thread
        for (int i=0;i<10;i++)
        {
            System.out.println("Main Thread");
        }
    }
};
```

**First approach:-**

important point is that when extending the Thread class, the sub class cannot extend any other base classes because Java allows only single inheritance.

**Second approach:-**

- 1) Implementing the Runnable interface does not give developers any control over the thread itself, as it simply defines the unit of work that will be executed in a thread.
- 2) By implementing the Runnable interface, the class can still extend other base classes if necessary.

**Creating two threads by extending Thread class using anonymous inner classes:-**

```
class ThreadDemo
{
    public static void main(String[] args)
    {
        Thread t1 = new Thread()          //anonymous inner class
        {
            public void run()
            {System.out.println("user Thread-1");}
        };
        Thread t2 = new Thread()          //anonymous inner class
        {
            public void run()
            {System.out.println("user thread-2");}
        };
        t1.start();
        t2.start();
    }
};
```

**Creating two threads by implementing Runnable interface using anonymous inner classes:-**

```
class ThreadDemo
{
    public static void main(String[] args)
    {
        Runnable r1 = new Runnable()
        {
            public void run()
            {System.out.println("user Thread-1");}
        };
        Runnable r2 = new Runnable()
        {
            public void run()
            {System.out.println("user thread-2");}
        };
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        t1.start();
        t2.start();
    }
};
```

**Different ways to start the Thread:-**

```
class MyThread extends Thread
{
    public void run()
    {System.out.println("user thread is running extends Thread");}
};

class MyRunnable implements Runnable
```

```

{     public void run()
    {System.out.println("user thread is Running implements Runnable");
    }
};

class ThreadDemo
{     public static void main(String[] args)
    {      //creating Thread class object by passing anonymous classes
        new Thread(new MyThread()).start();
        new Thread(new MyRunnable()).start();
    }
};

```

#### Internal Implementation of multiThreading:-

```

interface Runnable
{     public abstract void run();
}

class Thread implements Runnable
{     public void run()
    {      //empty implementation
    }
};

class MyThread extends Thread
{     public void run()      //overriding run() to write business logic
    {      for (int i=0;i<5 ;i++ )
        {      System.out.println("user implementation");
        }
    }
};

```

#### Difference between t.start() and t.run():-

- In the case of t.start(), Thread class start() is executed a new thread will be created that is responsible for the execution of run() method.
- But in the case of t.run() method, no new thread will be created and the run() is executed like a normal method call by the main thread.

**Note :- Here we are not overriding the run() method so thread class run method is executed which is having empty implementation so we are not getting any output.**

```

class MyThread extends Thread
{     }
class ThreadDemo
{     public static void main(String[] args)
    {      MyThread t=new MyThread();
        t.start();
        for (int i=0;i<5;i++ )
        {      System.out.println("main thread");
        }
    }
};

```

```

        }
    }

Note :- If we are overriding start() method then JVM is executes override start() method at this situation we are not giving chance to the thread class start() hence n new thread will be created only one thread is available the name of that thread is main thread.

class MyThread extends Thread
{
    Public void start()
    {
        System.out.println("override start method");
    }
}

class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
        for (int i=0;i<5 ;i++ )
        {
            System.out.println("main thread");
        }
    }
}

```

**Different Threads are performing different tasks:-**

- 1) Particular task is performed by the number of threads here number of threads(t1,t2,t3) are executing same method (functionality).
- 2) In the above scenario for each and every thread one stack is created. Each and every method called by particular Thread the every entry stored in the particular thread stack.

```

class MyThread1 extends Thread
{
    public void run()
    {
        System.out.println("ratan task");
    }
};

class MyThread2 extends Thread
{
    public void run()
    {
        System.out.println("Sravya task");
    }
};

class MyThread3 extends Thread
{
    public void run()
    {
        System.out.println("anu task");
    }
};

class ThreadDemo
{
    public static void main(String[] args) //1- main Thread
    {
        MyThread1 t1 = new MyThread1();
        MyThread2 t2 = new MyThread2();
        MyThread3 t3 = new MyThread3();
        t1.start(); //2
        t2.start(); //3
        t3.start(); //4
    }
}

```

```

        }
    };

```

**Here Four Stacks are created**

**Main -----stack1**

**t1-----stack2**

**t2-----stack3**

**t3-----stack4**

**Multiple threads are performing single task:-**

```

class MyThread extends Thread
{
    public void run()
    {
        System.out.println("Sravyasoft task");
    }
}

class ThreadDemo
{
    public static void main(String[] args)//main Thread is started
    {
        MyThread t1=new MyThread();           //new Thread created
        MyThread t2=new MyThread();           //new Thread created
        MyThread t3=new MyThread();           //new Thread created
        t1.start();                         //Thread started
        t2.start();                         //Thread started
        t3.start();                         //Thread started
    }
}

```

**Getting and setting names of Thread:-**

1) Every Thread in java having name

- a. default name of the main thread is main
- b. default name of user created threads starts from **Thread-0**.

t1 → Thread-0

t2 → Thread-1

t3 → Thread-2

2) To set the name use **setName()** & to get the name use **getName()**,

**Public final String getName()**

**Public final void setName(String name)**

**Example:-**

```

class MyThread extends Thread
{
    public void run()
    {
        System.out.println("thread is running");
    }
}

class ThreadDemo
{
    public static void main(String args[])
    {
        MyThread t1=new MyThread();
        MyThread t2=new MyThread();
        System.out.println("t1 Thread name="+t1.getName());
        System.out.println("t2 Thread name="+t2.getName());
        System.out.println(Thread.currentThread().getName());
        t1.setName("ratan");
    }
}

```

```

        System.out.println("after changeing t1 Thread name="+t1.getName());
    }
}

```

### Thread Priorities:-

1. Every Thread in java has some property. It may be default priority provided by the JVM or customized priority provided by the programmer.
2. The valid range of thread priorities is 1 – 10. Where one is lowest priority and 10 is highest priority.
3. The default priority of main thread is 5. The priority of child thread is inherited from the parent.
4. Thread defines the following constants to represent some standard priorities.
5. Thread Scheduler will use priorities while allocating processor the thread which is having highest priority will get chance first and the thread which is having low priority.
6. If two threads having the same priority then we can't expect exact execution order it depends upon Thread Scheduler.
7. The thread which is having low priority has to wait until completion of high priority threads.
8. Three constant values for the thread priority.
  - a. **MIN\_PRIORITY = 1**
  - b. **NORM\_PRIORITY = 5**
  - c. **MAX\_PRIORITY = 10**

Thread class defines the following methods to get and set priority of a Thread.

**Public final int getPriority()**  
**Public final void setPriority(int priority)**

Here 'priority' indicates a number which is in the allowed range of 1 – 10. Otherwise we will get Runtime exception saying "IllegalArgumentException".

**Thread priority decide when to switch from one running thread to another this process is called context switching.**

```

class MyThread extends Thread
{
    public void run()
    {
        System.out.println("current Thread name = "+Thread.currentThread().getName());
        System.out.println("current Thread priority = "+Thread.currentThread().getPriority());
    }
}
class ThreadDemo
{
    public static void main(String[] args)//main thread started
    {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        t1.setPriority(Thread.MIN_PRIORITY);
        t2.setPriority(Thread.MAX_PRIORITY);
        t1.start();
        t2.start();
    }
}

```

```
};
```

#### Java.lang.Thread.yield():-

- ❖ Yield() method causes to pause current executing Thread for giving the chance for waiting threads of same priority.
- ❖ If there are no waiting threads or all threads are having low priority then the same thread will continue its execution once again.

#### **Syntax:-**

```
Public static native void yield();
```

**Ex:**

```
class MyThread extends Thread
{
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            Thread.yield();
            System.out.println("child thread");
        }
    }
}

class ThreadYieldDemo
{
    public static void main(String[] args)
    {
        MyThread t1=new MyThread();
        t1.start();
        for(int i=0;i<10;i++)
        {
            System.out.println("main thread");
        }
    }
}
```

#### Java.lang.Thread.join(-,-) method:-

- Join method allows one thread to wait for the completion of another thread.
  - t.join(); ---> here t is a Thread Object whose thread is currently running.
- Join() is used to stop the execution of the thread until completion of some other Thread.
- 

***if a t1 thread is executed t2.join() at that situation t1 must wait until completion of the t2 thread.***

```
public final void join()throws InterruptedException
Public final void join(long ms)throws InterruptedException
Public final void join(long ms, int ns)throws InterruptedException
```

#### Methods of Thread class:-

```
class MyThread extends Thread
{
    public void run()
    {
        for (int i=0;i<5;i++)
```

```

        {
            try{    Thread.sleep(2000);  }
            catch(InterruptedException e)
            {e.printStackTrace();
            }
            System.out.println(i);
        }
    }
};

class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t1=new MyThread();
        MyThread t2=new MyThread();
        MyThread t3=new MyThread();
        t1.start();
        try
        {t1.join();      }
        catch (InterruptedException ie)
        {ie.printStackTrace();
        }
        t2.start();
        t3.start();
    }
}

```

#### **Java.lang.Thread.Interrupted():-**

- ❖ A thread can interrupt another sleeping or waiting thread. But one thread is able to interrupted only another sleeping or waiting thread.
- ❖ To interrupt a thread use Thread class interrupt() method.

#### **Public void interrupt()**

#### **Effect of interrupt() method call:-**

```

class MyThread extends Thread
{
    public void run()
    {
        try
        {
            for (int i=0;i<10;i++)
            {
                System.out.println("i am sleeping ");
                Thread.sleep(5000);
            }
        }
        catch (InterruptedException ie)
        {
            System.out.println("i got interupted by interrupt() call");
        }
    }
};

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

```

```

    {
        MyThread t=new MyThread();
        t.start();
        t.interrupt();
    }
};

No effect of interrupt() call:-
class MyThread extends Thread
{
    public void run()
    {
        for (int i=0;i<10;i++)
        {
            System.out.println("i am sleeping ");
        }
    }
};

class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
        t.interrupt();
    }
};

```

**NOTE:-The interrupt() is effected whenever our thread enters into waiting state or sleeping state and if the our thread doesn't enters into the waiting/sleeping state interrupted call will be wasted.**

### Shutdown Hook:-

- Shutdown hook used to perform cleanup activities when JVM shutdown normally or abnormally.
- Clean-up activities like
  - Resource release
  - Database closing
  - Sending alert message
- So if you want to execute some code before JVM shutdown use shutdown hook

### **The JVM will be shutdown in following cases.**

- a. When you typed ctrl+C
- b. When we used System.exit(int)
- c. When the system is shutdown .....etc

To add the shutdown hook to JVM use addShutdownHook(obj) method of Runtime Class.

**public void addShutdownHook(java.lang.Thread);**

To remove the shutdown hook from JVM use removeShutdownHook(obj) method of Runtime Class.

**public boolean removeShutdownHook(java.lang.Thread);**

To get the Runtime class object use static factory method getRuntime() & this method present in Runtime class

**Runtime r = Runtime.getRuntime();**

**Factory method:- one java class method is able to return same class object or different class object is called factory method.**

**Example :-**

```
class MyThread extends Thread
{
    public void run()
    {System.out.println("shutdown hook");
    }
};

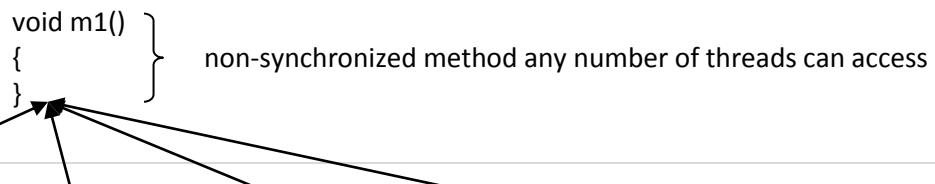
class ThreadDemo
{
    public static void main(String[] args) throws InterruptedException
    {
        MyThread t = new MyThread();
        //creating Runtime class Object by using factory method
        Runtime r = Runtime.getRuntime();
        r.addShutdownHook(t); //adding Thread to JVM hook
        for (int i=0;i<10 ;i++)
        {System.out.println("main thread is running");
        Thread.sleep(3000);
        }
    }
};

D:\DP>java ThreadDemo
main thread is running
main thread is running
main thread is running
main thread is running
shutdown hook
while running Main thread press Ctrl+C then hook thread will be executed.
```

**Synchronized :-**

- Synchronized modifier is the modifier applicable for methods but not for classes and variables.
- If a method or a block declared as synchronized then at a time only one Thread is allowed to operate on the given object.
- The main advantage of synchronized modifier is we can resolve data inconsistency problems.
- But the main disadvantage of synchronized modifier is it increases the waiting time of the Thread and effects performance of the system .Hence if there is no specific requirement it is never recommended to use.
- The main purpose of this modifier is to reduce the data inconsistence problems.

#### Non-synchronized methods

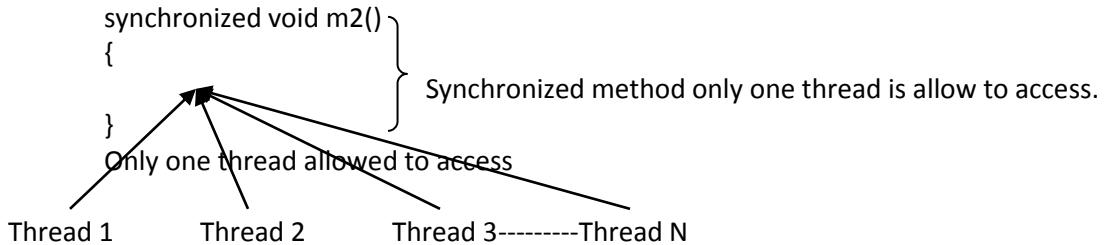


Every thread accessing simultaneously

Thread 1      Thread 2      Thread 3-----Thread N

- 1) In the above case multiple threads are accessing the same methods hence we are getting data inconsistency problems. These methods are not thread safe methods.
- 2) But in this case multiple threads are executing so the performance of the application will be increased.

### Synchronized methods



- 1) In the above case only one thread is allowed to operate on particular method so the data inconsistency problems will be reduced.
- 2) Only one thread is allowed to access so the performance of the application will be reduced.
- 3) If we are using above approach there is no multithreading concept.

Hence it is not recommended to use the synchronized modifier in the multithreading programming.

### 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 synchronize 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 Sravyasoft");
    }
};

class MyThread1 extends Thread
{
    Heroin h;
    MyThread1(Heroin h)
    {this.h=h;}
    public void run()
    {
        h.message("Anushka");
    }
};

class MyThread2 extends Thread
{
    Heroin h;
    MyThread2(Heroin h)
    {this.h=h;
}
```

```

        }
    public void run()
    {
        h.message("Ratan");
    }
};

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();
    }
};

```

**Daemon threads:-**

The threads which are executed at background is called daemon threads.

Ex:- garbage collector, ThreadScheduler.default exceptional handler.

Non-daemon threads:-

The threads which are executed fore ground is called non-daemon threads.

Ex:- normal java application.

- When we create a thread in java that is user defined thread and if it is running JVM will not terminate that process.
- If a thread is marked as a daemon thread JVM does not wait to finish and as soon as all the user defined threads are finished then it terminates the program and all associated daemon threads.
- Set the daemon nature to thread by using setDaemon() method
  - MyThread t = new Mythread();  
t.setDaemon(true);
- To know whether a thread is daemon or not use isDaemon() method
  - Thread.currentThread().isDaemon();

```

class MyThread extends Thread
{
    void message(String str)
    {
        try
        {
            System.out.println("message="+str);
            Thread.sleep(1000);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }

    public void run()
    {
        if(Thread.currentThread().isDaemon())
        {
            while (true)
            {
                message("print hi ratan");
            }
        }
    }
};

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

```

```

    {
        MyThread t = new MyThread();
        t.setDaemon(true); //setting daemon nature to Thread
        t.start();
        try{Thread.sleep(5000);}
        catch(InterruptedException e)
        {e.printStackTrace();}
        System.out.println("main thread completed");
    }
};

```

Note :- in above example make the setdaemon() is comment mode then the program never terminates even main thread finished it's execution.

```

class MyThread extends Thread
{
    int total;
    public void run()
    {
        synchronized(this){
            for (int i=0;i<10 ;i++)
            {
                total=total+i;
            }
            notify();
        }
    }
}
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t = new MyThread();
        t.start();
        synchronized(t)
        {
            System.out.println("MyThrad total is waiting for MyThread completion...");
            try{
                t.wait();
            catch(InterruptedException ie){System.out.println(ie);}
            }
            System.out.println("MyThrad total is =" +t.total);
        }
    }
};

```

### Volatile:-

- Volatile modifier is also applicable only for variables but not for methods and classes.
- If the values of a variable keep on changing such type of variables we have to declare with volatile modifier.
- If a variable declared as a volatile then for every Thread a separate local copy will be created.
- Every intermediate modification performed by that Thread will take place in local copy instead of master copy.
- Once the value got finalized just before terminating the Thread the master copy value will be updated with the local stable value. The main advantage of volatile modifier is we can resolve the data inconsistency problem.
- But the main disadvantage is creating and maintaining a separate copy for every Thread
- Increases the complexity of the programming and effects performance of the system.

### Nested classes

- Declaring the class inside another class is called nested classes. This concept is introduced in the 1.1 version.
- Without existing one type of object there is no chance of existing another type of object we should use Inner classes.
- Declaring the methods inside another method is called inner methods java not supporting inner methods concept.

**Example 1:-** University contains several departments but without university no chance of existing departments.

```
class University           //outer class
{
    class Department    //inner class
    {
    }
}
}
```

**Example 2:-** without existing bank object no chance of existing account object.

```
class Bank           //outer class
{
    class Account   //inner class
    {
    }
}
}
```

**Example 3:-** Map is a collection of key-value pairs & each key-value pair is entry. So map contains group of entry's .The entry can exists with presents of Map only.

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

There are two types of nested classes in java,

1. **Static nested classes**
2. **Non static nested classes( these are called inner classes)**
  - a. **Normal inner classes**
  - b. **Method local inner classes**
  - c. **Anonymous inner classes**



**Uses of nested classes:-**

- ✓ **It is the way logically grouping classes that are only used in the one place.**

If one class required another class only one time then it is logically embedded it into that classes make the two classes together.

- ✓ **it increase the encapsulation**

If we are taking two top level classes A and B the B class need the members of A that members even we are declaring private modifier the B class can access the private numbers moreover the B is not visible for outside the world.

- ✓ **It lead the more readability and maintainability of the code**

Nesting the classes within the top level classes at that situation placing the code is very closer to the top level class.

- ✓ **Code optimization** it reduce the length of the code.

**Example :-**

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

**Note 1:** The compiler will generate .class files for both inner & outer classes.

Outer class .class file name : Outer.class

Inner class .class file name : Outer\$Inner.class

**Note 2:-** it is possible to create the objects for both inner & outer classes.

Outer class Object creation : Outer o = new Outer();

Inner class object creation : Outer.inner I = new Outer().new inner();

**Note 3:-** By using outer class object it is possible to call only outer class properties & by using inner class object it is possible to call only inner class members.

**Note 4:-** The inner class methods are able to access outer class methods but outer class methods are unable to access inner class methods.

**Inner classes vs nested classes:-**

Inner classes are part of nested classes. The non-static nested classes are called inner classes.

**1. Member inner classes or regular inner classes or normal inner classes:-**

- ✓ Declaring the class inside another class is called normal inner classes.
- ✓ The inner class is able to access outer class private properties

**Syntax:-**

```
class Outer
{
    class Inner
    {
    };
}
```

**Example -1:-**

```
class Outer
{
    private int a=10;
    private int b=20;
    void m1()
    {
        System.out.println("outer class m1()");
    }
    class Inner
    {
        int i=100;
        int j=200;
        void m2()
        {
            System.out.println("inner class m2()");
            System.out.println(a+b);
            System.out.println(i+j);
            m1();
        }
    };
}
class Test
{
    public static void main(String... ratan)
    {
        Outer o = new Outer();
        o.m1();
        Outer.Inner i = o.new Inner();
        i.m2();
    }
};
```

**Observation :-** Reduce the length of the code

```
public static void main(String[] ratan)
{
    new Outer().m1();
    new Outer().new Inner().m2();
}
```

**Example 2:-** inside the inner classes it is not possible to declare static members hence inside the inner classes it is not possible to declare main method because main is static.

```
class Outer
{
    class Inner
    {
        public final static int a=10;
    }
}
error: Illegal static declaration in inner class Outer.Inner
```

**Example 3:-**

```

class Outer
{
    private int a=10;
    private int b=20;
    class Inner
    {
        int a=100;
        int b=200;
        void m1(int a,int b)
        {
            System.out.println(a+b);           //local variables
            System.out.println(this.a+this.b);  //Inner class variables
            System.out.println(Outer.this.a+Outer.this.b); //outer class variables
        }
    };
}
class Test
{
    public static void main(String... ratan)
    {
        new Outer().new Inner().m1(1000,2000);
    }
};

```

**Example 4:-**

```

class Outer
{
    void m1(){ System.out.println("outer class m1()"); }
    class Inner
    {
        void m1()
        {
            Outer.this.m1();
            System.out.println("inner class m1()");
        }
    };
}
class Test
{
    public static void main(String[] ratan)
    {
        Outer.Inner i = new Outer().new Inner();
        i.m1();
    }
};

```

**Example 5:- Inside the inner & outer classes it is possible to declare the constructor.**

```

class Outer
{
    Outer()
    {
        System.out.println("outer class cons");
    }
    class Inner
    {
        Inner()
        {
            System.out.println("Inner class cons");
        }
    }
    public static void main(String[] args)
    {
        new Outer().new Inner();
    }
}

```

**2. Method local inner classes:-**

- ✓ Declaring the class inside the method is called method local inner classes.
- ✓ The scope is only within the method.it means whenever the method is completed inner class object destroyed.
- ✓ The applicable modifiers on method local inner classes are **final & abstract** .

**Syntax:-**

```
class Outer
{
    void m1()
    {
        class Inner      //method local inner class
        {
        };
    }
};
```

**Example-1:-**

```
class Outer
{
    private int a=100;
    void m1()
    {
        class Inner
        {
            void m2()
            {
                System.out.println("inner class method");
                System.out.println(a);
            }
        }
        Inner i=new Inner();
        i.innerMethod();
    }
};
```

**class Test**

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

**Example 2:-**

```
class Outer
{
    void show()
    {
        for (int i=0;i<10 ;i++ )
        {
            class Inner
            {
                public void info()
                {
                    System.out.println("method local inner class");
                }
            }
            new Inner().info();
        }
    }
    public static void main(String[] args)
    {
        new Outer().show();
    }
}
```

**Example 3:-**

```

class Outer
{
    private int a=100;
    void m1()
    {final int b=200;//local variables must be final variables(upto 1.7 version) // 1.8 verison optional
        class Inner
        {
            void m1()
            {
                System.out.println("inner class m1()");
                System.out.println(a);
                System.out.println(b);
            }
        };
        Inner i = new Inner();
        i.m1();
    }
    public static void main(String[] args)
    {
        new Outer().m1();
    }
}

```

**3. Static nested classes:-**

- ✓ Declaring the class inside another class with static modifier is called static inner classes.
- ✓ The normal inner class is able to access both static & non-static members of outer class but static inner class is able to access only static members of outer class.

**Normal inner classes:-** Allows both static & non-static members of Outer class.

```

class Outer
{
    int a=10;      //instance variable
    static int b=20; //static variable
    class Inner
    {
        void m1()
        {
            System.out.println(a);
            System.out.println(b);
        }
    };
}

```

**Static nested classes:-** Allows only static members of outer class.

```

class Outer
{
    static int a=10;      //static variable
    int b=20;             //instance variable
    static class Inner
    {
        void m1()
        {
            System.out.println(a);
            System.out.println(b); //compilation error
        }
    };
}

```

**Example :-**

```

class Outer
{
    static int a=10;
    static int b=20;
    static class Inner
    {
        int c=30;
        void m1()
        {
            System.out.println(a);
            System.out.println(b);
            System.out.println(c);
        }
    };
    public static void main(String[] args)
    {
        Outer o=new Outer();
        Outer.Inner i=new Outer.Inner();
        i.m1();
    }
}

```

**4. Anonymous inner class:-**

- ✓ The name less inner class in java is called anonymous inner class.
- ✓ it can be used to provide the implementation of normal class or abstract class or interface

**Application without anonymous inner classes:**

```

class A //assume predefined class
{
    void m1(){}
    void m2(){}
}
class Test extends A //user class contains logics
{
    void m1(){System.out.println("m1 method");}
    void m2(){System.out.println("m2 method");}
}
class TestClient
{
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
        t.m2();
    }
}

```

**Application with anonymous inner classes:-**

```

class A //assume predefined class
{
    void m1(){}
    void m2(){}
}
class TestClient
{
    A a = new A()
    {
        void m1(){System.out.println("m1 method");}
    }
}

```

```

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

```

**Example 2:-**

```

class A           //predefined class
{
    void m1(){}
}

class TestClient
{
    A a = new A()
    {
        void m1()
        {
            System.out.println("m1 method");
            System.out.println(a.getClass().getName());
        }
    };
    public static void main(String[] args)
    {
        TestClient t = new TestClient();
        t.a.m1();
    }
}
G:\>java TestClient
m1 method
TestClient$1

```

In above example when we create the object of **A** class internally A class object is not creating, the compiler will generate one new class that class object will be created.

To get the compiler generated class use `getClass()` method of `Object` class.

***TestClient\$1 (.class file code ) open by using java de-compiler software***

```

class TestClient$1 extends A
{
    void m1()
    {
        System.out.println("m1 method");
        System.out.println(a.getClass().getName());
    }
    final TestClient this$0;
    TestClient$1()
    {
        this$0 = TestClient.this;
        super();
    }
}

```

**Example 3:**

```

abstract class Animal
{
    abstract void eat();
};

class Test
{
    //anonymous inner class
    Animal a=new Animal()
    {
        void eat()
        {
            System.out.println("animals eating gross");
        }
    }; //semicolon mandatory
public static void main(String[] args)
{
    Test t=new Test();
    t.a.eat();
}
}

```

**Example 4:-****Application without anonymous inner classes**

```

interface It1
{
    void m1();
}

class Test implements It1
{
    public void m1(){System.out.println("m1 method");}
}

class TestClient
{
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
    }
}

```

**Application with anonymous inner classes**

```

interface It1
{
    void m1();
}

class TestClient
{
    It1 i = new It1()
    {
        public void m1(){System.out.println("m1 method");}
    };
    public static void main(String[] args)
    {
        TestClient t = new TestClient();
        t.i.m1();
    }
}

```

*The applicable modifiers on Outer classes*

- ✓ *Public*
- ✓ *Abstract*
- ✓ *Strictfp*
- ✓ *Final*
- ✓ *Default (no modifier)*

*The applicable modifiers on inner classes*

- ✓ *final*
- ✓ *abstract*
- ✓ *public*
- ✓ *private*
- ✓ *protected*
- ✓ *strictfp*
- ✓ *static*
- ✓ *default (no modifier)*

**Example 1:- *private*** : it is possible to access private members inside the class only.

```
class Outer
{
    private class Inner1
    {
    }
}

class Test
{
    public static void main(String[] args)
    {
        Outer.Inner1 i = new Outer().new Inner1();
    }
}
```

**Compilation Error: Outer.Inner1 has private access in Outer**

**Example 2:- *abstract*** : it is not possible to instantiate the class.

```
class Outer
{
    abstract class Inner
    {
    }
}

class Test
{
    public static void main(String[] args)
    {
        Outer.Inner i = new Outer().new Inner();
    }
}
```

**Compilation Error: Outer.Inner is abstract; cannot be instantiated**  
**Outer.Inner i = new Outer().new Inner();**

**Example 3:- final :** it is not possible to create sub class.

**Case 1:-**

```
class Outer
{
    class Inner1
    {
        void m1(){System.out.println("Inner1 method");}
    }
    class Inner2 extends Inner1
    {
        void m2(){System.out.println("Inner2 method");}
    }
}
class Test
{
    public static void main(String[] args)
    {
        Outer.Inner2 i = new Outer().new Inner2();
        i.m1();
        i.m2();
    }
}
```

**Case 2:-**

```
class Outer
{
    final class Inner1
    {
    }
    class Inner2 extends Inner1
    {
    }
}
```

**Compilation error : cannot inherit from final Outer.Inner1**

**Some possibilities:-**

**Case 1:**

```
class A
{
    class B
    {
    }
}
```

**Case 2:-**

```
class A
{
    interface It1
    {
    }
}
```

**Case 3:-**

```
interface It1
{
    interface It2
    {
    }
}
```

**Case 4:-**

```
interface It1
{
    class A
    {
    }
}
```

**Functional interface:-**

- ✓ It is introduced in jdk1.8 version & it has exactly one abstract method.
- ✓ This interface is also known as single abstract method interface (SAM interface).
- ✓ Note that instances of functional interfaces can be created with lambda expressions, method references, or constructor references.
- ✓ Java 8 introduced @FunctionalInterface annotation which can be used for compilation errors when the interface you annotated violates the contracts of functional interface.

Example :    @FunctionalInterface  
               interface Greetings  
               {     void morning();  
               }

Example : if we declare @FunctionalInterface annotation but if we are declaring more than one abstract method then compiler will generate error message.

```
@FunctionalInterface
interface Greetings
{     void morning();
    void m1();
}
```

E:\>javac Durga.java

Test.java:1: error: Unexpected @FunctionalInterface annotation @FunctionalInterface

**Greetings is not a functional interface**

**multiple non-overriding abstract methods found in interface Greetings**

- ✓ Functional interface allows only one abstract method it is not allowed second abstract method in functional interface. If we remove @FunctionalInterface annotation then we are allowed to add second abstract method but it is not a functional interface.

**xample :-**

```
interface Executable
{     void execute();
}
class Runner
{     public void run(Executable e)
{       System.out.println("run method code.... ");
e.execute();
}
}
public class Test
{     public static void main(String[] args)
{       Runner r = new Runner();
//anonymous inner class
r.run(new Executable(){
    public void execute()
{           System.out.println("execute method block of java code..... ");
}
});
//lambda expression
System.out.println("=====");
r.run(() -> System.out.println("execute method block of code..... "));
```

```

        }
};

E:\>java Test
run method code....
execute method block of java code.....
=====
run method code....
execute method block of code.....
Observation-1:- to write multiple statements(more lines of code).
    r.run(() -> {System.out.println("execute method block of code-1.....");
                  System.out.println("execute method block of code-2.....");
                });
E:\>java Test
run method code....
execute method block of java code.....
=====
run method code....
execute method block of code-1.....
execute method block of code-2.....

```

**Example :-**

```

interface Executable
{
    int execute();
}

class Runner
{
    public void run(Executable e)
    {
        System.out.println("run method code....");
        int x = e.execute();
        System.out.println("return value="+x);
    }
};

public class Test
{
    public static void main(String[] args)
    {
        Runner r = new Runner();
        //anonymous inner class
        r.run(new Executable(){
            public int execute()
            {
                System.out.println("execute method anonymous block of java code....");
                return 10;
            }
        });
        //lambda expression code
        System.out.println("=====");
        r.run(() -> {System.out.println("execute method lambda expression code....");
                      return 20;
                    });
    }
}

```

```
E:\>java Test
run method code....
execute method anonymous block of java code.....
return value=10
=====
```

```
run method code....
execute method lambda expression code.....
return value=20
```

**Observation:-**

If you don't write any code just you want return value use below code  
*r.run(() -> 20);*

```
E:\>java Test
run method code....
execute method anonymous block of java code.....
return value=10
=====
run method code....
return value=20
```

**Example :-**

```
interface Executable
{
    int execute(int a,int b);
}

class Runner
{
    public void run(Executable e)
    {
        System.out.println("run method code....");
        int x = e.execute(100,200);
        System.out.println("return value="+x);
    }
};

public class Test
{
    public static void main(String[] args)
    {
        Runner r = new Runner();
        //anonymous inner class code
        r.run(new Executable(){
            public int execute(int a,int b)
            {
                System.out.println("execute method anonymous block of java code.... ");
                return 10+a;
            }
        });
        //lambda expression code
        System.out.println("=====");
        r.run((int a,int b) -> 20+a+b);
    }
};

E:\>java Test
run method code....
```

```
execute method anonymous block of java code.....
return value=110
=====
run method code....
return value=320
```

***Observation :-***

```
r.run((a,b) -> 20+a);
E:\>java Test
run method code....
execute method anonymous block of java code.....
return value=110
=====
run method code....
return value=120
```

***Example:- lambda expression ambiguity problems***

```
interface Executable
{
    int execute(int a);
}

interface StringExecutable
{
    int execute(String a);
}

class Runner
{
    public void run(Executable e)
    {
        System.out.println("run method code....");
        int x = e.execute(10);
        System.out.println("return value="+x);
    }

    public void run(StringExecutable e)
    {
        System.out.println("run method code....");
        int x = e.execute("ratan");
        System.out.println("return value="+x);
    }
};

public class Test
{
    public static void main(String[] args)
    {
        Runner r = new Runner();
        //lambda expression
        System.out.println("=====");
        r.run((int a) -> 10);
        r.run((String a) -> 20);
        // r.run(a ->100); compilation error: ambiguity problem
    }
};
```

**Example :-**

```

interface Executable
{
    int execute(int a,int b);
}

class Runner
{
    public void run(Executable e)
    {
        System.out.println("run method code....");
        int x = e.execute(100,200);
        System.out.println("return value="+x);
    }
};

public class Test
{
    public static void main(String[] args)
    {
        int c=10;
        //c=c+10; must be comments
        Runner r = new Runner();
        //anonymous inner class
        r.run(new Executable(){
            public int execute(int a,int b)
            {
                System.out.println("execute method anonymous block of java code.....");
                return a+b+c;
            }
        });
        //lambda expression
        System.out.println("=====");
        r.run((int a,int b) -> a+b+c);
    }
};

E:\>java Test
run method code....
execute method anonymous block of java code.....
return value=310
=====
run method code....
return value=310

```

**Observation:-**

If we remove comments on `c=c+10` then we will get following two errors,

```

E:\>javac Test.java
Test.java:20: error: local variables referenced from an inner class must be final or effectively final
    return a+b+c;
               ^
Test.java:25: error: local variables referenced from a lambda expression must be final or effectively final
    r.run((int a,int b) -> a+b+c);
               ^
2 errors

```

**Annotations :-(meta data)**

- ✓ Annotations are introduced in 1.5 version it represent metadata of the program.
- ✓ Annotations can be used attach the some additional information to the interfaces, classes, methods, constructors which can be used by compiler and JVM.
- ✓ Annotations are executed by using predefined tool APT(Annotation Processing Tool ).

**Meta annotations:-**

- ✓ it specify information about annotation.
- ✓ These annotations are present in **java.lang.annotation** package.
- ✓ The meta annotations are
  - **@Retention**
  - **@Target**
  - **@Documented**
  - **@Inherited**
  - **@Repeatable**

**Annotation Retention policy:- @Retention**

Retention policy determines at what point the annotation is available

**@Retention(RetentionPolicy.SOURCE)**

The marked annotation is retained only in the source level and is ignored by the compiler.

**@Retention(RetentionPolicy.CLASS)**

The marked annotation is retained by the compiler at compile time, but is ignored by the Java Virtual Machine (JVM).

**@Retention(RetentionPolicy.RUNTIME)**

The marked annotation is retained by the JVM so it can be used by the runtime environment

**Annotation Target specification: @Target**

- ✓ Marks another annotation to restrict what kind of Java elements the annotation may be applied to.
  - ✓ The target annotation specify one of the fallowing element,
- |  |  |
|--|--|
| <b>@Target (ElementType.ANNOTATION_TYPE)</b> | can be applied to an annotation type.                  |
| <b>@Target (ElementType.CONSTRUCTOR )</b>    | can be applied to a constructor.                       |
| <b>@Target ( ElementType.FIELD )</b>         | can be applied to a field or property.                 |
| <b>@Target (ElementType.LOCAL_VARIABLE)</b>  | can be applied to a local variable.                    |
| <b>@Target (ElementType.METHOD )</b>         | can be applied to a method-level annotation.           |
| <b>@Target (ElementType.PACKAGE)</b>         | can be applied to a package declaration.               |
| <b>@Target (ElementType.PARAMETER)</b>       | can be applied to the parameters of a method.          |
| <b>@Target (ElementType.TYPE)</b>            | can be applied to any class, interface or enumeration. |

**@Documented - Marks another annotation for inclusion in the documentation.**

Whenever we are using this annotation those elements should be documented by using javadoc tool.

**Syntax:-**

Annotation can be declared using '@' character as prefix of annotation name.

```
@Annotation
public void annotatedMethod()
{
    //logics here
}
```

Annotations has elements in the form of key=value ,the elements are properties of annotation.

```
@Annotataion(name="ratan",age="28")
public void annotatedMethod()
{
    //logics here
}
```

If the annotation contains single we can use like this.

```
@Annotataion("I am hero")
public void annotatedMethod()
{
    //logics here
}
```

It possible to declare multiple annotations at class-level

```
@Annotataion1(name="ratan")
@Annotataion2
public void annotatedMethod()
{
    //logics here
}
```

### Uses of annotations:-

#### ❖ Information for the compiler

Annotations are used by the compiler to detect suppress warnings or errors based on rules.

Example :- **@Override** this one makes the compiler to check the method correctly override or not

**@FunctionalInterface** this one makes the compiler to validate annotated interface is functional interface or not.

#### ❖ Documentation

Annotations can be used in software applications to ensure the quality of the code like bug finding, report generation...etc

#### ❖ Code generation

Annotations used to generate the code or xml files using meta data information present in the code.

#### ❖ Runtime processing

Annotations that are used in runtime objectives like unit testing, dependency injection...etc

There are three types of annotations

- 1) Java built in annotations
- 2) Marker annotations
- 3) Custom annotation
- 4) Meta annotations

**@Override - Checks that the method is an override. Causes a compile error if the method is not found in one of the parent classes or implemented interfaces.**

```

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}

@override :-
✓ It instructs the compiler to check parent class method is overriding in child class or not if it is not overriding compiler will generate error message.
✓ In below example if are not declaring @override annotation a new method of marry(int a) created in child class.

class Parent
{
    void marry(String name)
    { //logics-here }
};

class Child extends Parent
{
    @Override
    void marry(int n)
    { //logics here  }
};

E:\>javac Test.java
error: method does not override or implement a method from a supertype
    @Override

```

**@Deprecated - Marks the method as obsolete. Causes a compile warning if the method is used.**

This annotation represent the marked element is no longer be used. The compiler generates warning message when we used that marked element.

**Example :-**

```

@Deprecated
class Test
{
    @Deprecated
    void m1()
    { //logics here
    }
}

class Demo
{
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
    }
};

E:\>javac Test.java

```

Note: Test.java uses or overrides a deprecated API.

Note: Recompile with -Xlint:deprecation for details.

```

E:\>javac -Xlint Test.java
Test.java:10: warning: [deprecation] Test in unnamed package has been deprecated
    Test t = new Test();
Test.java:10: warning: [deprecation] Test in unnamed package has been deprecated
    Test t = new Test();

```

```
Test.java:11: warning: [deprecation] m1() in Test has been deprecated
    t.m1();
3 warnings
```

**Example -2:-**

```
import java.awt.*;
class Student
{
    public static void main(String[] args)
    {
        Frame f = new Frame();
        f.show();
    }
};
```

E:\>javac Test.java

Note: Test.java uses or overrides a deprecated API.

Note: Recompile with -Xlint:deprecation for details.

**@SuppressWarnings - Instructs the compiler to suppress the compile time warnings specified in the annotation parameters.**

```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
    String[] value();
}
```

Whenever we are using deprecated annotate method then compiler will generate warning messages but to ignore that warning messages use @suppressWarings annotation.

When you compiled below application it won't generate any warnings even it uses @Deprecated annotation.

**Example-1 :-**

```
class Test
{
    @Deprecated
    void m1()
    {//logics here
    }
}
class Demo
{
    @SuppressWarnings("deprecation")
    public static void main(String[] args)
    {
        new Test().m1();
    }
};
```

**Example-2:-**

```
import java.util.*;
class Student
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add("ratan");
        al.add("anu");
        al.add("sravya");
```

```

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

@SafeVarargs - Suppress warnings for all callers of a method or constructor with a generics varargs parameter, since Java 7.
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD})
public @interface SafeVarargs { }

```

**@FunctionalInterface - Specifies that the type declaration is intended to be a functional interface, since Java 8. Annotations applied to other annotations (also known as "Meta Annotations"):**

```

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FunctionalInterface { }

```

**@Inherited -**

Marks another annotation to be inherited to subclasses of annotated class (by default annotations are not inherited to subclasses).

**Example :-**

```

@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface MyAnnotation
{ }

```

- ✓ In above example your annotation can be applied only on methods & constructors because you specified this information by using **@Target** annotation.
- ✓ This annotation is used by JVM at runtime because we specified this information by using **@Retention**.
- ✓ Basically parent class annotation is not visible in child classes but it is possible to inherit parent call annotation in child class by using **@Inherited** annotation.
- ✓ we can create documentation by using javadoc tool because we declared annotation by using **@Documented** annotation.

**@Repeatable - Specifies that the annotation can be applied more than once to the same declaration, since Java 8.**

**Customized Annotations:-**

- ✓ Annotations can be created by using **@interface** followed by annotation name.

```

public @interface MyAnnotation
{
}

```

- ✓ Every annotation is extending `java.lang.annotation.Annotation` interface hence annotation can't include extend clause.

```
public interface MyAnnotation extends Annotation
{
}
```

**Example :**

**Step 1: create the annotation**

```
import java.lang.annotation.*;
import java.lang.reflect.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface MyAnnotation
{
    int value();
}
```

**Step-2: Apply the annotation**

```
class Test
{
    @MyAnnotation(value=100)
    public void m1()
    {
        System.out.println("m1 method");
    }
}
```

**Step 3: Accessing annotation**

```
import java.lang.reflect.*;
class TestClient
{
    public static void main(String[] args) throws NoSuchMethodException
    {
        Test t = new Test();
        Method m = t.getClass().getMethod("m1");
        MyAnnotation mm = m.getAnnotation(MyAnnotation.class);
        System.out.println("value is="+mm.value());
    }
}
```

**Example :-**

**File-1: ProjectInfo.java**

```
import java.lang.annotation.*;
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface ProjectInfo
{
    int pid();
    String pname() default "bank";
    int pteamsize();
    String pstatus();
}
```

**File-2:- Emp.java**

```
import java.lang.annotation.*;
@ProjectInfo(pid=111,pstatus="not released",pteamsize=5)
class Emp
{
    int eid;
    String ename;
    Emp(int eid,String ename)
    {
        this.eid=eid;
        this.ename=ename;
    }
    void disp()
    {
        System.out.println("****Employee details****");
        System.out.println("emp id="+eid);
        System.out.println("emp name="+ename);
    }
    public static void main(String[] args)
    {
        Emp e = new Emp(111,"ratan");
        e.disp();
        Class c = e.getClass();
        Annotation a = c.getAnnotation(ProjectInfo.class);
        ProjectInfo p = (ProjectInfo)a;
        System.out.println("****project details details****");
        System.out.println("project id="+p.pid());
        System.out.println("project name="+p.pname());
        System.out.println("project status="+p.pstatus());
        System.out.println("project teamsizes="+p.pteamsize());
    }
}
```

**Observation :-**

```
E:\>javadoc Emp.java
Loading source file Emp.java...
Constructing Javadoc information...
Standard Doclet version 1.8.0_65
Building tree for all the packages and classes...
```

**Observation :-**

```
@Target({ElementType.METHOD}) //method level we can use
```

```
@ProjectInfo(pid=111,pstatus="not released",pteamsize=5) //but we are using at class level
public class Emp
{ //logics here }
```

```
E:\>javac Emp.java
```

*Emp.java:2: error: annotation type not applicable to this kind of declaration*

```
@ProjectInfo(pid=111,pstatus="not released",pteamsize=5)
```

**Observation :-**

```
@Retention(RetentionPolicy.SOURCE) // ignored at source file level or
@Retention(RetentionPolicy.CLASS) //ignored at class level
```

```
E:\>java Emp
```

*\*\*\*Employee details\*\*\**

```
emp id=111
```

```
emp name=ratan
```

*\*\*\*project details details\*\*\**

```
Exception in thread "main" java.lang.NullPointerException
```

**Observation :-**

```
@Documented //if we removed this annotation the documentation is not performed
```

**ENUMARATION (1.5 version)**

- ✓ Enumeration is used to declare group of named constants.
- ✓ You define an enum type by using the enum keyword. For example, you would specify a days-of-the-week enum type as,

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

- ✓ Every enum constant represents object of type enum.

**Enum representation****enum Directions**

```
{      NORTH, SOUTH, EAST, WEST;
}
```

**Internal implementation:-****class Directions**

```
{      public static final Directions NORTH = new Directions();
      public static final Directions SOUTH = new Directions();
      public static final Directions EAST = new Directions();
      public static final Directions WEST = new Directions();
}
```

- ✓ Compiler will generate .class file for enum.
- ✓ The enum constants are by default **public static final**.
- ✓ The main purpose of the enum is to declare the own data types.

**Java enum is powerfull compare to other language enums:-**

- ✓ Enum improves the type safety.
- ✓ In java it is possible to declare fields, methods, constructors whereas it is not possible in other languages.
- ✓ Enum may implements interfaces.
- ✓ Inside the switch it is possible to use enum.
- ✓ Inside the enum it is possible to take main method also.

**Example :**

```

enum EnumMail
{
    INBOX,COMPOSE,SENT;
}

class ClassMail
{
    public static final int INBOX=1;
    public static final int COMPOSE=2;
    public static final int SENT=3;
}

class Test
{
    static void classConstants(int i)
    {
        if (i==ClassMail.INBOX)
        {
            System.out.println("mail inbox operation");
        }
        else if (i==ClassMail.COMPOSE)
        {
            System.out.println("mail compose operation");
        }
        else
        {
            System.out.println("mail sent operation");
        }
    }

    static void enumConstants(EnumMail e)
    {
        if (e==EnumMail.INBOX)
        {
            System.out.println("mail inbox operation");
        }
        else if (e==EnumMail.COMPOSE)
        {
            System.out.println("mail compose operation");
        }
        else
        {
            System.out.println("mail sent operation");
        }
    }

    public static void main(String[] args)
    {
        Test.classConstants(1);
        Test.classConstants(2);
        Test.classConstants(3);
        Test.classConstants(10); //no type safety we can pass any value

        Test.enumConstants(EnumMail.COMPOSE);
        Test.enumConstants(EnumMail.INBOX);
        Test.enumConstants(EnumMail.SENT);
        //here were providing type safety it means it is not possible to pass any other values
    }
}

```

- ✓ In above example `classConstants` methods it is possible to provide any value as a input but we can pass fixed values to `enumconstants` method, so it provides the type safety.
- ✓ When we change the constants value the program will not generate any error messages the program might not work as expected. But when we change the enum constants we will get compiler error which removes runtime error possibilities.

**Example :- Inside the enum `toString()` method overridden to return the name of the constant.**

```
public enum Heroin
{
    SAMANTHA,TARA,ANU;
}

class Test
{
    public static void main(String... ratan)
    {
        Heroin s=Heroin.SAMANTHA;
        System.out.println(s);
        Heroin t=Heroin.TARA;
        System.out.println(t);
        Heroin a=Heroin.ANU;
        System.out.println(a);
    }
};
```

**Internal implementation of above example :- (.class file code)**

```
public final class Heroin extends Enum
{
    public static Heroin[] values()
    {
        return (Heroin[])$VALUES.clone();
    }

    public static Heroin valueOf(String s)
    {
        return (Heroin)Enum.valueOf(Heroin, s);
    }

    private Heroin(String s, int i)
    {
        super(s, i);
    }

    public static final Heroin SAMANTHA;
    public static final Heroin TARA;
    public static final Heroin ANU;
    private static final Heroin $VALUES[];
    static
    {
        SAMANTHA = new Heroin("SAMANTHA", 0);
        TARA = new Heroin("TARA", 1);
        ANU = new Heroin("ANU", 2);
        $VALUES = (new Heroin[])
        {
            SAMANTHA, TARA, ANU
        };
    }
}
```

- ❖ Every enum constant by default **public static final**.
- ❖ Every enum constant represents object of type enum.  
`public static final Heroin SAMANTHA = new Heroin();`  
`public static final Heroin TARA = new Heroin();`  
`public static final Heroin ANU = new Heroin();`
- ❖ **Enum constructor is by default private hence it is not possible to create the object in outside of the enum.**
- ❖ Every enum constant contains index value & it starts from 0.
- ❖ Every enum is **final** by default hence other classes are unable to extend.

**Example :-**

- ✓ Values() method used to retrieve the all the constants at a time.
- ✓ Ordinal() method is used to print the index numbers of constants & index starts from 0.

```
public enum Heroin
{
    SAMANTHA,TARA,ANU;
}
class Test
{
    public static void main(String... ratan)
    {
        Heroin[] h = Heroin.values();
        for (Heroin hh :h)
        {
            System.out.println(hh+"----"+hh.ordinal());
        }
    }
};
```

**Example :-The enum constructor is by default private if we are declaring or not.**

- ✓ Inside the enum if we are declaring only constants then the group of constants ends with semicolon is optional.
- ```
public enum Beer
{
    KF,RC,KALYANI //ends with semicolon is optional
}
```
- ✓ Inside the enum it is possible to declare the constructor in this case the group of constants must be first line must ends with semicolon.

```
public enum Beer
{
    KF,RC,KALYANI; //semicolon is mandatory
    Beer()
    {
        System.out.println("Beer class constructor");
    }
}
```

```
public enum Beer
{
    KF,RC,KALYANI;
    Beer() // by default private modifier provided by compiler
    {
        System.out.println("Beer class constructor");
    }
}
class Test
{
    public static void main(String[] args)
    {
        Beer[] b = Beer.values();
        for (Beer bb :b)
        {
            System.out.println(bb);
        }
    }
}
```

G:\>java Test

**Beer class constructor**

**Beer class constructor**

**Beer class constructor**

**KF**

**RC**

**KALYANI**

- ❖ In above example every constant is object three times the constructor will be executed & it is a static hence it will be executed during .class file loading.

**Example :-** Inside the enum it is possible to declare parameterized constructors.

```
enum Heroin
{
    ANUSHKA,UBANU(10),DEEPIKA(10,20);
    Heroin() { System.out.println("ratan"); }
    Heroin(int a) { System.out.println("raghava"); }
    Heroin(int a,int b) { System.out.println("sanki"); }
}
class Test
{
    public static void main(String[] arhss)
    {
        Heroin[] h = Heroin.values();
        for (Heroin h1 : h)
        {
            System.out.println(h1+"---"+h1.ordinal());
        }
    }
};
```

**Example :** Constructor is initializing specific values enum constants.

```
enum Heroin
{
    ANUSHKA(10),UBANU(1),DEEPIKA(5);

    int rating;
    private Heroin(int rating)
    {
        this.rating=rating;
    }
}
class Test
{
    public static void main(String[] arhss)
    {
        Heroin[] h = Heroin.values();
        for (Heroin hh : h)
        {
            System.out.println(hh+"--Rating--"+hh.rating);
        }
    }
};
```

```
G:\>java Test
ANUSHKA--Rating--10
UBANU--Rating--1
DEEPIKA--Rating—5
```

**Example :**

Inside the enum it is possible to declare the main method but to execute this main method just execute the .class file.

```
enum Heroin
{
    ANUSHKA,UBANU,DEEPIKA;
    public static void main(String[] args)
    {
        System.out.println("enum class main method");
    }
}
G:\>java Heroin
enum class main method
```

**Example :-**

```
enum Heroin
{
    //group of constants
    ANUSHKA(10),UBANU(20),DEEPIKA(5);
    //variable declaration
    int value;
    //constructor declaration
    Heroin(int value)
    {
        this.value=value;
    }
    //enum main method
    public static void main(String[] args)
    {
        System.out.println("enum main method");
    }
}
class Test
{
    public static void main(String[] arhss)
    {
        //accessing enum constants
        Heroin[] h = Heroin.values();
        for (Heroin hh : h)
        {
            System.out.println(hh+"---"+hh.value);
        }
    }
}
G:\>java Test
ANUSHKA---10
UBANU---20
DEEPIKA---5
```

```
G:\>java Heroin
enum main method
```

**Declaring enum inside the class :-**

- ✓ If we declare the enum outside of the class the applicable modifiers are public,<default>, strictfp.
- ✓ If we are declaring enum inside the class the applicable modifiers are
  - Public , <default> , strictfp , private , protected , static
- ✓ It is not possible to declare the enum inside the methods it means locally.

```
class Gmail
{
    enum Mail{ INBOX,COMPOSE,SENT; } //; semicolon optional here
    public static void main(String[] args)
    {
        //class room code
        Mail[] mm = Mail.values();
        for (Mail mmm : mm)
        {
            System.out.println(mmm);
        }
        //project level code
        for (Mail m : Mail.values())
        {
            System.out.println(m);
        }
    }
}
```

**Example : Enum vs Switch**

```
enum Day{ SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY}
class Test
{
    public static void main(String args[])
    {
        Day day=Day.SUNDAY;
        switch(day)
        {
            case SUNDAY: System.out.println("sunday");
                           break;
            case MONDAY: System.out.println("monday");
                           break;
            default: System.out.println("other day");
        }
    }
}
```

In above example all case labels are must be enum constants otherwise compiler will generate error message .      case XXX: System.out.println("monday"); //error XXX not a enum type

**Example :-**

- ✓ Every enum is final by default hence it is not possible to extends other classes.
- ✓ Every enum is a final by default hence it is not possible to create child enums.

```
enum Day //every is enum is by default final
{ ****}
class Test extends Day
{****}
error: cannot inherit from final Day
```

**Example :** Every enum internally extends `java.lang.Enum` hence it is not possible to extends other classes.

```
class Test
{*****
enum Day extends Test
{*****}
```

**Example :** An enum can implements any number of interfaces.

```
interface It1
{*****
enum Day implements It1
{*****}
```

**Example :- Enum vs packages**

```
package pack1;
public enum Fish
{
    STAR,GOLD;
}
```

```
package pack2;
import static pack1.Fish.*;
class Test2
{
    public static void main(String[] args)
    {
        System.out.println(GOLD);
    }
}
```

```
package pack3;
import pack1.*;
class Test3
{
    public static void main(String[] args)
    {
        Fish f = Fish.STAR;
        System.out.println(f);
    }
}
```

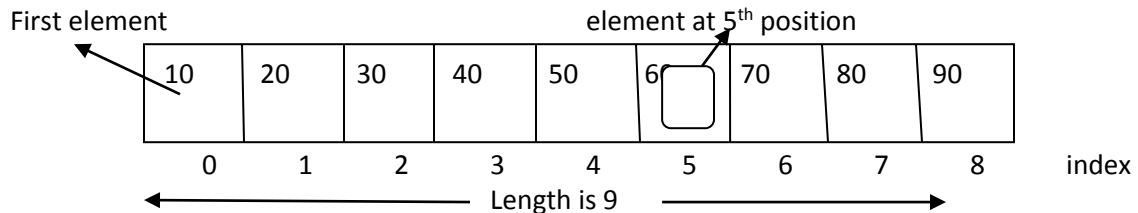
```
package pack4;
import pack1.*;
import static pack1.Fish.*;
class Test4
{
    public static void main(String[] args)
    {
        Fish f = Fish.STAR;
        System.out.println(GOLD);
    }
}
```

## Arrays

- ❖ Arrays are used to represent group of elements as a single entity but these elements are homogeneous & fixed size.
- ❖ The size of Array is fixed it means once we created Array it is not possible to increase and decrease the size.
- ❖ Array in java is index based first element of the array stored at 0 index.

### Advantages of array:-

- ✓ Instead of declaring individual variables we can declare group of elements by using array it reduces length of the code.
- ✓ We can store the group of objects easily & we are able to retrieve the data easily.
- ✓ We can access the random elements present in the any location based on index.
- ✓ Array is able to hold reference variables of other types.



### Different ways to declare a Array:-

```
int[] values;
```

```
int []values;
```

```
int values[];
```

### declaration & instantiation & initialization :-

```
Approach 1:- int a[]={10,20,30,40}; //declaring, instantiation, initialization
```

```
Approach 2:- int[] a=new int[100]; //declaring, instantiation
```

```
a[0]=10; //initialization
a[1]=20;
::::::::::::::::::
a[99]=40;
```

```
// declares an array of integers
```

```
int[] anArray;
```

```
// allocates memory for 10 integers
```

```
anArray = new int[10];
```

```
// initialize first element
```

```
anArray[0] = 10;
```

```
// initialize second element
```

```
anArray[1] = 20;
```

```
// and so forth
```

```
anArray[2] = 30; anArray[3] = 40; anArray[4] = 50; anArray[5] = 60;
```

```
anArray[6] = 70; anArray[7] = 80; anArray[8] = 90; anArray[9] = 100;
```

**Example :- 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);
        }
    }
}
```

**Example :- 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);
    }
}
```

**Method parameter is array & method return type is array:-**

```
class Test
{
    static void m1(int[] a) //method parameter is array
    {
        for (int a1:a)
        {
            System.out.println(a1);
        }
    }
    static int[] m2() //method return type is array
    {
        System.out.println("m1 method");
        return new int[]{100,200,300};
    }
    public static void main(String[] args)
    {
        Test.m1(new int[]{10,20,30,40});
        int[] x = Test.m2();
        for (int x1:x)
        {
            System.out.println(x1);
        }
    }
}
```

**Example:- adding the objects into Array and printing the objects.**

```

class Test
{
    public static void main(String[] args)
    {
        int[] a = new int[5];
        a[0]=111;
        for (int a1:a)
        {
            System.out.println(a1);
        }
        Emp e1 = new Emp(111,"ratan");
        Emp e2 = new Emp(222,"anu");
        Emp e3 = new Emp(333,"sravya");
        Emp[] e = new Emp[5];
        e[0]=e1;
        e[1]=e2;
        e[2]=e3;
        for (Emp ee:e)
        {
            System.out.println(ee);
        }
    }
}

```

**Output:-**

E:\>java Test  
111 0 0 0 0  
Emp@530daa Emp@a62fc3 Emp@89ae9e null null

**Example:- printing array elements with elements and default values.**

```

class Test
{
    public static void main(String[] args)
    {
        Emp[] e = new Emp[5];
        e[0]=new Emp(111,"ratan");
        e[1]=new Emp(222,"anu");
        e[2]=new Emp(333,"sravya");
        for (Object ee:e)
        {
            if (ee instanceof Emp)
            {
                Emp eee = (Emp)ee;
                System.out.println(eee.eid+"----"+eee.ename);
            }
            if (ee==null)
            {
                System.out.println(ee);
            }
        }
    }
}

```

**Output:-**

E:\>java Test  
111----ratan  
222----anu  
333----sravya  
null  
null

**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);
    }
}

```

**Example :- copy the data from one array to another array**

```

class Test
{
    public static void main(String[] args)
    {
        int[] copyfrom={10,20,30,40,50,60,70,80};
        int[] copyto = new int[7];
        System.arraycopy(copyfrom,1,copyto,0,7);
        for (int cc:copyto)
        {
            System.out.println(cc);
        }
    }
}

```

**Example :- copy the data from one array to another array**

```

class Test
{
    public static void main(String[] args)
    {
        int[] copyfrom={10,20,30,40,50,60,70,80};
        int[] newarray=java.util.Arrays.copyOfRange(copyfrom,1,4);
        for (int aa:newarray)
        {
            System.out.println(aa); //20 30 40
        }
    }
}

```

**Example:- 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);
            }
        }
    }
}

```

**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());
    }
}

```

**Example:-process of adding different types Objects in Object array**

```

class Test
{
    public static void main(String[] args)
    {
        Object[] a= new Object[6];
        a[0]=new Emp(111,"ratan");
        a[1]=new Integer(10);
        a[2]=new Student(1,"anu");
        for (Object a1:a)
        {
            if (a1 instanceof Emp)
            {
                Emp e1 = (Emp)a1;
                System.out.println(e1.eid+"---"+e1.ename);
            }
            if (a1 instanceof Student)
            {
                Student s1 = (Student)a1;
                System.out.println(s1.sid+"---"+s1.sname);
            }
            if (a1 instanceof Integer)
            {
                System.out.println(a1);
            }
            if (a1==null)
            {
                System.out.println(a1);
            }
        }
    }
}

```

**Emp.java:**

```
class Emp
{
    int eid; String ename;
    Emp(int eid, String ename)
    {
        //conversion of local to instance
        this.eid=eid;
        this.ename=ename;
    }
}
```

**Student.java:-**

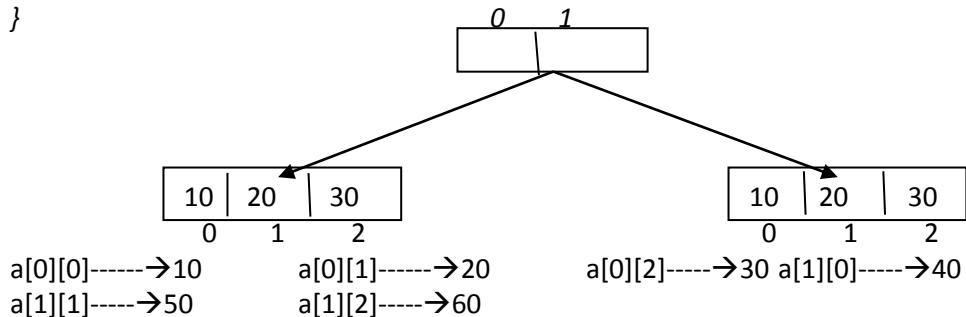
```
class Student
{
    int sid; String sname;
    Student(int sid, String sname)
    {
        //conversion of local to instance
        this.sid=sid;
        this.sname=sname;
    }
}
```

**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
    }
}
```

**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]);
    }
}
```

## Collections framework (java.util)

### Pre-requisite topics for Collections framework:-

- 1) **Arrays**
- 2) **toString() method.**
- 3) **type-casting.**
- 4) **interfaces.**
- 5) **for-each loop.**
- 6) **implementation classes.**
- 7) **compareTo() method.**
- 8) **Wrapper classes.**
- 9) **Marker interfaces advantages.**
- 10) **Anonymous inner classes.**
- 11) **For-each loop**
- 12) **Auto-boxing**

### Importance of collections:-

- The main objective of collections framework is to represent group of object as a single entity.
- In java Collection framework provide very good architecture to store and manipulate the group of objects.
- Collection API contains group of classes and interfaces that makes it easier to handle group of objects.
- Collections are providing flexibility to store, retrieve, and manipulate data.

### The key interfaces of collection framework:-

1. **Java.util.Collection**
2. **Java.util.List**
3. **Java.util.Set**
4. **Java.util.SortedSet**
5. **Java.util.NavigableSet**
6. **Java.util.Queue**
7. **Java.util.Map**
8. **Java.util.SortedMap**
9. **Java.util.NavigableMap**
10. **Map.Entry**
11. **Java.util.Enumeration**
12. **Java.util.Iterator**
13. **Java.util.ListIterator**
14. **Java.lang.Comparable**
15. **Java.util.Comparator**

- ❖ All collection framework classes and interfaces are present in **java.util** package.
- ❖ The root interface of Collection framework is **Collection**.
- ❖ Collection interface contains 15 methods so all collection implementation classes are able to use these methods because collections is a root interface.

**Collection vs Collections:-**

*Collection is interface it is used to represent group of objects as a single entity.*  
*Collections is utility class it contains methods to perform operations.*

**Arrays vs Collections:-**

*Both Arrays and Collections are used to represent group of objects as a single entity but the differences are as shown below.*

**Limitations of Arrays**

- 1) *Arrays are used to represent group of objects as a single entity.*
- 2) *Arrays are used to store homogeneous data(similar data).*
- 3) *Arrays are capable to store primitive & Object type data*
- 4) *Arrays are fixed in size, it means once we created array it is not possible to increase & decrease the size based on our requirement.*
- 5) *With respect to memory arrays are not recommended to use.*
- 6) *If you know size in advance arrays are recommended to use because it provide good performance.*
- 7) *Arrays does not contains underlying Data structure hence it is not supporting predefined methods.*
- 8) *While working with arrays operations(add,remove,update...) are become difficult because it is not supporting methods.*

**Advantages of Collections**

- 1) *Collections are used to represent group of objects as a single entity.*
- 2) *Collections are used to store both heterogeneous data(different type)& homogeneous data.*
- 3) *Collections are capable to store only object data.*
- 4) *Collections are growable in nature, it means based on our requirement it is possible to increase & decrease the size.*
- 5) *With respect to memory collections are recommended to use.*
- 6) *In performance point of view collections will give low performance compare to arrays.*
- 7) *Collection classes contains underlying data structure hence it supports predefined methods.*
- 8) *Here operations are become easy because collections supports predefined methods.*

**Characteristics of Collection framework classes:-**

The collections framework contains group of classes but every class is used to represent group of objects as a single entity but characteristics are different.

**1) The collection framework classes introduced Versions**

Different classes are introduced in different versions.

**2) Heterogeneous data allowed or not allowed.**

All most all collection framework classes allowed heterogeneous data except two classes

- i. TreeSet
- ii. TreeMap

**3) Null insertion is possible or not possible.**

Some classes are allowed null insertion but some classes are not allowed.

**4) Duplicate objects are allowed or not allowed.**

Inserting same object more than one time is called duplication. Some classes are allowed duplicates but some classes are not allowed duplicates.

add(e1)

add(e1)

**5) Insertion order is preserved or not preserved.**

In which order we are inserting element same order output is printed then say insertion order is preserved otherwise not.

Input --->e1 e2 e3 output --->e1 e2 e3 insertion order is preserved

Input --->e1 e2 e3 output --->e2 e1 e3 insertion order is not-preserved

**6) Collection classes' methods are synchronized or non-synchronized.**

If the methods are synchronized only one thread is allow to access, these methods are thread safe but performance is reduced.

If the methods are non-synchronized multiple threads are able to access, these methods are not thread safe but performance is increased.

**7) Collection classes underlying data structures.**

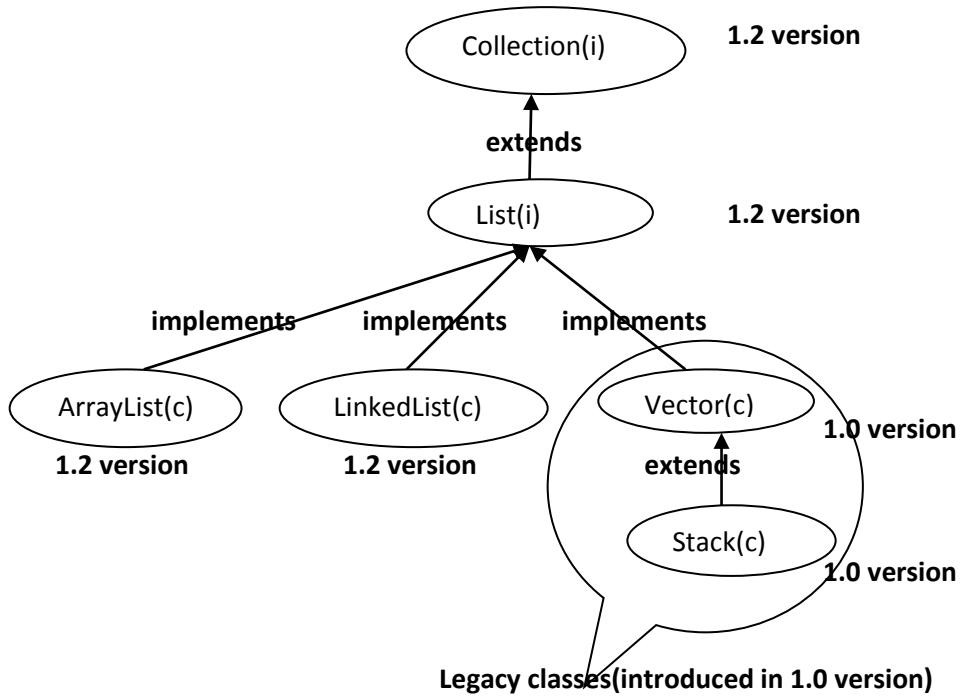
Arrays do not contain underlying data structure hence it is not supporting predefined methods.

Every collection class contains underlying data structure hence it supports predefined methods.

Based on underlying data structure the element will be stored.

**8) Collection classes supported cursors.**

The collection classes are used to represent group of objects as a single entity & To retrieve the objects from collection class we are using cursors.

List interface:-

I = Interface    c=class

Implementation classes of List interface :-

- 1) ArrayList
- 2) LinkedList
- 3) Vector
- 4) Stack

Legacy classes:-

The java classes which are introduced in 1.0 version are called legacy classes and **java.util** package contains 5 legacy classes.

- 1) Dictionary
- 2) HashTable
- 3) Properties
- 4) Stack
- 5) Vector

Legacy interfaces:-

The java interfaces which are introduced in 1.0 version are called legacy interfaces and **java.util** package contains only one interface is **Enumeration**.

List interface common properties:-

- 1) All list class allows heterogeneous data.
- 2) All List interface implementation classes allows null insertion.
- 3) All classes allows duplicate objects.
- 4) All classes preserved insertion order.

**Java.util.ArrayList**:- To check parent class and interface use below command.

D:\ratan>javap java.util.ArrayList

```
public class java.util.ArrayList<E>
    extends java.util.AbstractList<E>
    implements java.util.List<E>,
               java.util.RandomAccess,
               java.lang.Cloneable,
               java.io.Serializable
```



### **ArrayList Characteristics:-**

- 1) ArrayList Introduced in 1.2 version.
- 2) ArrayList stores Heterogeneous objects(different types).
- 3) In ArrayList it is possible to insert Null objects.
- 4) Duplicate objects are allowed.
- 5) ArrayList preserved Insertion order it means whatever the order we inserted the data in the same way output will be printed.
- 6) ArrayList methods are non-synchronized methods.
- 7) The underlying data structure is growable array.
- 8) By using cursor we are able to retrieve the data from ArrayList : **Iterator**, **ListIterator**

### **Constructors to create ArrayList:-**

#### **Constructor-1:-**

**ArrayList al = new ArrayList();**

The default capacity of the ArrayList is 10 once it reaches its maximum capacity then size is automatically increased by   **New capacity = (old capacity\*3)/2+1**

#### **Constructor-2:-      ArrayList al = new ArrayList ( int initial-capacity);**

It is possible to create ArrayList with initial capacity

#### **Constructor-3:-**

**ArrayList al = new ArrayList(Collection c);**

Adding one collection data into another collection(Vector data intoArrayList) use following constructor.

### **Example :-**

#### **Collections vs Autoboxing**

Up to 1.4 version we must create wrapper class object then add that object into ArrayList.

```
import java.util.ArrayList;
class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        Integer i = new Integer(10);           //creation of Integer Object
        Character ch = new Character('c');    //creation of Character Object
        Double d = new Double(10.5);          //creation of Double Object
        //adding wrapper objects into ArrayList
        al.add(i);
        al.add(ch);
        al.add(d);
        System.out.println(al);
    }
}
```

From 1.5 version onwards add the primitive data into ArrayList that data is automatically converted into wrapper object format is called Autoboxing.

#### Code before compilation:-

```
import java.util.ArrayList;
class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(10); //AutoBoxing
        al.add('a'); //AutoBoxing
        al.add(10.5); //AutoBoxing
        System.out.println(al);
    }
}
```

#### Code after compilation:-

```
import java.io.PrintStream;
import java.util.ArrayList;
class Test
{
    Test()
    {
    }
    public static void main(String args[])
    {
        ArrayList arraylist = new ArrayList();
        arraylist.add(Integer.valueOf(10));
        arraylist.add(Character.valueOf('a'));
        arraylist.add(Double.valueOf(10.5));
        System.out.println(arraylist);
    }
}
```

#### Example-2:-ArrayList vs toString()

##### Emp.java:-

```
class Emp
{
    int eid;
    String ename;
    Emp(int eid,String ename)
    {
        this.eid=eid;
        this.ename=ename;
    }
}
```

##### Student.java

```
class Student
{
    int sid;
    String sname;
    Student(int sid,String sname)
    {
        this.sid=sid;
        this.sname = sname;
    }
}
```

##### Case 1:-

In java when we print reference variable internally it calls `toString()` method on that object.

```
import java.util.ArrayList;
class Test
{
    public static void main(String[] args)
    {
        Emp e1 = new Emp(111,"ratan");
        Student s1 = new Student(222,"xxx");
        ArrayList al = new ArrayList();
        al.add(10); //toString()
        al.add('a'); //toString()
        al.add(e1); //toString()
        al.add(s1); //toString()
        System.out.println(al); //[[10, a, Emp@d70d7a, Student@b5f53a]
        System.out.println(al.toString()); //[[10, a, Emp@d70d7a, Student@b5f53a]
    }
}
```

**Case2:-**

```

import java.util.ArrayList;
class Test
{
    public static void main(String[] args)
    {
        Emp e1 = new Emp(111,"ratan");
        Student s1 = new Student(222,"xxx");
        ArrayList al = new ArrayList();
        al.add(10);
        al.add('a');
        al.add(e1);
        al.add(s1);
        System.out.println(al.toString()); // [10, a, Emp@d70d7a, Student@b5f53a]
        for (Object o : al)
        {
            if (o instanceof Integer)
                System.out.println(o.toString());
            if (o instanceof Character)
                System.out.println(o.toString());
            if (o instanceof Emp){
                Emp e = (Emp)o;
                System.out.println(e.eid+"---"+e.ename);
            }
            if (o instanceof Student){
                Student s = (Student)o;
                System.out.println(s.sid+"---"+s.sname);
            }
        }
    }
}

```

**Example:- Basic operations of ArrayList**

- add()** to add the objects into ArrayList & by default it add the data at last but it is possible to insert the data at specified index.
- remove()** it removes Objects from ArrayList based on Object & index.  
(for the remove(10) method if we are passing integer value that is always treated as index )
- size()** to find the size of ArrayList.
- isEmpty()** to check the objects are available or not.
- Clear()** to remove all objects from ArrayList.

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList al =new ArrayList();
        al.add(10);
        al.add("ratan");
        al.add("anu");
        al.add('a');
        al.add(10);
        al.add(null);
    }
}

```

```

System.out.println("ArrayList data="+al);
System.out.println("ArrayList size-->"+al.size());
al.add(1,"A1");           //add the object at first index
System.out.println("after adding objects ArrayList size-->"+al.size());
System.out.println("ArrayList Data="+al);
al.remove(1);             //remove the object index base
al.remove("A");           //remove the object on object base
System.out.println("after removeing elemetns arrayList size "+al.size());
System.out.println("ArrayList data="+al);
System.out.println(al.isEmpty());
al.clear();
System.out.println(al.isEmpty());
}

}

E:\>java Test
ArrayList data=[10, ratan, anu, a, 10, null]
ArrayList size-->6
after adding objects ArrayList size-->7
ArrayList Data=[10, A1, ratan, anu, a, 10, null]
after removeing elemetns arrayList size 6
ArrayList data=[10, ratan, anu, a, 10, null]
false
true

```

**observation:-**

*in above example when we remove the data by passing numeric value that is by default treated as a index value.*

```

ArrayList al = new ArrayList();
al.add(10);
al.add("ratan");
al.add('a');
System.out.println(al);
al.remove(10); // 10 is taken as index value

```

*whenever we are executing above code then JVM treats that 10 is index value but 10<sup>th</sup> position value is not available hence it is generating exception **java.lang.IndexOutOfBoundsException: Index: 10, Size: 3***

*in above case to remove 10 Integer object then use below code.*

```

ArrayList al = new ArrayList();
Integer i = new Integer(10);
al.add(i);
al.remove(i);
System.out.println(al);

```

**All collection classes are having 2-formats:-**

- 1) Normal version (no type safety).
- 2) Generic version. (provide type safety )

The main purpose of the generics is to provide the type safety to avoid type casting problems.

**Arrays [Type safety]:-**

Arrays are always type safe it means we can give guarantee for the type of element present in arrays.

For example if we want to store String objects create `String[]`. By mistake if we are trying to add any other data compiler will generate compilation error.

**Example:-**

```
String[] str= new String[50];
str[0]="ratan";
str[1]="anu";
str[2]=new Integer(10);
```

E:\>javac Test.java

```
incompatible types
    str[2]=new Integer(10);
required: String
found: Integer
```

Based on above error we can give guarantee `String[]` is able to store only String type of objects. Hence with respect to the type arrays are recommended to use because it is type safe.

**[Collection]Not type safe:-**

Collections are not type safe it means we can't give guarantee for the type of elements present in collection.

If programming requirement is to hold the String type of the objects we are choosing `ArrayList` , by mistake if we are adding any type of data then compiler is unable to generate compilation error but runtime program is failed.

**Example :-**

```
ArrayList al = new ArrayList();
al.add("ratan");
al.add("anu");
al.add(new Integer(10)); //allowed
String s1 = (String)al.get(0);
String s2 = (String)al.get(1);
String s3 = (String)al.get(2); //java.lang.ClassCastException
```

Based on above exception we can decide collection is not type safe.

**Example :-**

```
ArrayList al = new ArrayList();
al.add("ratan");
String str = l.get(0); // java.lang.ClassCastException
String str =(String l.get(0)); //type casting is mandatory
```

In above example type-casting is mandatory it is bigger problem in collection.

To overcome above problems use generics,

- 1) To provide type safety.
- 2) To overcome type casting problems.

*in java it is recommended to use generic version of collections to provide the type safety.*

**Syntax:-**

```
ArrayList<type-name> al = new ArrayList<type-name>();
```

*The ArrayList is able to store only String data if we are trying to add any other data compiler will generate error message.*

```
Example :- ArrayList<String> al = new ArrayList<String>();
al.add("ratan");
al.add("anu");
al.add(new Integer(10)); //compilation error
```

*if we are using generics we will get type safety. At the time retrieval not required to perform type casting.*

```
ArrayList<String> al = new ArrayList<String>();
al.add("ratan");
String s1 = al.get(0);
```

**Normal version of ArrayList(no type safety)**

- 1) Normal version is able to hold any type of data(heterogeneous data) hence it is not a type safe.

```
ArrayList al = new ArrayList();
al.add(10);
al.add('a');
System.out.println(al);
```

- 2) At the time of retrieval Always check the type of the object by using **instanceof** operator.

- 3) In normal it is holding different types of data hence while retrieving data must perform **type casting**.

- 4) If we are using normal version while compilation compiler generate warning message like **unchecked or unsafe operations**.

**Example:- normal version of ArrayList**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(10);
        al.add('a');
        al.add(10.4);
        System.out.println(al);
    }
}
```

**Generic version of ArrayList(type safety)**

- 1) Generic version is able to hold specified type of data hence it is a type safe.

```
ArrayList<type-name> al = new ArrayList<type-name>();
ArrayList<Integer> al = new ArrayList<Integer>();
al.add(10);
al.add(20);
al.add("ratan");//compilation error
System.out.println(al);
```

- 2) Type checking is not required because it contains only one type of data.

- 3) It is holding specific data hence at the time of retrieval type casting is not required.

- 4) If we are using generic version compiler won't generate warning messages.

**Example :- generic version of ArrayList.**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<Integer> al = new ArrayList<Integer>();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        System.out.println(al);
    }
}
```

**Example :- retrieving data from generic version of ArrayList.**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<Emp> al = new ArrayList<Emp>();
        al.add(new Emp(111, "ratan"));
        al.add(new Emp(222, "anu"));
        al.add(new Emp(333, "Sravya"));
        for (Emp e : al)
        {
            System.out.println(e.eid+"---"+e.ename);
        }
    }
}
```

**Example :-**

|                                  |                                                                     |
|----------------------------------|---------------------------------------------------------------------|
| <b>add(E);</b>                   | ----> <b>to add the Object.</b>                                     |
| <b>remove(java.lang.Object);</b> | ----> <b>to remove the object.</b>                                  |
| <b>addAll();</b>                 | -----> <b>to add one collection object into another collection.</b> |
| <b>contains()</b>                | -----> <b>to check object is available or not.</b>                  |
| <b>containsAll()</b>             | ----> <b>to check entire collection data is available or not.</b>   |

**Example:-**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Emp e1 = new Emp(111, "ratan");
        Emp e2 = new Emp(222, "Sravya");
        Emp e3 = new Emp(333, "aruna");
        Emp e4 = new Emp(444, "anu");

        ArrayList<Emp> a1 = new ArrayList<Emp>();
        a1.add(e1);
        a1.add(e2);

        ArrayList<Emp> a2 = new ArrayList<Emp>();
        a2.addAll(a1);
        a2.add(e3);
        a2.add(e4);

        System.out.println(a2.contains(e1));
        System.out.println(a2.containsAll(a1));
        a2.remove(e1);
        System.out.println(a2.contains(e1));
        System.out.println(a2.containsAll(a1));
        //printing the data
        for (Emp e:a2)
        {
            System.out.println(e.eid+"---"+e.ename);
        }
    }
}
```

```

Example :-removeAll(Obj):-      a2.removeAll(a1); // it removes all a1 data.
retainAll(Obj):- a2.retainAll(a1); // it removes all a2 data except a1

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<Emp> a1 = new ArrayList<Emp>();
        a1.add(new Emp(111,"ratan"));
        a1.add(new Emp(222,"Sravya"));

        ArrayList<Emp> a2 = new ArrayList<Emp>();
        a2.addAll(a1);
        a2.add(new Emp(333,"aruna"));
        a2.add(new Emp(444,"anu"));
        a2.removeAll(a1);
        a2.retainAll(a1);
        for (Emp e:a2)
        {
            System.out.println(e.eid+"---"+e.ename);
        }
    }
}

```

#### **Creation of sub ArrayList & swapping data :-**

Create sub ArrayList by using **subList(int,int)** method of ArrayList.

**public java.util.List<E> subList(int, int);**

to swap the data from one index position to another index position then use **swap()** method of Collections class.

```

public static void swap(java.util.List<?>, int, int);
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> a1 = new ArrayList<String>();
        a1.add("ratan");
        a1.add("anu");
        a1.add("Sravya");
        a1.add("yadhu");

        ArrayList<String> a2 = new ArrayList<String>(a1.subList(1,3));
        System.out.println(a2);      // [anu,Sravya]
        ArrayList<String> a3 = new ArrayList<String>(a1.subList(1,a1.size()));
        System.out.println(a3);      // [anu,Sravya,yadhu]
//java.lang.IndexOutOfBoundsException: toIndex = 7
//ArrayList<String> a4 = new ArrayList<String>(a1.subList(1,7));
        System.out.println("before swapping="+a1);//[ratan, anu, Sravya, yadhu]
        Collections.swap(a1,1,3);
        System.out.println("after swapping="+a1);// [ratan, yadhu, Sravya, anu]
    }
}

```

**ArrayList Capacity:-**

```

import java.util.*;
import java.lang.reflect.Field;
class Test
{
    public static void main(String[] args) throws Exception
    {
        ArrayList<Integer> al = new ArrayList<Integer>(5);
        for (int i=0;i<10 ;i++)
        {
            al.add(i);
            System.out.println("size="+al.size()+" capacity="+getcapacity(al));
        }
    }
    static int getcapacity(ArrayList l) throws Exception
    {
        Field f = ArrayList.class.getDeclaredField("elementData");
        f.setAccessible(true);
        return ((Object[])f.get(l)).length;
    }
}
D:\>java Test
size=1 capacity=5
size=2 capacity=5
size=3 capacity=5
size=4 capacity=5
size=5 capacity=5
size=6 capacity=8
size=7 capacity=8
size=8 capacity=8
size=9 capacity=13
size=10 capacity=13

```

**Different ways to initialize values to ArrayList:-****Case 1: initializing ArrayList by using asList()**

```

import java.util.*;
class ArrayListDemo
{
    public static void main(String[] args)
    {
        ArrayList<String> al = new ArrayList<String>(
            Arrays.asList("ratan", "Sravya", "anu"));
        System.out.println(al);
    }
}

```

**Case 2:- adding objects into ArrayList by using anonymous inner classes.**

```

import java.util.ArrayList;
class ArrayListDemo
{
    public static void main(String[] args)
    {
        ArrayList<String> al = new ArrayList<String>()

```

```

    {
        {add("anu");
         add("ratan");
        }
    };//semicolon is mandatory
    System.out.println(al);
}
}

```

**Case 3:- normal approach to initialize the data**

```

import java.util.ArrayList;
class ArrayListDemo
{
    public static void main(String[] args)
    {
        ArrayList<String> al = new ArrayList<String>();
        al.add("anu");
        al.add("Sravya");
        System.out.println(al);
    }
}

```

**Case 4:-**

```

ArrayList<Type> obj = new ArrayList<Type>(Collections.nCopies(count, object));
import java.util.*;
class ArrayListDemo
{
    public static void main(String[] args)
    {
        Emp e1 = new Emp(111,"ratan");
        ArrayList<Emp> al = new ArrayList<Emp>(Collections.nCopies(5,e1));
        for (Emp e:al)
        {
            System.out.println(e.ename+"---"+e.eid);
        }
    }
}

```

**Case 5:-adding Objects into ArrayList by using addAll() method of Collections class.**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> al = new ArrayList<String>();
        String[] strArray={"ratan","anu","Sravya"};
        Collections.addAll(al,strArray);
        System.out.println(al);
    }
}

```

Q. **How to get synchronized version of ArrayList?**

Ans:- by default ArrayList methods are synchronized but it is possible to get synchronized version of ArrayList by using following method.

To get synchronized version of List interface use following Collections class static method  
**public static List synchronizedList(List l)**

To get synchronized version of Set interface use following Collections class static method  
**public static Set synchronizedSet(Set s)**

To get synchronized version of Map interface use following Collections class static method  
**public static Map synchronized Map(Map m)**

to get synchronized version of TreeSet use following Collections class static method  
**Collections.synchronizedSortedSet(SortedSet<T> s)**

to get synchronized version of TreeMap use following Collections class static method  
**Collections.synchronizedSortedMap(SortedMap<K,V> m)**

**Example:-**

```
ArrayList al = new ArrayList();           //non- synchronized version of ArrayList
List l = Collections.synchronizedList(al); // synchronized version of ArrayList
```

```
HasSet h = new HashSet();                 //non- synchronized version of HashSet
Set h1 = Collections.synchronizedSet(h); // synchronized version of HashSet
```

```
HashMap h = new HashMap();               //non- synchronized version of HashMap
Map m = Collections.synchronizedMap(h); // synchronized version of HashMap
```

```
TreeSet t = new TreeSet();                //non- synchronized version of TreeSet
SortedSet s = Collections.synchronizedSortedSet(t); // synchronized version of TreeSet
```

```
TreeMap t = new TreeMap();                //non- synchronized version of TreeMap
SortedMap s = Collections.synchronizedSortedMap(t); // synchronized version of TreeMap
```

### Conversion of Arrays to ArrayList & ArrayList to Arrays:

#### Example-1:

##### Conversion of String array to ArrayList (by using asList() method):-

```
import java.util.*;
class ArrayListDemo
{
    public static void main(String[] args)
    {
        String[] str={"ratan","Sravya","aruna"};
        ArrayList<String> al = new ArrayList<String>(Arrays.asList(str));
        al.add("newperson-1");
        al.add("newperson-2");
        //printing data by using enhanced for loop
        for (String s: al)
        {
            System.out.println(s);
        }
    }
}
```

#### Example-2:-

##### Conversion of ArrayList to String array by using toArray( T )

```
public abstract <T extends java/lang/Object> T[] toArray(T[]);
import java.util.*;
class ArrayListDemo
{
    public static void main(String[] args)
    {
        //interface ref-var & implementaiton class Object
        List<String> al = new ArrayList<String>();
        al.add("anu");
        al.add("Sravya");
        al.add("ratan");
        al.add("natraj");
        String[] a = new String[al.size()];
        al.toArray(a);
        //for-each loop to print the data
        for (String s:a)
        {
            System.out.println(s);
        }
    }
}
```

**Example-3:-****Case-1 :- conversion of ArrayList to Array**

```

        public abstract java.lang.Object[] toArray();
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(10);
        al.add('c');
        al.add("ratan");
//conversion of ArrayList to array
        Object[] o = al.toArray();
        for (Object oo :o)
        {
            System.out.println(oo);
        }
    }
}

```

**Case-2 :-**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(new Emp(111,"ratan"));
        al.add(new Student(1,"xxx"));
        al.add("ratan");
//conversion of ArrayList to array
        Object[] o = al.toArray();
        for (Object oo :o)
        {
            if (oo instanceof Emp)
            {
                Emp e = (Emp)oo;
                System.out.println(e.eid+"---"+e.ename);
            }
            if (oo instanceof Student)
            {
                Student s = (Student)oo;
                System.out.println(s.sid+"---"+s.sname);
            }
            if (oo instanceof String)
            {
                System.out.println(oo.toString());
            }
        }
    }
}

```

**Cursors in Collections:****Property****1. Purpose****2. Legacy or not****3. Applicable for which type of classes****4. Universal cursor or not****5. How to get the object****6. How many methods****7. Operations****8. Cursor moment****9. Class or interface****10. Versions supports****Enumeration**

- 1) Used to retrieve the data from collection classes.
- 2) Introduced in 1.0 version it is legacy
- 3) It is used to retrieve the data from only legacy classes like vector, Stack...etc
- 4) Not a universal cursor because it is applicable for only legacy classes.
- 5) Get the Enumeration Object by using elements() method.

```
Vector v =new Vector();
```

```
v.add(10);
```

```
v.add(20);
```

```
Enumeration e = v.elements();
```

- 6) It contains two methods  
hasMoreElements(): to check objects.  
nextElement() : to retrieve the objects.

7) Only read operations.

8) Only forward direction.

9) Interface

10) It supports both normal and generic version.

**Iterator**

- 1) Used to retrieve the objects from collection classes.
- 2) Introduced in 1.2 version it is not a legacy
- 3) It is used to retrieve the data from all collection classes.
- 4) It is a universal cursor because it is applicable for all collection classes.
- 5) Get the iterator Object by using iterator() method.

```
Vector v =new Vector();
```

```
v.add(10);  
v.add(20);
```

```
Enumeration e = v.iterator();
```

6) It contains two methods  
**hasNext()**: to check the objects available or not.  
**Next()** : to retrieve the objects.

- 7) read & remove operations are possible.
- 8) Only forward direction.
- 9) Interface
- 10) It supports both normal and generic version.

**ListIterator**

- 1) Used to retrieve the data from collection classes.
- 2) Introduced in 1.2 version it is not a legacy
- 3) It is used to retrieve the data from only List type of classes like ArrayList,LinkedList,Vector,Stack.
- 4) Not a universal cursor because it is applicable for only List interface classes.
- 5) Get the ListIterator Object by using listIterator() method.

```
Vector v =new Vector();
```

```
v.add(10);  
v.add(20);
```

```
Enumeration e = v.listIterator();
```

- 6) It contains 9 methods
- 7) Read, remove, add, and replace operations.
- 8) Bidirectional cursor direction.
- 9) Interface
- 10) It supports both normal and generic version.

**ListIterator methods:-**

```
public abstract boolean hasNext();  
public abstract E next();  
public abstract boolean hasPrevious();  
public abstract E previous();  
public abstract int nextIndex();  
public abstract int previousIndex();  
public abstract void remove();  
public abstract void set(E); //replacement  
public abstract void add(E);
```

**Retrieving objects of collections classes:-**

We are able to retrieve the objects from collection classes in 3-ways

- 1) By using for-each loop.
- 2) By using get() method.
- 3) By using cursors.

**Example application:-**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> al =new ArrayList<String>();
        al.add("ratan");
        al.add("anu");
        al.add("sravya");
//1st approach to print Collection data
        for (String a : al )
        {
            System.out.println(a);
        }

//2nd approach to print Collection data
        int size = al.size();
        for (int i=0;i<size;i++)
        {
            System.out.println(al.get(i));
        }

//3rd approach to print Collection data
//normal version of Iterator(type casting required at the time of retrieving)
        Iterator itr1 = al.iterator();
        while (itr1.hasNext())
        {
            String str =(String)itr1.next();
            System.out.println(str);
        }

//generic version of Iterator(type casting not required at the time of retrieving)
        Iterator<String> itr2 = al.iterator();
        while (itr2.hasNext())
        {
            String str =itr2.next();
            System.out.println(str);
        }
    }
}
```

**Example:-**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> al =new ArrayList<String>();
        al.add("ratan");
        al.add("anu");
        al.add("sravya");
        ListIterator<String> lstr = al.listIterator();
        lstr.add("suneel");
        while(lstr.hasNext())
        {
            if ((lstr.next()).equals("anu"))
            {
                lstr.set("Anushka");
            }
        }
        lstr.add("aaa");
        for (String str:al)
        {
            System.out.println(str);
        }
    }
}

```

E:\>java Test

suneel

ratan

Anushka

sravya

aaa

**if we want remove the data:-**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> al =new ArrayList<String>();
        al.add("ratan");
        al.add("anu");
        al.add("sravya");
        ListIterator<String> lstr = al.listIterator();
        while(lstr.hasNext())
        {
            if ((lstr.next()).equals("ratan"))
            {
                lstr.remove();
            }
        }
        for (String str:al)
        {
            System.out.println(str);
        }
    }
}

```

E:\>java Test

anu

sravya

**Example:-printing data in forward and backward directions.**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> al =new ArrayList<String>();
        al.add("ratan");
        al.add("anu");
        al.add("sravya");
        ListIterator<String> lstr = al.listIterator();
        System.out.println("printing data forward direction");
        while(lstr.hasNext())
        {
            System.out.println(lstr.next());
        }
        System.out.println("printing data backward direction");
        while(lstr.hasPrevious())
        {
            System.out.println(lstr.previous());
        }
    }
}
E:\>java Test
printing data forward direction
ratan
anu
sravya
printing data backward direction
sravya
anu
ratan
```

**Sorting data by using sort() method of Collections class:-**

we are able to sort ArrayList data by using sort() method of Collections class and by default it perform ascending order .

**public static <T extends java/lang/Comparable<? super T>> void sort(java.util.List<T>);**  
if we want to person ascending order your class must implements Comparable interface of java.lang package.

If we want to perform descending order use **Collections.reverseOrder()** method along with **Collection.sort()** method.

**Collections.sort(list , Collections.reverseOrder());**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> al = new ArrayList<String>();
        al.add("ratan");
        al.add("anu");
        al.add("Sravya");
        //printing ArrayList data
        System.out.println("ArrayList data before sorting");
        for (String str :al)
        {
            System.out.println(str);
        }
        //sorting ArrayList in ascending order
        Collections.sort(al);
        System.out.println("ArrayList data after sorting ascending order");
        for (String str1 :al)
        {
            System.out.println(str1);
        }
        //sorting ArrayList in decending order
        Collections.sort(al,Collections.reverseOrder());
        System.out.println("ArrayList data after sorting decending order");
        for (String str2 :al)
        {
            System.out.println(str2);
        }
    }
}
```

**Example:-**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> al = new ArrayList<String>();
        al.add("ratan");
        al.add("anu");
        al.add("Sravya");
        Collections.sort(al);
        System.out.println("ArrayList data after sorting");
        for (String str1 :al)
        {
            System.out.println(str1);
        }
    }
}

```

- ✓ in above example to perform the sorting of data by using natural sorting order then your objects must be homogeneous and must implements comparable interface.
- ✓ The default natural sorting order internally uses compareTo() method to perform sorting and it compare to objects and it return int value as a return value.

|                            |    |     |                    |
|----------------------------|----|-----|--------------------|
| "ratan".compareTo("anu")   | => | +ve | =>change the order |
| "ratan".compareTo("ratan") | => | 0   | =>no change        |
| "anu".compareTo("ratan")   | => | -ve | =>no change        |

**Example:-**

**The sorting object(Emp) Not implementing Comparable interface hence it does not perform sorting.**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(new Emp(111, "ratan"));
        Collections.sort(al);
    }
}

```

When we execute the above example JVM will generate Exception,

**"java.lang.ClassCastException: Emp cannot be cast to java.lang.Comparable"**

**Example :-**

**If the Class contains Heterogeneous data sorting is not possible.**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add("ratan");
        al.add(10);
        Collections.sort(al); //java.lang.ClassCastException
        System.out.println(al);
    }
}

```

To overcome above two cases exception use Comparable or Comparator interfaces to perform sorting.

**Comparable vs Comparator :-**

- ✓ If we want to perform default natural sorting order then your objects must be homogeneous & comparable.
  - ✓ Comparable objects are nothing but the objects which are implements comparable interface.
  - ✓ All wrapper classes & String objects are implementing Comparable interface hence it is possible to perform sorting.
  
  - ❖ If we want to sort user defined class like Emp based on eid or ename with default natural sorting order then your class must implements Comparable interface.
  - ❖ Comparable interface present in java.lang package it contains only one method compareTo(obj) then must override that method to write the sorting logics.
- public abstract int compareTo(T);***
- ❖ If your class is implementing Comparable interface then that objects are sorted automatically by using **Collections.sort()**. And the objects are sorted by using compareTo() method of that class.

**Normal version of comparable:-****Emp.java:-**

```
class Emp implements Comparable
{
    int eid;
    String ename;
    Emp(int eid, String ename)
    {
        this.eid=eid;
        this.ename=ename;
    }
    public int compareTo(Object o)
    {
        Emp e = (Emp)o;
        if (eid == e.eid)
        {
            return 0;
        }
        else if (eid > e.eid)
        {
            return 1;
        }
        else
        {
            return -1;
        }
    }
}
```

**Test.java:-**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<Emp> al = new ArrayList<Emp>();
        al.add(new Emp(333, "ratan"));
        al.add(new Emp(222, "anu"));
```

```

        al.add(new Emp(111,"Sravya"));
        Collections.sort(al);
        Iterator itr = al.iterator();
        while (itr.hasNext())
        {
            Emp e = (Emp)itr.next();
            System.out.println(e.eid+"---"+e.ename);
        }
    }
}

```

**Generic version of Comparable:-**

```

class Emp implements Comparable<Emp>
{
    int eid;
    String ename;
    Emp(int eid,String ename)
    {
        this.eid=eid;
        this.ename=ename;
    }
    public int compareTo(Emp e)
    {
        return ename.compareTo(e.ename);
    }
}

```

**Java.utilComparator :-**

- ✓ For the default sorting order use comparable but for customized sorting order we can use Comparator.
- ✓ The class whose objects are stored do not implements this interface some third party class can also implements this interface.
- ✓ Comparable present in **java.lang** package but Comparator present in **java.util** package.
- ✓ Comparator interface contains two methods,

```

public interface java.util.Comparator<T> {
    public abstract int compare(T, T);
    public abstract boolean equals(java.lang.Object);
}

```

**Normal version of Comparator:-****Emp.java:-**

```

class Emp
{
    int eid;
    String ename;
    Emp(int eid,String ename)
    {
        this.eid=eid;
        this.ename=ename;
    }
}

```

**EidComp.java:-**

```
import java.util.Comparator;
class EidComp implements Comparator
{
    public int compare(Object o1, Object o2)
    {
        Emp e1 = (Emp)o1;
        Emp e2 = (Emp)o2;
        if (e1.eid==e2.eid)
        {
            return 0;
        }
        else if (e1.eid>e2.eid)
        {
            return 1;
        }
        else
        {
            return -1;
        }
    }
}
```

**EnameComp.java:-**

```
import java.util.Comparator;
class EnameComp implements Comparator
{
    public int compare(Object o1, Object o2)
    {
        Emp e1 = (Emp)o1;
        Emp e2 = (Emp)o2;
        return (e1.ename).compareTo(e2.ename);
    }
}
```

**Test.java:-**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<Emp> al = new ArrayList<Emp>();
        al.add(new Emp(333, "ratan"));
        al.add(new Emp(222, "anu"));
        al.add(new Emp(111, "Sravya"));
        al.add(new Emp(444, "xxx"));
    }
}
```

**System.out.println("sorting by eid");**

```
Collections.sort(al, new EidComp());
Iterator<Emp> itr = al.iterator();
while (itr.hasNext())
{
    Emp e = itr.next();
    System.out.println(e.eid+"---"+e.ename);
}
```

**System.out.println("sorting by ename");**

```
Collections.sort(al, new EnameComp());
Iterator<Emp> itr1 = al.iterator();
```

```

        while (itr1.hasNext())
    {
        Emp e = itr1.next();
        System.out.println(e.eid+"---"+e.ename);
    }
}

D:\vikram>java Test
sorting by eid
111---Sravya
222---anu
333---ratan
444---xxx
sorting by ename
222---anu
111---Sravya
333---ratan
444---xxx

```

**The above example code:-(with generic version)**

**EnameComp.java:-**

```

import java.util.Comparator;
class EnameComp implements Comparator<Emp>
{
    public int compare(Emp e1,Emp e2)
    {
        return (e1.ename).compareTo(e2.ename);
    }
}

```

**EidComp.java:-**

```

import java.util.Comparator;
class EidComp implements Comparator<Emp>
{
    public int compare(Emp e1,Emp e2)
    {
        ****
        ****
    }
}

```

**Java.lang.Comparable vs java.util.Comparator:-**

| <u>Property</u>                             | <u>Comparable</u>                                                                                                                                                                                                                                                                           | <u>Comparator</u>                                                                                                                                                                                                                                                   |
|---------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. <u>Sorting logics</u>                    | 1) Sorting logics must be in the class whose class objects are sorting.                                                                                                                                                                                                                     | I. Sorting logics in separate class hence we are able to sort the data by using dif attributes.                                                                                                                                                                     |
| 2. <u>Sorting method</u>                    | 2) <i>Int compareTo(Object o1)</i><br>This method compares this object with o1 object and returns a integer. Its value has following meaning<br><b>positive</b> – this object is greater than o1<br><b>zero</b> – this object equals to o1<br><b>negative</b> – this object is less than o1 | II. <i>int compare(Object o1, Object o2)</i><br>This method compares o1 and o2 objects. and returns a integer. Its value has following meaning.<br><b>positive</b> – o1 is greater than o2<br><b>zero</b> – o1 equals to o2<br><b>negative</b> – o1 is less than o1 |
| 3. <u>Method calling to perform sorting</u> | 3) <b>Collections.sort(List)</b><br>Here objects will be sorted on the basis of CompareTo method.                                                                                                                                                                                           | III. <b>Collections.sort(List, Comparator)</b><br>Here objects will be sorted on the basis of Compare method in Comparator                                                                                                                                          |
| 4. <u>package</u>                           | 4) Java.lang                                                                                                                                                                                                                                                                                | IV. <b>Java.util</b>                                                                                                                                                                                                                                                |
| 5. <u>which type of sorting</u>             | 5) Default natural sorting order                                                                                                                                                                                                                                                            | V. <b>For customized sorting order.</b>                                                                                                                                                                                                                             |

**java.util.LinkedList:-**

```
public class java.util.LinkedList extends java.util.AbstractSequentialList
    implements java.util.List<E>,
               java.util.Deque<E>,
               java.lang.Cloneable,
               java.io.Serializable
```

- 1) Introduced in 1.2 version.
- 2) Heterogeneous objects are allowed.
- 3) Null insertion is possible.
- 4) Insertion order is preserved.
- 5) LinkedList methods are non-synchronized.
- 6) Duplicate objects are allowed.
- 7) The under laying data structure is double linkedlist.
- 8) cursors :- Iterator, ListIterator

**constructors:-**

**LinkedList();** it builds a empty LinkedList.  
**LinkedList(java/util/Collection<? extends E>);**  
 it builds a LinkedList that initialized with the collection data.

**Example:- LinkedList basic operations.**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        LinkedList<String> l=new LinkedList<String>();
        l.add("B");
        l.add("C");
        l.add("D");
        l.add("E");
        l.addLast("Z"); //it add object in last position
        l.addFirst("A"); //it add object in first position
        l.add(1,"A1"); //add the Object specified index
        System.out.println("original content:-"+l);
        l.removeFirst(); //remove first Object
        l.removeLast(); //remove last t Object
        System.out.println("after deletion first & last:-"+l);
        l.remove("E"); //remove specified Object
        l.remove(2); //remove the object of specified index
        System.out.println("after deletion :-"+l);//A1 B D
        String val = l.get(0); //get method used to get the element
        l.set(2,val+"cahged"); //set method used to replacement
        System.out.println("after seting:-"+l);
    }
};
```

D:\>java Test

original content:-[A, A1, B, C, D, E, Z]  
 after deletion first & last:-[A1, B, C, D, E]  
 after deletion :-[A1, B, D]  
 after seting:-[A1, B, A1cahged]

**Example:-Adding one collection data into another Collection.**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> al = new ArrayList<String>();
        al.add("ratan");
        al.add("balu");

        LinkedList<String> linked = new LinkedList<String>(al);
        linked.add("anu");
        linked.add("simran");
        System.out.println(linked);
    }
}
```

**E:\>java Test  
[ratan, balu, anu, simran]**

**Example :- LinkedList cloning process:-**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        LinkedList<String> arrl = new LinkedList<String>();
        arrl.add("First");
        arrl.add("Second");
        arrl.add("Third");
        arrl.add("Random");
        System.out.println("Actual LinkedList:"+arrl);
        LinkedLis copy = (LinkedLis) arrl.clone();
        System.out.println("Cloned LinkedList:"+copy);
    }
}
```

**E:\>java Test**  
**Actual LinkedList:[First, Second, Third, Random]**  
**Cloned LinkedList:[First, Second, Third, Random]**

**Vector:- (legacy class introduced in 1.0 version)**

```
public class java.util.Vector extends java.util.AbstractList
    implements java.util.List<E>,
               java.util.RandomAccess,
               java.lang.Cloneable,
               java.io.Serializable
```

- 1) Introduced in 1.0 version it is a legacy class.
- 2) Heterogeneous objects are allowed.
- 3) Duplicate objects are allowed.
- 4) Null insertion is possible.
- 5) Insertion order is preserved.
- 6) The underlying data structure is growable array.
- 7) Vector methods are synchronized.
- 8) Applicable cursors are Iterator,Enumeration,ListIterator.

**Vector constructors:-**

```
Vector();
Vector(int initialCapacity);
Vector(int initialCapacity, int increment);
Vector(java.util.Collection<? extends E>);
```

**Constructor 1:-**

The default initial capacity of the Vector is 10 once it reaches its maximum capacity it means when we trying to insert 11 element that capacity will become double[20].

```
Vector v = new Vector();
System.out.println(v.capacity());      //10
v.add("ratan");
System.out.println(v.capacity());      //10
System.out.println(v.size());         //1
```

**Constructor 2:-**

*It is possible to create vector with specified capacity by using fallowing constructor. in this case once vector reaches its maximum capacity then size is double based on provided initial capacity.*

```
Vector v = new Vector(int initial-capacity);
Vector<String> vv = new Vector<String>(3);
System.out.println(vv.capacity());   //3
vv.add("aaa");
vv.add("bbb");
vv.add("ccc");
vv.add("ddd");
System.out.println(vv.capacity());   //6
System.out.println(vv.size());       //4
```

**Constructor 3:-**

*It is possible to create vector with initial capacity and providing increment capacity by using fallowing constructor.*

```
Vector v = new Vector(int initial-capacity, int increment-capacity);
Vector<String> v = new Vector<String>(2,5);
System.out.println(v.capacity());     //2
v.add("ratan");
v.add("aruna");
v.add("Sravya");
System.out.println(v.capacity());     //7
System.out.println(v.size());        //3
```

**Constructor 4:-**

```
Vector(java.util.Collection<? extends E>);
```

*It creates the Vector that contains another Collection data.*

**Example:-**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> al = new ArrayList<String>();
        al.add("no1");
        al.add("no2");

        Vector<String> v = new Vector<String>(al);
        v.add("ratan");
        v.add("aruna");
        System.out.println(v);
```

```

        ArrayList<String> a2 = new ArrayList<String>(v);
        a2.add("xxx");
        a2.add("yyy");
        System.out.println(a2);
    }
}

E:\>java Test
[no1, no2, ratan, aruna]
[no1, no2, ratan, aruna, xxx, yyy]

```

**Example:-**

In below example Vector class removeElement() method removes the data always based on object but not index.

|                                                                                                                           |                                                                                                              |
|---------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|
| Vector v=new Vector();<br>v.addElement("ratan");<br>v.removeElement("ratan");<br>System.out.println(v); //[] empty output | Vector v=new Vector();<br>v.addElement("ratan");<br>v.removeElement(0);<br>System.out.println(v); // [ratan] |
|---------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|

The List interface remove() method removes the data based on index and object.

```

Vector v=new Vector();
v.addElement("ratan");
v.remove(0);
System.out.println(v); //[] empty output

```

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Vector<Integer> v=new Vector<Integer>(); //generic version of vector
        for (int i=0;i<5 ;i++ )
        {
            v.addElement(i);
        }
        v.addElement(6);
        v.removeElement(1); //it removes element object based
        Enumeration<Integer> e = v.elements();
        while (e.hasMoreElements())
        {
            Integer i = e.nextElement();
            System.out.println(i);
        }
        v.clear(); //it removes all objects of vector
        System.out.println(v);
    }
}

E:\>java Test
01246[]

```

**Copying data from Vector to ArrayList:-**

To copy data from one class to another class use **copy()** method of Collections class.

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> al = new ArrayList<String>();
        al.add("10");
        al.add("20");
        al.add("30");
        Vector<String> v = new Vector<String>();
        v.add("ten");
        v.add("twenty");
        //copy data from vector to ArrayList
        Collections.copy(al,v);
        System.out.println(al);
    }
}
D:\vikram>java Test
[ten, twenty, 30]
```

**Passing data {ArrayList to Vector} & Vector to ArrayList:-**

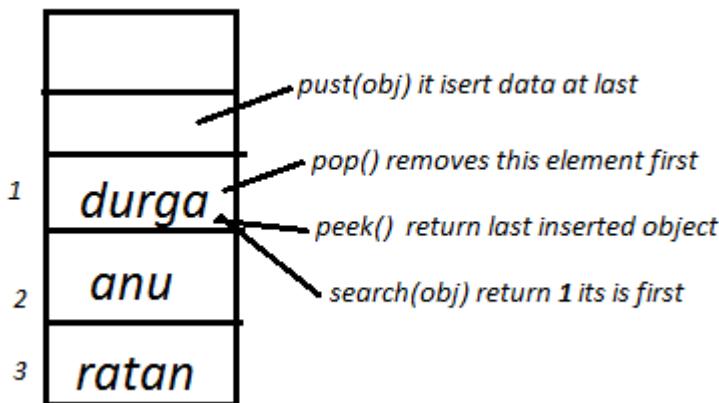
```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> a1 = new ArrayList<String>();
        a1.add("ratan");
        a1.add("anu");
        a1.add("Sravya");
        a1.add("yadhu");
        //ArrayList - Vector
        Vector<String> v = new Vector<String>(a1);
        v.add("xxx");
        v.add("yyy");
        System.out.println(v);      //|[ratan, anu, Sravya, yadhu, xxx, yyy]
        //Vector-ArrayList
        ArrayList<String> a2 = new ArrayList<String>(v);
        a2.add("suneel");
        System.out.println(a2);      //|[ratan, anu, Sravya, yadhu, xxx, yyy, suneel]
    }
}
```

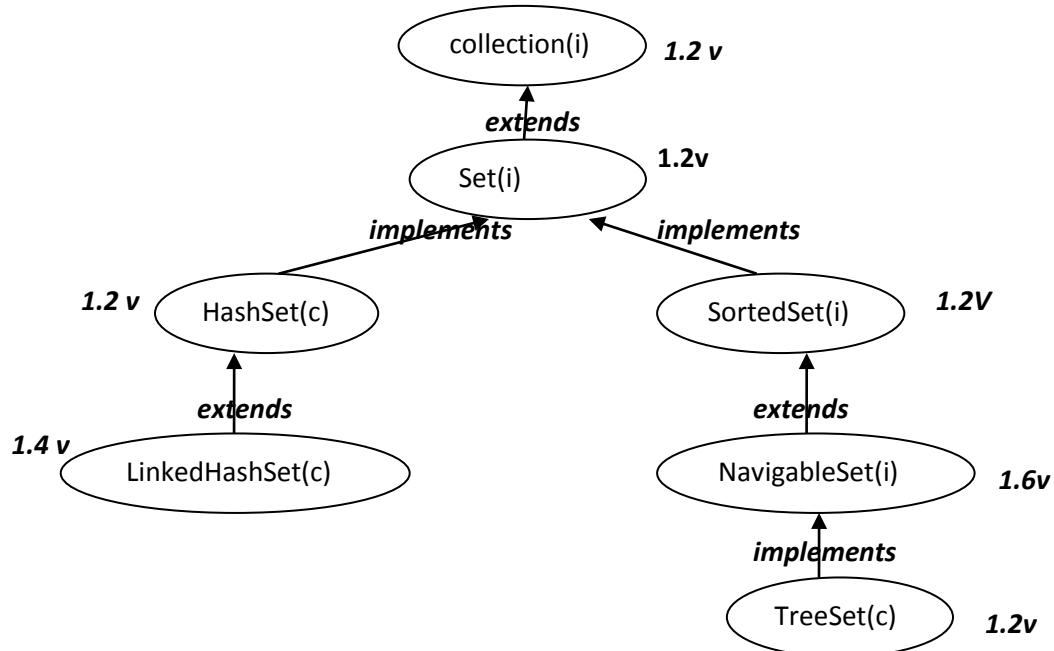
**Stack:- (legacy class introduced in 1.0 version)**

- 1) It is a child class of vector.
- 2) Introduce in 1.0 version it is a legacy class.
- 3) It is designed for LIFO(last in fist order ).

Example:-

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Stack<String> s = new Stack<String>();
        s.push("ratan");           //insert the data top of the stack
        s.push("anu");            //insert the data top of the stack
        s.push("Sravya");
        System.out.println(s);
        System.out.println(s.search("Sravya")); //1 last added object will become first
        System.out.println(s.size());
        System.out.println(s.peek());      //to return last element of the Stack
        s.pop();                      //remove the data top of the stack
        System.out.println(s);
        System.out.println(s.isEmpty());
        s.clear();
        System.out.println(s.isEmpty());
    }
}
```



**Set interface:-****Java.util.HashSet:-**

```

public class java.util.HashSet extends java.util.AbstractSet
        implements java.util.Set<E>,
                   java.lang.Cloneable,
                   java.io.Serializable

```

**Note:- in entire Collections <E> specifies the type of the Object the Collection implementation classes will hold.**

**constructors:-**

**HashSet();** it creates default HashSet.

**HashSet(java.util.Collection<? extends E>);**

It initialize the HashSet by passing another collection data.

**HashSet(int capacity);**

It create the HashSet by specified capacity. And the default capacity of HashSet is 16.

**HashSet(int capacity, float fillRatio);**

It initialize both capacity and fillratio(also called as load factor) and fillratio must be 0.0 to 1.0 after filling this ratio a new HashSet object is created.

The default fill ratio is 0.75.

**Note :-** The Set interface and HashSet,LinkedHashSet class does not contains new methods it uses super class methods if you want check the predefined support by using **javap** command.

**Javap java.util.Set**

**Javap java.util.HashSet**

- 1) Introduced in 1.2 version.
- 2) Heterogeneous objects are allowed.
- 3) Duplicate objects are not allowed if we are trying to insert duplicate values then we won't get any compilation & Execution errors simply add method returns false .
- 4) Null insertion is possible but if we are inserting more than one null it return only one null value (because duplicates are not allowed).
- 5) The underlying data structure is HashTable.
- 6) Insertion order is not preserved it is based on the hash code of the object (hashing mechanism).
- 7) Methods are non-synchronized.
- 8) It supports only Iterator cursor to retrieve the data.

**Example:-HashSet data duplication.**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        HashSet<String> h = new HashSet<String>();
        h.add("C");
        h.add("D");
        System.out.println(h.add("D"));
        System.out.println(h.add("D"));
        System.out.println(h);
    }
}
E:\>java Test
false
false
[D, C]
```

**Example:- HashSet with cursor.**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        HashSet<String> h = new HashSet<String>();
        h.add("A");
        h.add("B");
        h.add("C");
        h.add("D");
        h.add("D");
        //creation of Iterator Object
        Iterator<String> itr = h.iterator();
        while (itr.hasNext())
        {
            String str = itr.next();
            System.out.println(str);
        }
    }
}
```

**Example:-Adding one collection data into another.**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        HashSet<String> h1 = new HashSet<String>();
        h1.add("ratan");
        h1.add("anu");
        h1.add("Sravya");
        HashSet<String> h2 = new HashSet<String>(h1);
        h2.add("no1");
        h2.add("no2");
        System.out.println(h2);
    }
}
E:\>java Test
[Sravya, ratan, anu, no2, no1]

```

**Java.util.LinkedHashSet:-**

```

public class java.util.LinkedHashSet      extends      java.util.HashSet
  implements   java.util.Set<E>,
  java.lang.Cloneable,
  java.io.Serializable

```

1. Introduced in 1.4 version and It is a child class of HashSet.
2. Heterogeneous objects are allowed.
3. Duplicate objects are not allowed if we are trying to insert duplicate values then we won't get any compilation & Execution errors simply add method return false.
4. Insertion order is preserved.
5. Null insertion is possible only once(because duplication is not possible).
6. The underlying data structure is LinkedList & hashTable.
7. Methods are non-synchronized.
8. It supports only Iterator cursor to retrieve the data.

**Constructors:-**

```

LinkedHashSet();
LinkedHashSet(java.util.Collection<? extends E>);
LinkedHashSet(int capacity);
LinkedHashSet(int capacity,float fillRatio);

```

**Example:-**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Set<String> h = new LinkedHashSet<String>();
        h.add("A");
        h.add("B");
        h.add("C");
        h.add("D");
        h.add("D");
        Iterator<String> itr = h.iterator();
    }
}

```

```

        while (itr.hasNext())
        {String str = itr.next();
         System.out.print(str);
        }
    }
E:\>java Test
ABCD

```

**Java.util.TreeSet:-**

```

public class java.util.TreeSet      extends      java.util.AbstractSet<E>
   implements   java.util.NavigableSet<E>,
   java.lang.Cloneable,
   java.io.Serializable

```

*<E> specifies the type of the Object the set will hold.*

**Constructors:-**

**TreeSet();**

*It will create empty TreeSet that will be sorted in ascending order according to natural order of its elements.*

**TreeSet(java/util/Collection<? extends E>);**

*It creates the TreeSet with some collection data.*

**TreeSet(java/util/Comparator<? super E>);**

*It will create empty TreeSet with comparator specified sorting order (customization or sorting).*

**TreeSet(java/util/SortedSet<E>);**

*It builds the TreeSet that contains the elements of SortedSet.*

1. *TreeSet introduced in 1.2 version.*
2. *Heterogeneous data is not allowed.*
3. *Insertion order is not preserved but it sorts the elements in some sorting order.*
4. *Duplicate objects are not allowed.*
5. *Null insertion is possible only once.*
6. *TreeSet Methods are non-synchronized.*
7. *The underlying data Structure is Balanced Tree.*
8. *It supports Iterator cursor to retrieve the data.*

**Case -1:**      `TreeSet<String> t=new TreeSet<String>();`

```

t.add("ratan");
t.add("anu");
t.add("sravya");
System.out.println(t); // [anu, ratan, sravya]

```

- *When we insert the data in TreeSet, by default it prints the data in sorting order(ascending or alphabetical order) because it is implementing SortedSet interface.*

- To perform the sorting internally it uses `compareTo()` method and it compare the two objects it returns int value as a return value.

|                                         |               |     |                                |
|-----------------------------------------|---------------|-----|--------------------------------|
| <code>"ratan".compareTo("anu")</code>   | $\Rightarrow$ | +ve | $\Rightarrow$ change the order |
| <code>"ratan".compareTo("ratan")</code> | $\Rightarrow$ | 0   | $\Rightarrow$ no change        |
| <code>"anu".compareTo("ratan")</code>   | $\Rightarrow$ | -ve | $\Rightarrow$ no change        |

#### Case 2:-

```
TreeSet t=new TreeSet();
t.add("ratan");
t.add("anu");
t.add(10);      // java.lang.ClassCastException
System.out.println(t);
```

- ✓ TreeSet allows homogeneous data, if we are trying to insert heterogeneous data while performing sorting by using `compareTo()` JVM will generate `java.lang.ClassCastException` (because it is not possible to compare integer data with String) .

#### Case 3:-

```
TreeSet t=new TreeSet();
t.add("ratan");
t.add(null); //java.lang.NullPointerException
System.out.println(t);
```

- If the TreeSet contains data if we are trying to insert null value at the time of comparison JVM will generate `//java.lang.NullPointerException`.
- In java any object with comparison of null it will generate `java.lang.NullPointerException`.

#### Case 4:-`

```
TreeSet t=new TreeSet();
t.add(null);
System.out.println(t);//[null]
```

- ✓ In empty TreeSet it is possible to insert null value because it is not performing any comparisons.

#### Example:-TreeSet default sorting order.

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        TreeSet<String> t = new TreeSet<String>();
        t.add("ratan");
        t.add("anu");
        t.add("sravya");
        System.out.println(t);
        TreeSet<Integer> t1 = new TreeSet<Integer>();
        t1.add(10);
        t1.add(12);
        t1.add(8);
        System.out.println(t1);
    }
}
```

E:\>java Test  
[anu, ratan, sravya]  
[8, 10, 12]

**Example:-TreeSet customized Sorting Order.**

```

import java.util.*;
class Fruit
{
    public static void main(String[] args)
    {
        TreeSet<String> t = new TreeSet<String>(new MyComp());
        t.add("orange");
        t.add("banana");
        t.add("apple");
        System.out.println(t);
    }
}
class MyComp implements Comparator<String>
{
    public int compare(String s1, String s2)
    {
        return s1.compareTo(s2); // [apple, banana, orange]
        //return -s1.compareTo(s2); // [orange, banana, apple]
    }
};

```

**Example:-Different possibilities of sorting order.**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        TreeSet<Integer> t = new TreeSet<Integer>(new MyComp()); // line-1
        t.add(50);
        t.add(20);
        t.add(40);
        t.add(10);
        t.add(30);
        System.out.println(t);
    }
}
import java.util.*;
class MyComp implements Comparator
{
    public int compare(Object o1, Object o2)
    {
        Integer i1 = (Integer)o1;
        Integer i2 = (Integer)o2;
        //check all possibilities by placing comments
        //return i1.compareTo(i2);
        //return -i1.compareTo(i2);
        //return i2.compareTo(i1);
        //return -i2.compareTo(i1);
        //return -1;
        //return +1;
        //return 0;
    }
}

```

**Observation-1:**

in above example at line number-1 if we are not passing comparator object then JVM will call `compareTo()` method as part of default sorting order .

Based on above line the default sorting will done by using `compareTo()` method.

**Observation-2:-**

In above example at line number 1 if we are passing comparator object then JVM will call `compare()` method to perform sorting instead of `compareTo()` method.

**Example :-write a program to insert String data into TreeSet to perform sorting in reverse of alphabetical order.**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        TreeSet<String> t = new TreeSet<String>(new MyComp());
        t.add("ratan");
        t.add("anu");
        t.add("aravya");
        t.add("aruna");
        System.out.println(t);
    }
}
class MyComp implements Comparator<String>
{
    public int compare(String s1,String s2)
    {
        return s2.compareTo(s1);
        //return -s1.compareTo(s2);
    }
};
```

**Example :-write a program to insert StringBuffer data into TreeSet to perform sorting in alphabetical order.**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        TreeSet<StringBuffer> t = new TreeSet<StringBuffer>(new MyComp());
        t.add(new StringBuffer("ccc"));
        t.add(new StringBuffer("aaa"));
        t.add(new StringBuffer("ddd"));
        t.add(new StringBuffer("bbb"));
        System.out.println(t);
    }
}
class MyComp implements Comparator<StringBuffer>
{
    public int compare(StringBuffer sb1,StringBuffer sb2)
    {
        String s1 = sb1.toString();
        String s2 = sb2.toString();
        //return s2.compareTo(s1);
        return -s1.compareTo(s2);
    }
};
```

**Example :- write a program to insert String & StringBuffer object into TreeSet, where sorting is increasing length order. If two objects are having same length then use alphabetical order.**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        TreeSet t = new TreeSet(new MyComp());
        t.add("ratan");
        t.add(new StringBuffer("sravya"));
        t.add("anu");
        t.add(new StringBuffer("suneelbabu"));
        t.add("sri");
        System.out.println(t);
    }
}
class MyComp implements Comparator
{
    public int compare(Object o1, Object o2)
    {
        String s1 = o1.toString();
        String s2 = o2.toString();
        int l1=s1.length();
        int l2=s2.length();
        if (l1<l2)
        {
            return -1;
        }
        else if (l1>l2)
        {
            return 1;
        }
        else
        {
            return s1.compareTo(s2);
        }
    }
};
```

**Example:- passing sortedset object to TreeSet constructor.**

```
import java.util.*;
class Sravya
{
    public static void main(String[] args)
    {
        TreeSet<Integer> t=new TreeSet<Integer>();
        t.add(20);
        t.add(40);
        t.add(10);
        t.add(30);
        System.out.println(t); //10 20 30 40
        SortedSet s = t.headSet(30);
        TreeSet tt = new TreeSet(s);
        System.out.println(tt); //10 20
    }
}
```

**Example :-Elimination duplicate objects by using set interface.**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        String[] str={"ratan","anu","sravya","anu"};
        List<String> l = Arrays.asList(str);

        TreeSet<String> t = new TreeSet<String>(l);
        System.out.println(t);
    }
}
E:\>java Test
[anu, ratan, sravya]

```

**Example :-basic operations on TreeSet.**

|                                             |                                            |
|---------------------------------------------|--------------------------------------------|
| public E first();                           | it print first element                     |
| public E last();                            | it print last element                      |
| public E lower(E);                          | it print lower object of specified object  |
| public E higher(E);                         | it print higher object of specified object |
| public java/util/SortedSet<E> subSet(E, E); | it print subset                            |
| public java/util/SortedSet<E> headSet(E);   | it print specified object above objects    |
| public java/util/SortedSet<E> tailSet(E);   | it print specified objects below values    |
| public E pollFirst();                       | it print and remove first                  |
| public E pollLast();                        | it print and remove last.                  |

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        TreeSet<Integer> t=new TreeSet<Integer>();
        t.add(50);
        t.add(20);
        t.add(40);
        t.add(10);
        t.add(30);
        System.out.println(t);//10 20 30 40 50
        System.out.println(t.headSet(30));//[10,20]
        System.out.println(t.tailSet(30));//[30,40,50]
        System.out.println(t.subSet(20,50));//[20,30,40]
        System.out.println("last element="+t.last());//50
        System.out.println("first element="+t.first());//10
        System.out.println("lower element="+t.lower(50));//10
        System.out.println("higher element="+t.higher(20));//30
        System.out.println("print & remove first element="+t.pollFirst());//10
        System.out.println("print & remove last element="+t.pollLast());//50
        System.out.println("final elements="+t);//20 30 40
        System.out.println("TreeSet size="+t.size());//3
        System.out.println(t.remove(30));
        System.out.println("TreeSet size="+t.size());//2
    }
}

```

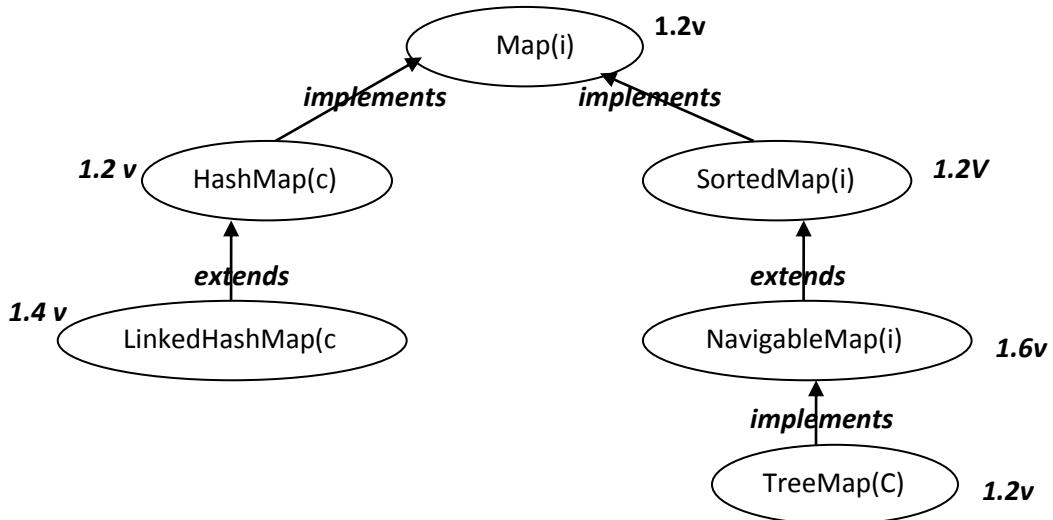
```

        System.out.println("final elements="+t);//20 40
    }
}

E:\>java Test
[10, 20, 30, 40, 50]
[10, 20]
[30, 40, 50]
[20, 30, 40]
last element=50
first element=10
lower element=40
higher element=30
print & remove first element=10
print & remove last element=50
final elements=[20, 30, 40]
TreeSet size=3
TreeSet size=true
TreeSet size=2
final elements=[20, 40]

```

### Map interface:-



### Java.util.HashMap:-

```

public class java.util.HashMap extends
    implements
        java.util.AbstractMap
        java.util.Map
        java.lang.Cloneable,
        java.io.Serializable

```

- 1) introduced in 1.2 version.
- 2) Heterogeneous data allowed.
- 3) Underlying data Structure is HashTable.
- 4) Duplicate keys are not allowed but values can be duplicated.
- 5) Insertion order is not preserved it is based on hashCode.
- 6) Null is allowed for key(only once) and allows for values any number of times.
- 7) Every method is non-synchronized so multiple Threads are operate at a time hence permanence is high.

**Constructors:-**

**HashMap();** it creates default HashMap.

**HashMap(java.util.Map<? extends K, ? extends V> var);**

it creates the HashMap by initializing the values specified in var.

**HashMap(int capacity);**

It creates the hashmap with specified capacity but the default capacity is **16**.

**HashMap(int capacity, float fillRatio);**

It creates the hashMap with specified capacity & fillRatio.(default capacity is 16 & default fill ratio 0.75)

**Entry:-**

- ✓ The each and every key value pair is called **Entry**.
- ✓ The Map contains group of entries.
- ✓ Entry is sub interface of Map interface hence get the entry interface by using Map interface.

**interface Map**

```
{     interface Entry
    {
        public abstract Object getKey();
        public abstract Object getValue();
        public abstract Object setValue();
    }
}
```

- ✓ To get all the keys use keyset() method.

**public java.util.Set<K> keySet();**

- ✓ To get all the values use values() method.

**public java.util.Collection<V> values();**

- ✓ To get all the entries use entrySet() method.

**public java.util.Set<java.util.Map\$Entry<K, V>> entrySet();**

**Example :-**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        HashMap h = new HashMap();
        h.put(111,"ratan");
        h.put(222,"anu");
        h.put(333,"banu");
        //keySet() to get all keys.
        Set s1=h.keySet();
        System.out.println("all keys:--->" + s1);
    }
}
```

```

//values() to get all the values.
Collection c = h.values();
System.out.println("all values--->" + c);
//entrySet() to get all the entries.
Set ss = h.entrySet();
System.out.println("all entries--->" + ss);
Iterator itr = ss.iterator();
while (itr.hasNext())
{
    Map.Entry m = (Map.Entry)itr.next();
    System.out.println(m.getKey() + "----" + m.getValue());
}
}

};

E:\>java Test
all keys:--->[222, 111, 333]
all values--->[anu, ratan, banu]
all entries--->[222=anu, 111=ratan, 333=banu]
222----anu
111----ratan
333----banu

```

**Java.util.LinkedHashMap:-**

public class java.util.LinkedHashMap     **extends**     java.util.HashMap  
                                          **implements**     java.util.Map

- 1) interdicted in 1.4 version
- 2) Heterogeneous data allowed.
- 3) Underlying data Structure is HashTable & linkedlist.
- 4) Duplicate keys are not allowed but values can be duplicated.
- 5) Insertion order is preserved.
- 6) Null is allowed for key(only once)and allows for values any number of times.
- 7) Every method is non-synchronized so multiple Threads are operate at a time hence permanence is high.

**Constructors:-**

**LinkedHashMap();** it creates default HashMap.

**LinkedHashMap(java/util/Map<? extends K, ? extends V> var);**

it creates the HashMap by initializing the values specified in var.

**LinkedHashMap(int capacity);**

It creates the hashmap with specified capacity but the default capacity is **16**.

**LinkedHashMap(int capacity, float fillRatio);**

It creates the hashMap with specified capacity & fillRatio.(default capacity is 16 & default fill ratio 0.75)

**Emp.java:-**

```
class Emp
{
    int eid;
    String ename;
    Emp(int eid,String ename)
    {this.eid=eid;
     this.ename=ename;
    }
}
```

**//Student.java**

```
class Student
{
    //instance variables
    int sid;
    String sname;
    Student(int sid,String sname)//local
variables
    { this.sname=sname; this.sid=sid;
    }
}
```

**Test.java:-**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        LinkedHashMap<Emp,Student> h = new LinkedHashMap<Emp,Student>();
        h.put(new Emp(111,"ratan"), new Student(1,"budha"));
        h.put(new Emp(222,"anu"), new Student(2,"ashok"));
        Set s = h.entrySet();
        Iterator itr = s.iterator();
        while (itr.hasNext())
        {
            Map.Entry m = (Map.Entry)itr.next();
            Emp e = (Emp)m.getKey();
            System.out.println(e.ename+"--"+e.eid);
            Student ss = (Student)m.getValue();
            System.out.println(ss.sname+"--"+ss.sid);
        }
    }
}
```

**Example:-**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Map<Integer,String> h1 = new LinkedHashMap<Integer,String>();
        h1.put(111,"ratan");
        h1.put(222,"sravya");

        Map<Integer,String> h2 = new LinkedHashMap<Integer,String>(h1);
        h2.put(333,"anu");

        for (Map.Entry m : h2.entrySet())
        {
            System.out.println(m.getKey()+"---"+m.getValue());
        }
    }
}
```

**Java.util.HashTable:-**

```
public class java.util.Hashtable extends java.util.Dictionary
    implements java.util.Map,java.lang.Cloneable, java.io.Serializable
```

1. Introduced in the 1.0 version it's a legacy class.
2. Heterogeneous data allowed for both key & value.
3. Duplicate keys are not allowed but values can be duplicated.
4. Every method is synchronized hence only one thread is allowed to access it is a Thread safe but performance is decreased.
5. Null is not allowed for both key & Value , if we are trying to insert null values we will get NullPointerException.
6. HashTable was made generic by JDK5.
7. The underlaying datastructure is hashtable.

**Constructors:-**

**HashTable();** it creates default HashMap.

**HashTable (java/util/Map<? extends K, ? extends V> var);**

it creates the HashMap by initializing the values specified in var.

**HashTable (int capacity);**

It creates the hashmap with specified capacity but the default capacity is **11**.

**HashTable (int capacity, float fillRatio);**

It creates the hashMap with specified capacity & fillRatio.(default capacity is 11 & default fill ratio 0.75)

Ex:-

```
import java.util.Hashtable;
import java.util.Collection;
import java.util.Set;
class Test
{
    public static void main(String[] args)
    {
        Hashtable<String, String> h = new Hashtable<String, String>();
        //adding data in HashTable
        h.put("1", "one");
        h.put("2", "two");
        h.put("3", "three");
        System.out.println(h);
        System.out.println(h.get("1")); //one
        System.out.println(h.isEmpty());
        h.remove("3");
        System.out.println(h.containsKey("1"));
        System.out.println(h.containsKey("3"));
        System.out.println(h.containsValue("one"));
        System.out.println(h.size());
        //to get all values objects
        Collection<String> c = h.values();
        for (String i : c)
        {
            System.out.println(i);
        }
    }
}
```

```

//to get all key objects
Set<String> s = h.keySet();
for (String ss : s)
{
    System.out.println(ss);
}
}
}

```

**Java.util.TreeMap:-**

- public class java.util **TreeMap** extends java.util.AbstractMap  
**implements** java.util.NavigableMap,java.lang.Cloneable, java.io.Serializable
- 1) This class is introduced in 1.2 version.
  - 2) It allows homogeneous data if we are trying to insert heterogeneous data at runtime while performing sorting JVM will generate ClassCastException.
  - 3) Duplicate keys are not allowed but values can be duplicated.
  - 4) Insertion order is not preserved it is based on some sorting order of keys.
  - 5) The underlying data structure is red-black trees.
  - 6) For empty tree set it is possible to insert null key once, but if the tree set contains data if we are inserting null keys at runtime we will get NullPointerException but for the values any number of null values insertion possible.

**Constructors:-****TreeMap();**

it will create empty treemap that will be sorted by using natural order of its keys.

**TreeMap(java/util/Comparator<? super K>);**

It creates treemap that will be sorted by using customized sorting order.

**TreeMap(java/util/Map<? extends K, ? extends V>);**

It creates the treemap with specified data.

**TreeMap(java/util/SortedMap<K, ? extends V>);**

It creates the treemap by initializing sortedmap data.

**Observations of TreeMap:****Case 1:-**      TreeMap h = new TreeMap();

```

h.put(444, "ratan");
h.put(222, "anu");
h.put(111, "aaa");
System.out.println(h); //{{111=aaa, 222=anu, 444=ratan}}

```

In treemap when we insert the data that will be printed in sorting order based on key.

**Case 2:-**      TreeMap h = new TreeMap();

```

h.put(444, "ratan");
h.put("ratan", "aaa"); //java.lang.ClassCastException
System.out.println(h);

```

Treemap allows homogeneous data, if we are inserting heterogeneous data while performing sorting it will generate **java.lang.ClassCastException**.

**Case 3:-**      TreeMap h = new TreeMap();

```

h.put(444, "ratan");
h.put(null, "aaa"); //java.lang.NullPointerException
System.out.println(h);

```

If the treemap contains data then we are adding null value hence while performing sorting it will generate **java.lang.NullPointerException**(any object with comparision of null it will generate NullPointerException )

**Case 4:-**

```
TreeMap h = new TreeMap();
h.put(null,"aaa");
System.out.println(h); // {null=aaa}
```

In empty treemap it is possible to insert null value.

**Example:-**

```
import java.util.TreeMap;
import java.util.Set;
import java.util.Collection;
import java.util.Map.Entry;
class Test
{
    public static void main(String[] args)
    {
        TreeMap<String,String> tmain = new TreeMap<String,String>();
        tmain.put("ratan","no1");
        tmain.put("anu","no2");

        TreeMap<String,String> tsub = new TreeMap<String,String>();
        tsub.putAll(tmain);
        tsub.put("x","no3");
        tsub.put("y","no4");
        System.out.println(tsub);

        if (tmain.containsKey("ratan"))
        {System.out.println("ratan is great");
        }
        if (tsub.containsValue("no1"))
        {System.out.println("no1 ratan only");
        }

        //printing all the keys
        Set<String> s = tsub.keySet();
        for (String ss : s)
        {
            System.out.println(ss);
        }

        //printing all the values
        Collection<String> s1 = tsub.values();
        for (String ss1 : s1)
        {
            System.out.println(ss1);
        }

        Set<Entry<String,String>> s2 = tsub.entrySet();
        for (Entry<String,String> ss2 : s2)
        {
            System.out.println(ss2);
        }

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

**Example:-**

```

import java.util.*;
class MyComp implements Comparator
{
    public int compare(Object o1, Object o2)
    {
        String s1 = (String)o1;
        String s2 = (String)o2;
        return s2.compareTo(s1);
    }
}
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        TreeMap h = new TreeMap(new MyComp());
        h.put("ratan",111);
        h.put("anu",222);
        h.put("zzzz",333);
        System.out.println(h); // {zzzz=333, ratan=111, anu=222}
    }
}

```

**Example:-**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        TreeMap h = new TreeMap();
        h.put(111,"ratan");
        h.put(222,"anu");
        h.put(333,"aaa");
        h.put(444,"aaa");
        System.out.println(h);

        Map m = h.subMap(222,444);
        System.out.println(m);

        System.out.println(h.firstEntry());
        System.out.println(h.lastEntry());
        System.out.println(h.firstKey());
        System.out.println(h.lastKey());
        System.out.println(h.lowerKey(222));
        System.out.println(h.higherKey(222));

        SortedMap s1 = h.headMap(333);
        TreeMap t1 = new TreeMap(s1);
        System.out.println(t1);

        SortedMap s2 = h.tailMap(333);
        TreeMap t2 = new TreeMap(s2);
    }
}

```

```

        System.out.println(t2);
    }
}

```

**Example :-****Ceiling()**

it return current provided value or greater value but if treemap does not contains same or grater value then it returns null .

**floor():-**

it returns current value or less value but if treemap does not contains same value or less then it return null.

**pollFirstEntry:-** it removes first entry & it prints that entry.

**pollLastEntry():-** it removes last entry and it prints that entry.

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        TreeMap h = new TreeMap();
        h.put(111,"ratan");
        h.put(222,"anu");
        h.put(444,"aaa");
        System.out.println(h);

        System.out.println(h.ceilingKey(222));
        System.out.println(h.ceilingEntry(333));
        System.out.println(h.floorKey(222));
        System.out.println(h.floorEntry(333));
        System.out.println(h.ceilingKey(666));

        Map.Entry m1 = h.pollFirstEntry();
        System.out.println(m1.getKey()+"---"+m1.getValue());
        Map.Entry m2 = h.pollLastEntry();
        System.out.println(m2.getKey()+"---"+m2.getValue());

        System.out.println(h);
    }
}

```

**Java.util.IdentityHashMap:-**

```

public class java.util.IdentityHashMap extends           java.util.AbstractMap
   implements     java.util.Map,java.io.Serializable, java.lang.Cloneable

```

**It is same as hashmap except one difference,**

In case of Hashmap JVM will use equals( ) method to identify duplicate keys.(it performs content comparison)

In case of identityhashmap JVM will use == operator to identify the duplicate keys.(it perform reference comparison)

**Example:-**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        //equals() method to identify duplicate keys.
        HashMap<Integer, String> h = new HashMap<Integer, String>();
        h.put(new Integer(10), "ratan");
        h.put(new Integer(10), "anu");
        System.out.println(h);

        //== operator to identify duplicate keys.
        IdentityHashMap<Integer, String> h1 = new IdentityHashMap<Integer, String>();
        h1.put(new Integer(10), "ratan");
        h1.put(new Integer(10), "anu");
        System.out.println(h1);
    }
}
E:\>java Test
{10=anu}
{10=anu, 10=ratan}

```

#### Java.util.WeakHashMap:-

```

public class java.util.WeakHashMap      extends      java.util.AbstractMap
   implements   java.util.Map

```

**WeakHashMap is same as HashMap except following difference,**

If an object is associated with hashmap that object is not destroyed even though it does not contains any reference type.

But in case of weakhashmap if the object does not contains reference type that object is eligible for garbage collector even though it associated with weakhashmap.

**HashMap**

```

import java.util.*;
class A
{
    public String toString()
    {
        return "A";
    }
    public void finalize()
    {System.out.println("object destroyed");
    }
}
class Test
{
    public static void main(String[] args)
    {
        HashMap h = new HashMap();
        A a= new A();
        h.put(a,"ratan");
        System.out.println(h);
        a=null;
        System.gc();
        System.out.println(h);
    }
}

```

E:\>java Test  
{A=ratan}  
{A=ratan}

**WeakHashMap**

```

import java.util.*;
class A
{
    public String toString()
    {
        return "A";
    }
    public void finalize()
    {System.out.println("object destroyed");
    }
}
class Test
{
    public static void main(String[] args)
    {WeakHashMap h = new WeakHashMap();
        A a= new A();
        h.put(a,"ratan");
        System.out.println(h);
        a=null;
        System.gc();
        System.out.println(h);
    }
}
E:\>java Test
{A=ratan}
{}
object destroyed

```

**Java.util.Properties:-**

- ✓ In standalone applications(JDBC) or web-applications(web sites) the data is frequently changing like,
  - a. Database username
  - b. Database password
  - c. url
  - d. driver ...etc
- ✓ in above scenario for every change must perform modifications in all .java files but it is complex.to overcome this problem use properties file.
- ✓ Properties file is a normal text file with .properties extension & it contains key=value formatted data but both key and value is string format.
- ✓ Once we done modifications on .properties file that modifications are reflected all the .java files.

**Abc.properties :-**

username = system  
password = manager  
driver = oracle.jdbc.driver.OracleDriver  
trainer = Ratan

**Test.java:-**

```
import java.util.*;
import java.io.*;
class Test
{
    public static void main(String[] args) throws FileNotFoundException,IOException
    {
        //locate properties file
        FileInputStream fis=new FileInputStream("abc.properties");
        //load the properties file by using load() method of Properties class
        Properties p = new Properties();
        p.load(fis);
        //get the data from properties class by using getProperty()
        String username = p.getProperty("username");
        String driver = p.getProperty("driver");
        String password = p.getProperty("password");
        String trainer = p.getProperty("trainer");
        //use the properties file data
        System.out.println("DataBase username="+username);
        System.out.println("DataBase password =" +password);
        System.out.println("driver =" +driver);
        System.out.println("trainer=" +trainer);
    }
}
```

**Collections:-**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> al = new ArrayList<String>();
        al.add("ratan");
        al.add("anu");
        al.add("Sravya");
        //to perform sorting use sort method of collections class
        Collections.sort(al);
        Iterator itr =al.iterator();
        while (itr.hasNext())
        {System.out.println(itr.next());
        }
    }
}
```

## **Networking(java.net package)**

### **Introduction to networking:-**

- 1) The process of connecting the resources (computers) together to share the data is called networking.
- 2) Java.net is package it contains number of classes by using that classes we are able to connection between the devices (computers) to share the information.
- 3) Java socket programming provides the facility to share the data between different computing devices.
- 4) In the network we are having to components
  - a. Sender(source) : The person who is sending the data is called sender.
  - b. Receiver(destination) : The person who is receiving the data is called receiver

**In the network one system can acts as a sender as well as receiver.**
- 5) In the networking terminology we have client and server.  
**Client :-** who takes the request & who takes the response is called client.  
**Server :-** The server contains the project
  1. It takes the request from the client
  2. It identifies the requested resource
  3. It process the request
  4. It will generate the response to client

### **Categories of network:-**

We are having two types of networks

- 1) Peer-to-peer network.
- 2) Client-server network.

**Client-server:-** In the client server architecture always client system acts as a client and server system acts as a server.

**Peer-to-peer:-** In the peer to peer client system sometimes behaves as a server, server system sometimes behaves like a client the roles are not fixed.

### **Types of networks:-**

**Intranet:-** It is also known as a private network. To share the information in limited area range (within the organization) then we should go for intranet.

**Extranet:-** This is extension to the private network means other than the organization , authorized persons able to access.

**Internet:-** It is also known as public networks. Where the data maintained in a centralized server hence we are having more sharability. And we can access the data from anywhere else.

### **The frequently used terms in the networking:-**

- 1) IP Address
- 2) URL(Uniform Resource Locator)
- 3) Protocol
- 4) Port Number
- 5) MAC address.
- 6) Connection oriented and connection less protocol
- 7) Socket.

**IP Address:-**

- 1) IP Address is a unique identification number given to the computer to identify the computer uniquely in the network.
- 2) The IP Address is uniquely assigned to the computer it should not be duplicated.
- 3) The IP Address range is 0-255.  
125.0.4.255      ----> Valid      124.654.5.6      ----> Invalid
- 4) Each and every website contains its own IP Address we can access the sites through the names otherwise IP Address.

Site Name : -www.google.com      IP Address : -74.125.224.72

**Example:-**

```
import java.net.*;
import java.util.*;
class Test
{
    public static void main(String[] args) throws Exception
    {
        Scanner s = new Scanner(System.in);
        System.out.println("please enter site name");
        String sitename=s.nextLine();
        InetAddress in=InetAddress.getByName(sitename);
        System.out.println("the ip address is:"+in);
    }
}
```

**G:\ratan>java Test**

please enter site name  
www.google.com  
the ip address is:www.google.com/216.58.199.196

**G:\ratan>java Test**

please enter site name  
aaa  
Exception in thread "main" java.net.UnknownHostException: aaa

**Protocol:-**

The protocol is a set of rules followed in communication.

- ✓ TCP(Transmission Control Protocol)(connection oriented protocol)
- ✓ UDP (User Data Gram Protocol)(connection less protocol)
- ✓ Telnet
- ✓ SMTP(Simple Mail Transfer Protocol)
- ✓ IP (Internet Protocol)

**MAC (media access control):-**

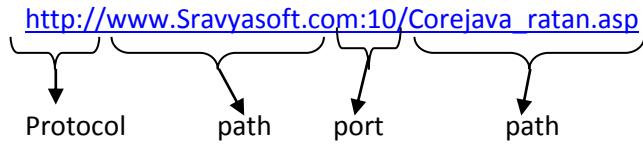
MAC address is a unique identifier for NIC (Network interface protocol). A network protocol can have multiple NIC but one unique MAC.

**Port number:**

The port number is used to identify the different applications uniquely .And it is associated with the ip address for communication between two applications.

**URL(Uniform Resource Locator):-**

- 1) URL is a class present in the java.net package.
- 2) By using the URL we are accessing some information present in the world wide web.



The URL contains information like

- a. Protocol : `http://`
- b. Server name IP address : `www.Sravyasoft.com`
- c. Port number of the particular application and it is optional(:10)
- d. File name or directory name `Corejava_ratan.asp`

**Example:-**

```
import java.net.*;
class Test
{
    public static void main(String[] args) throws Exception
    {
        URL url=new URL("http://www.Sravyasoft.com:10/index.html");
        System.out.println("protocol is:"+url.getProtocol());
        System.out.println("host name is:"+url.getHost());
        System.out.println("port number is:"+url.getPort());
        System.out.println("path is:"+url.getPath());
        System.out.println(url);
    }
}
```

**Communication using networking :-**

In the networking it is possible to do two types of communications.

- 1) Connection oriented(TCP/IP communication)
- 2) Connection less(UDP Communication)

**Connection Oriented:-**

- a) In this type of communication we are using combination of two protocols TCP,IP.
  - b) In this communication acknowledgement sent by receiver so it is reliable but slow.
- To achieve the following communication the java peoples are provided the following classes.

- a. Socket
- b. ServerSocket

**Connection Less :- (UDP)**

- 1) UDP is a protocol by using this protocol we are able to send data without using Physical Connection.
  - 2) In this communication acknowledgement not sent by receiver so it is not reliable but fast.
  - 3) This is a light weight protocol because no need of the connection between the client and server .
- To achieve the UDP communication the java peoples are provided the following classes.

1. DatagramPacket.
2. DatagramSocket.

**Socket:-**

- 1) *Socket is used to create the connection between the client and server.*
- 2) *Socket is nothing but a combination of IP Address and port number.*
- 3) *The socket is created at client side.*

```
Socket s=new Socket(int IPAddress, int portNumber);
Socket s=new Socket("125.125.0.5",123);           Server IP Address. Server port number.
Socket s=new Socket(String HostName, int PortNumber);  Socket s=new Socket(Sravyasoft,123);
```

**Client.java:-**

```
import java.net.*;
import java.io.*;
class Client
{
    public static void main(String[] args) throws Exception
    {
        //to write the data to server
        Socket s=new Socket("localhost",5555);
        String str="ratan from client";
        OutputStream os=s.getOutputStream();
        PrintStream ps=new PrintStream(os);
        ps.println(str); //write the data to server
        //read the data from server
        InputStream is=s.getInputStream();
        BufferedReader br=new BufferedReader(new InputStreamReader(is));
        String str1=br.readLine();
        System.out.println(str1);
    }
}
```

**Server.java:-**

```
import java.io.*;
import java.net.*;
class Server
{
    public static void main(String[] args) throws Exception
    {
        //To read the data from client
        ServerSocket ss=new ServerSocket(5555);
        Socket s=ss.accept();
        System.out.println("connection is created ");
        Thread.sleep(2000);
        //Read the data from client
        InputStream is=s.getInputStream();
        BufferedReader br=new BufferedReader(new InputStreamReader(is));
        String data=br.readLine();
        System.out.println(data);
        Thread.sleep(2000);
        //write the data to the client
        data=data+"this is from server";
        OutputStream os=s.getOutputStream();
        PrintStream ps=new PrintStream(os);
        ps.println(data);
    }
}
```

### ***Java.awt package***

- ❖ AWT(Abstract Window Tool kit) is an **API** it supports graphical user interface programming.
  - ❖ AWT components are platform dependent it displays the application according to the view of operating system.
  - ❖ By using java.awt package we are able to prepare static components to provide the dynamic nature to the component use **java.awt.event** package.(it is a sub package of java.awt).
1. This application not providing very good look and feel hence the normal users facing problem with these types of applications.
  2. By using AWT we are preparing application these applications are called console based or CUI application.

**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 :-**

- ✓ The root class of java.awt package is Component class.
- ✓ Component is an object which is displayed pictorially on the screen.  
Ex:- Button,Label,TextField.....etc

**Container:-**

- it is a component in awt that contains another components like Button,TextField...etc
- Container is a sub class of Component class.
- The classes that extends container classes those classes are containers such as Frame, Dialog and Panel.

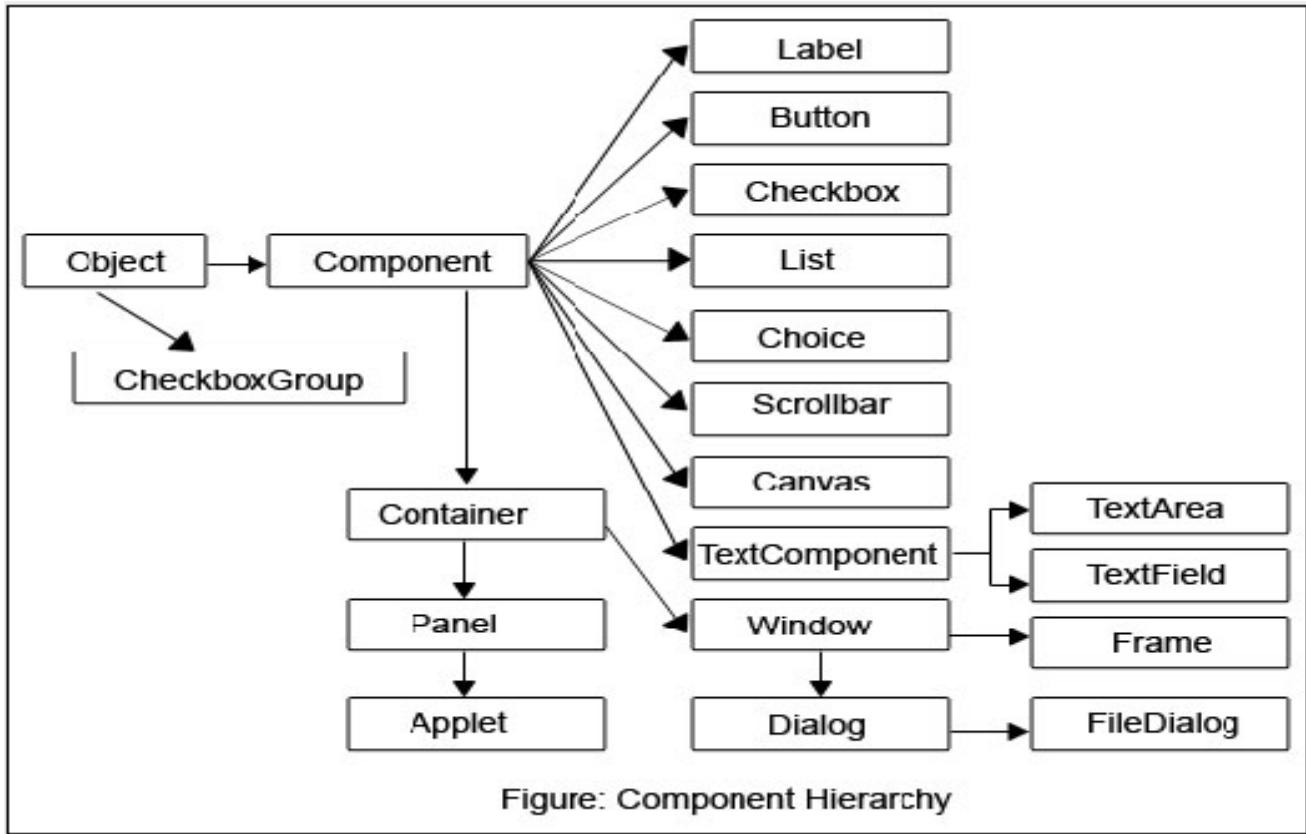
**Event:-**

The event nothing but a 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.

**AWT Component Hierarchy:-**



#### Java.awt.Frame:-

Frame is a container it contains other components like title bar, Button, Text Field...etc.

- ✓ When we create a Frame class object. Frame will be created automatically with invisible mode, so to provide the visible nature to the frame use setVisible() method of Frame class.

**public void setVisible(boolean b)**

where b==true visible mode

b==false means invisible mode.

- ✓ When we created a frame, the frame is created with initial size 0 pixel heights & 0 pixel width hence it is not visible. To provide particular size to the Frame use setSize() method.

**public void setSize(int width,int height)**

- ✓ To provide title to the frame use, **public void setTitle(String Title)**

- ✓ 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)**

#### **Example-1:- There are two approaches to create Frame in java**

1. By creating object of Frame class.
2. By extending the Frame class.

#### **Approach 1:- Creation of Frame by creating Object of Frame class.**

```
import java.awt.*;
class Demo
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setSize(400,400);
        f.setBackground(Color.red);
        f.setTitle("myframe");
    }
};
```

#### **Approach 2:- Taking user defined class by extending Frame class.**

```
import java.awt.*;
class MyFrame extends Frame
{
    MyFrame()
    {
        setVisible(true);
        setSize(500,500);
        setTitle("myframe");
        setBackground(Color.green);
    }
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
}
```

#### **Displaying text on the screen:-**

1. To display some textual message on the frame override paint() method.

```
public void paint(Graphics g)
```

2. To set a particular font to the text use Font class present in java.awt package

```
Font f=new Font("String type , int style , int size);  
Font f= new Font("arial",Font.Bold,30);
```

Example :-

```
import java.awt.*;  
class MyFrame extends Frame  
{    public static void main(String[] args)  
    {        MyFrame t = new MyFrame();  
        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);  
    }  
}
```

- ✓ In above example when we create the MyFrame class object , jvm will executes MyFrame class constructor just before this JVM will execute Frame class zero argument constructor.
- ✓ The Frame class zero argument constructor calling repaint() method & this method will access predefined Frame class paint() method. But as per the requirement overriding paint() method will be executed.

**Preparation of components:-**

1. **Label** :- *Label is a constant text which is displayed along with a TextField or TextArea.*

**Constructor:**      Label l=new Label();  
                         Label l=new Label("user name");

2. **TextField**:- *TextField is an editable area & it is possible to provide single line of text.*

*Enter Button doesn't work on TextField.*

- a. To set Text to the textarea : **t.setText("Sravya");**
- b. To get the text form TextArea : **String s=t.getText();**

**Constructors:**      TextFiled tx=new TextFiled();  
                         TextField tx=new TextField("ratan");

3. **TextArea**:- *TextArea is a Editable Area & enter button will work on TextArea.*

**Constructors:**      TextArea t=new TextArea();  
                         TextArea t=new TextArea(int rows,int columns);  
                         TextArea t=new TextArea(String text, int rows,int columns);

- ✓ To set Text to the textarea : **ta.setText("Sravya");**
- ✓ To get the text form TextArea : **String s=ta.getText();**

4. **Button** :- *Used to perform operations by clicking.*

**Example :-**

```
import java.awt.*;
class MyFrame
{
    MyFrame()
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);
        f.setLayout(new FlowLayout());           //information about layout check next page
        Label l1=new Label("user name:");
        Label l2=new Label("user password:");

        TextField tx1 = new TextField(30);
        TextField tx2 = new TextField(30);

        Button b = new Button("login");

        f.add(l1);
        f.add(tx1);
        f.add(l2);
        f.add(tx2);
        f.add(b);
    }
    public static void main(String[] args)
    {
        MyFrame f = new MyFrame();
    }
}
```

5. **Choice**:- List is allows to select multiple items but choice is allow to select single Item.

**Choice ch=new Choice();**

- |                                                           |                                         |
|-----------------------------------------------------------|-----------------------------------------|
| ✓ To add items to the choice                              | : <b>add()</b>                          |
| ✓ To remove item from the choice based on String          | : <b>choice.remove("HYD");</b>          |
| ✓ To remove the item based on the index position          | : <b>choice.remove(2);</b>              |
| ✓ To remove the all elements                              | : <b>ch.removeAll();</b>                |
| ✓ To inset the data into the choice based on the position | : <b>choice.insert(2,"ratan");</b>      |
| ✓ To get selected item from the choice                    | : <b>String s=ch.getSelectedItem();</b> |
| ✓ To get the selected item index number                   | : <b>int a=ch.getSelectedIndex();</b>   |

6. **List**: List is providing list of options to select.

**CONSTRUCTOR:-**

**List l=new List();** It will creates the list by default size is four elements.

**List l=new List(3);** It will display the three items size and it is allow selecting the only one.

**List l=new List(5,true);** It will display five items and it is allow selecting the multiple items.

- |                                                      |                                           |
|------------------------------------------------------|-------------------------------------------|
| ✓ To add the elements to the List                    | : <b>list.add("c");</b>                   |
| ✓ To add the elements to the List at specified index | : <b>list.add("ratan",0);</b>             |
| ✓ To remove element from the List                    | : <b>list.remove("c");</b>                |
| ✓ To get selected item from the List                 | : <b>String x=l.getSelectedItem();</b>    |
| ✓ To get selected items from the List                | : <b>String[] x=s.getselectedItems();</b> |

**Example :-**

```
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());
        //Choice information
        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",2);        f.add(ch);
        System.out.println(ch.getItem(0));
        System.out.println(ch.getItemCount());
        //List information
        List l=new List(3,true);
        l.add("c");                 l.add("cpp");       l.add("java");
        l.add("ratan");             l.add("arun",0);    l.remove(0);
        f.add(l);
        System.out.println(l.getItem(0));
        System.out.println(l.getItemCount());
    }
}
```



**7. Checkbox:** - The user can select more than one checkbox at a time.

```
Checkbox cb2=new Checkbox("MCA");
Checkbox cb3=new Checkbox("BSC",true);
    ✓ To set a label to the CheckBox explicitly : cb.setLabel("BSC");
    ✓ To get the label of the checkbox : String str=cb.getLabel();
    ✓ To get state of the CheckBox : Boolean b=ch.getState();
```

**8. RADIO BUTTON:-**

- ✓ AWT does not provide any predefined support to create Radio Buttons directly.
- ✓ It is possible to create RadioButton by using two classes.
  - CheckboxGroup
  - Checkbox

step 1:- Create Checkbox group object. **CheckboxGroup cg=new CheckboxGroup();**

step 2:- pass Checkboxgroup object to the Checkbox class argument.

```
Checkbox cb1=new Checkbox("male",cg,true);
Checkbox cb2=new Checkbox("female",cg,false);
    ✓ To get the status of the RadioButton : String str=Cb.getState();
    ✓ To get Label of the RadioButton : String str=getLabel().
```

**Example:-**

```
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 l1 = new Label("Qualifications:");
        Label l2 = new Label("Gender:");
        Checkbox cb1=new Checkbox("SSC",true);
        Checkbox cb2=new Checkbox("DEGREE");
        Checkbox cb3=new Checkbox("MCA");
        Checkbox cb4=new Checkbox("BTECH");
        f.add(l1);       f.add(cb1);     f.add(cb2);     f.add(cb3);   f.add(cb4);
        System.out.println(cb1.getLabel());
        System.out.println(cb2.getState());

        CheckboxGroup cg=new CheckboxGroup();
        Checkbox r1=new Checkbox("male",cg,true);
        Checkbox r2=new Checkbox("female",cg,false);
        f.add(l2);           f.add(r1);          f.add(r2);
        System.out.println(cb1.getLabel());
        System.out.println(cb1.getState());
    }
}
```



**Layout Managers:-**

When we are trying to add the components into container without using layout manager the components are overriding hence the last added component is visible on the container instead of all.

To overcome above problem to arrange the components into container in specific manner use layout manager.

[http://java.icmc.usp.br/books/ooc/html/gui\\_layout\\_manager.html](http://java.icmc.usp.br/books/ooc/html/gui_layout_manager.html)

**Definitions:-**

The layout managers are used to arrange the components in a Frame in particular manner. or A layout manager is an object that controls the size and the position of components in a container

Different layouts in java,

- 1) `java.awtFlowLayout`
- 2) `java.awt.BorderLayout`
- 3) `java.awt.GridLayout`
- 4) `java.awt.CardLayout`
- 5) `java.awt.GridBagLayout`

**java.awt.FlowLayout**

The FlowLayout is used to arrange the components into row by row format. Once the first row is filled with components then it is inserted into second row. And it is the default layout of the applet.

**Java.awt.BorderLayout:-**

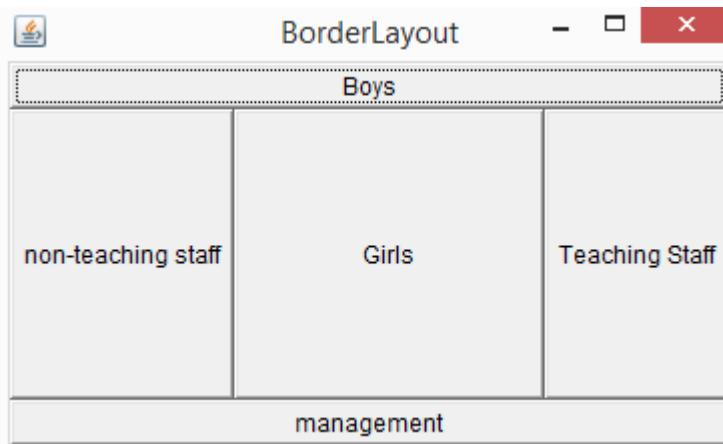
The BorderLayout is dividing the frame into five areas north,south,east,west,center so we can arrange the components in these five areas.

To represent these five areas borderlayout is providing the following 5-consts

```
public static final java.lang.String NORTH;
public static final java.lang.String SOUTH;
public static final java.lang.String EAST;
public static final java.lang.String WEST;
public static final java.lang.String CENTER;

import java.awt.*;
class MyFrame extends Frame
{
    Button b1,b2,b3,b4,b5;
    MyFrame()
    {
        //this keyword is optional because all methods are current class methods only
        this.setSize(400,400);
        this.setVisible(true);
        this.setTitle("BorderLayout");
        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);
```

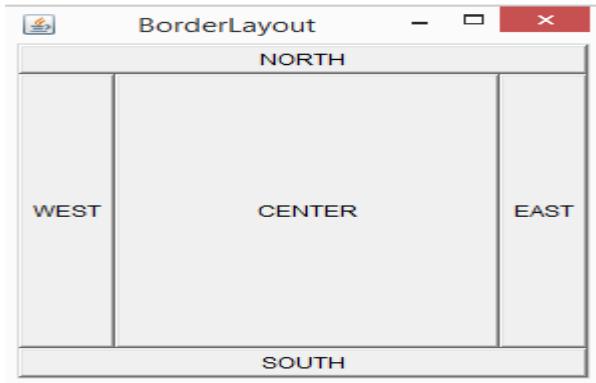
```
this.add("South",b3);
this.add("East",b4);
this.add("West",b5);
}
}
class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};
```



Example 2: project level reduce the length of the code.

```
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f = new Frame("BorderLayout");
        f.setVisible(true);
        f.setSize(300,300);
        f.setLayout(new BorderLayout());

        f.add(new Button("NORTH"),BorderLayout.NORTH);
        f.add(new Button("SOUTH"),BorderLayout.SOUTH);
        f.add(new Button("EAST"),BorderLayout.EAST);
        f.add(new Button("WEST"),BorderLayout.WEST);
        f.add(new Button("CENTER"),BorderLayout.CENTER);
    }
}
```



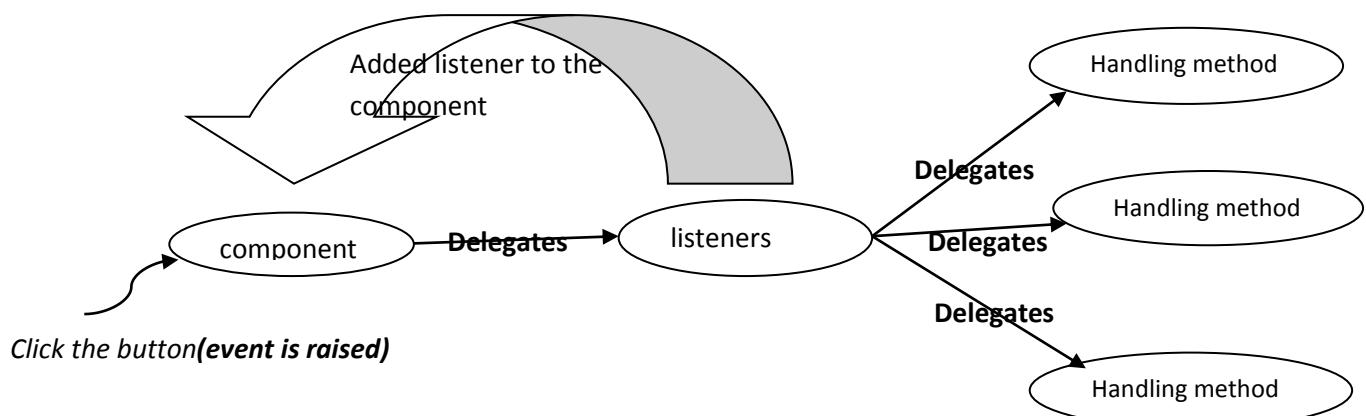
### Event delegation model:-

1. When we create a component the components visible on the screen but it is not possible to perform any action on that component because those components are by default static.  
**Example:** 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. Because frame is a static component so it is not possible to perform actions on the Frame.
2. To make static component into dynamic component we have to add some actions to the Frame. To attach these actions to the Frame component we need event delegation model.

**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. Wherever we are adding listeners to the component the component is able to respond based on the generated event.
3. **java.awt.event** package contains listeners and event classes for event handling.
4. 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”**

**ScrollBar:-**

1. By using ScrollBar we can move the Frame up and down.

ScrollBar s=new ScrollBar(int type)

Type of scrollbar

1. VERTICAL ScrollBar
2. HORIZONTAL ScrollBar

To create a HORIZONTAL ScrollBar:-

ScrollBar sb=new ScrollBar(ScrollBar.HORIZONTAL);

To get the current position of the scrollbar we have to use the following method.

public int getValue()

To create a VERTICAL ScrollBar:-

ScrollBar sb=new ScrollBar(ScrollBar.VERTICAL);

**Appropriate Listeners for Components:-**

| <b>GUI Component</b> | <b>Event Name</b> | <b>Listner Name</b> | <b>Lisener Methods</b>                                                                                                                                                                                                                                                                                                                   |
|----------------------|-------------------|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1.Frame              | Window Event      | Window Listener     | 1.Public Void WindowOpened(WindowEvent e)<br>2.Public Void WindowActivated(WindowEvent e)<br>3.Public Void WindowDeactivated(WindowEvent e)<br>4.Public Void WindowClosing(WindowEvent e)<br>5.Public Void WindowClosed(WindowEvent e)<br>6.Public Void WindowIconified(WindowEvent e)<br>7.Public Void WindowDeiconified(WindowEvent e) |
| 2.Textfield          | ActionEvent       | ActionListener      | Public Void Actionperformed(ActionEvent ae)                                                                                                                                                                                                                                                                                              |
| 3.TextArea           | ActionEvent       | ActionListener      | Public Void Actionperformed(ActionEvent ae)                                                                                                                                                                                                                                                                                              |
| 4.Menu               | ActionEvent       | ActionListener      | Public Void Actionperformed(ActionEvent ae)                                                                                                                                                                                                                                                                                              |
| 5.Button             | ActionEvent       | ActionListener      | Public Void Actionperformed(ActionEvent ae)                                                                                                                                                                                                                                                                                              |
| 6.Checkbox           | ItemEvent         | ItemListener        | Public Void ItemStatechanged(ItemEvent e)                                                                                                                                                                                                                                                                                                |
| 7.Radio              | ItemEvent         | ItemListener        | Public Void ItemStatechanged(ItemEvent e)                                                                                                                                                                                                                                                                                                |
| 8.List               | ItemEvent         | ItemListener        | Public Void ItemStatechanged(ItemEvent e)                                                                                                                                                                                                                                                                                                |
| 9.Choice             | ItemEvent         | ItemListener        | Public Void ItemStatechanged(ItemEvent e)                                                                                                                                                                                                                                                                                                |
| 10.Scrollbar         | AdjustmentEvent   | AdjustmentListener  | Public Void AdjustmentValueChanged<br>(AdjustementEvent e)                                                                                                                                                                                                                                                                               |
| 11.Mouse             | MouseEvent        | MouseListener       | 1.Public Void MouseEntered(MouseEvent e)<br>2.Public Void MouseExited(MouseEvent e)<br>3.Public Void MousePressed(MouseEvent e)<br>4.Public Void MouseReleased(MouseEvent e)<br>5.Public Void MouseClicked(MouseEvent e)                                                                                                                 |
| 12.Keyboard          | KeyEvent          | KeyListener         | 1.Public Void KeyTyped(KeyEvent e)<br>2.Public Void KeyPressed(KeyEvent e)<br>3.Public Void KeyReleased(KeyEvent e)                                                                                                                                                                                                                      |

**Event handling code:-**

It is possible to write the event handling code in following ways

5. In different class.
6. Same class
7. By using anonymous inner classes.

**Steps to perform event handling:-**

- 1) Prepare the required components.
- 2) Implements the listener interface override the methods to write the even handling code.
- 3) Add the listener to the component.

**Applying WindowListener on the Frame :-**

The Frame contains WindowLister it contains 7-methods listed below.

`windowActivated(WindowEvent e)`

Invoked when the Window is set to be the active Window.

`windowClosed(WindowEvent e)`

Invoked when a window has been closed as the result of calling dispose on the window.

`windowClosing(WindowEvent e)`

Invoked when the user attempts to close the window from the window's system menu.

`windowDeactivated(WindowEvent e)`

Invoked when a Window is no longer the active Window.

`windowIconified(WindowEvent e)`

Invoked when a window is changed from a minimized to a normal state.

`windowDeiconified(WindowEvent e)`

Invoked when a window is changed from a normal to a minimized state.

`windowOpened(WindowEvent e)`

Invoked the first time a window is made visible.

**In below example we are providing even handling code in separate class :-**

\*\*\*PROVIDING CLOSING OPTION TO THE FRAME\*\*\*

```
import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame
{
    MyFrame()
    {
        this.setSize(400,500); //here this keyword is optional because it is a current class
        this.setVisible(true);
        this.setTitle("myframe");
        this.setBackground(Color.green);
        this.addWindowListener(new myclassimpl());
    }
}
class myclassimpl implements WindowListener
{
    public void windowActivated(WindowEvent e)
    {
        System.out.println("window activated");
    }
    public void windowDeactivated(WindowEvent e)
    {
        System.out.println("window deactivated");
    }
}
```

```

public void windowIconified(WindowEvent e)
{
    System.out.println("window iconified");
}
public void windowDeiconified(WindowEvent e)
{
    System.out.println("window deiconified");
}
public void windowClosed(WindowEvent e)
{
    System.out.println("window closed");
}
public void windowClosing(WindowEvent e)
{
    System.out.println("window closing");
    System.exit(0);
}
public void windowOpened(WindowEvent e)
{
    System.out.println("window Opened");
}
};

class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};

```

**\*\*PROVIDING CLOSEING OPTION TO THE FRAME BY USING WINDOWADAPTOR CLASS \*\***

*In above example when our implements the windowListener interface then we must override all (7)the methods of WindowListener interface .but to close the frame we required only one method that is windowCloseing( ) method.*

*To overcome above limitation use WindowAdaptor class it contains all empty implementations of interface methods.*

**Note : if our class implements WindowListener interface we must override all the methods of windowListener interface . But if our class extends windowAdaptor class it is possible to override application required methods.**

```

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("MyFrame");
        this.addWindowListener(new Listenerimpl());
    }
};
class Listenerimpl extends WindowAdapter
{
    public void windowClosing(WindowEvent we)
    {
        System.exit(0);
    }
};

```

```

        }
};

class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
}

```

**Writing event handling code in anonymous inner class:-**

```

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("Myframe");
        this.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {
                System.exit(0);
            }
        });
    }
}
class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
}

```

\*\*\*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();
    }
};
```

\*\*\*\*\*CardLayout\*\*\*\*\*

```
import java.awt.*;
class MyFrame extends Frame
{
    MyFrame()
    {
        this.setSize(400,400);
        this.setVisible(true);
        this.setLayout(new CardLayout());
        Button b1=new Button("button1");
        Button b2=new Button("button2");
        Button b3=new Button("button3");
        Button b4=new Button("button4");
        Button b5=new Button("button5");
        this.add("First Card",b1);
        this.add("Second Card",b2);
        this.add("Thrid Card",b3);
        this.add("Fourth Card",b4);
        this.add("Fifth Card",b5);
    }
}
class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};
```

\*\*\*\*\*GRIDLAYOUT\*\*\*\*\*

```
import java.awt.*;
class MyFrame extends Frame
{
    MyFrame()
    {
        this.setVisible(true);
        this.setSize(500,500);
```

```

        this.setTitle("rattaiah");
        this.setBackground(Color.red);
        this.setLayout(new GridLayout(4,4));
        for (int i=0;i<10 ;i++ )
        {
            Button b=new Button(""+i);
            this.add(b);
        }
    }
};

class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};

```

**Example :-ACTIONLISTENER**

The below example we are performing addition and multiplications when we click add & mul buttons. So whenever we clicking button the button is able to listen the even to do this add the Listener to button.

The appropriate listener for button is **ActionListener**.

```

import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame implements ActionListener
{
    TextField tx1,tx2,tx3;
    Label l1,l2,l3;
    Button b1,b2;
    int result;
    MyFrame()
    {
        this.setSize(250,400);
        this.setVisible(true);
        this.setLayout(new FlowLayout());
        l1=new Label("First Value      :");
        l2=new Label("Second Value     :");
        l3=new Label("Result          :");

        tx1=new TextField(25);
        tx2=new TextField(25);
        tx3=new TextField(25);

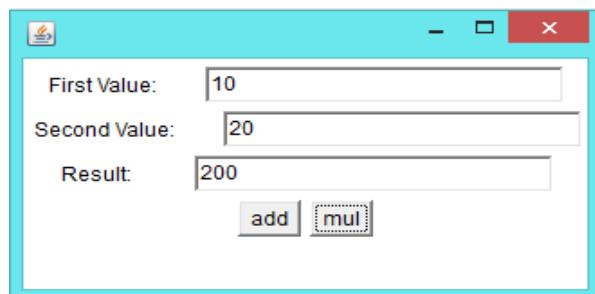
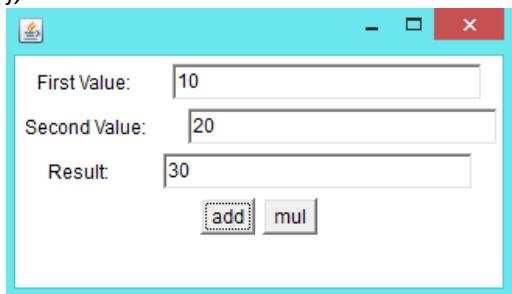
        b1=new Button("add");
        b2=new Button("mul");

        b1.addActionListener(this); // this represent current class object
        b2.addActionListener(this);
        this.add(l1);
        this.add(tx1);
        this.add(l2);

```

```
        this.add(tx2);
        this.add(l3);
        this.add(tx3);
        this.add(b1);
        this.add(b2);
    }
    public void actionPerformed(ActionEvent e)
    {
        try{
            int fval=Integer.parseInt(tx1.getText());
            int sval=Integer.parseInt(tx2.getText());
            String label=e.getActionCommand();
            if (label.equals("add"))
            {
                result=fval+sval;
            }
            if (label.equals("mul"))
            {
                result=fval*sval;
            }
            tx3.setText(""+result);
        }
        catch(Exception ee)
        {
            ee.printStackTrace();
        }
    }
};

class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
}
```



\*\*\*\*\* LOGIN STATUS \*\*\*\*\*

```
import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame implements ActionListener
{
    Label l1,l2;
    TextField tx1,tx2;
    Button b;
    String status="";
    MyFrame()
```

```
{      setVisible(true);
      setSize(400,400);
      setTitle("girls");
      setBackground(Color.red);
      l1=new Label("user name:");
      l2=new Label("password:");
      tx1=new TextField(25);
      tx2=new TextField(25);

      b=new Button("login");
      b.addActionListener(this);
      tx2.setEchoChar('*');

      this.setLayout(new FlowLayout());

      this.add(l1);
      this.add(tx1);
      this.add(l2);
      this.add(tx2);
      this.add(b);
}
public void actionPerformed(ActionEvent ae)
{
    String uname=tx1.getText();
    String upwd=tx2.getText();
    if (uname.equals("Sravya")&&upwd.equals("dss"))
    {
        status="login success";
    }
    else
    {
        status="login failure";
    }
    repaint();
}
public void paint(Graphics g)
{
    Font f=new Font("arial",Font.BOLD,30);
    g.setFont(f);
    this.setForeground(Color.green);
    g.drawString("Status:---"+status,50,300);
}
class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};
```

```
*****MENUITEMS*****  
  
import java.awt.*;  
import java.awt.event.*;  
class MyFrame extends Frame implements ActionListener  
{  
    String label="";  
    MenuBar mb;  
    Menu m1,m2,m3;  
    MenuItem mi1,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);  
  
        mi1=new MenuItem("open");  
        mi2=new MenuItem("save");  
        mi3=new MenuItem("saveas");  
  
        mi1.addActionListener(this);  
        mi2.addActionListener(this);  
        mi3.addActionListener(this);  
  
        m1.add(mi1);  
        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("Selected item....."+label,50,200);
}
}
class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};
```

---

**\*\*\*\*\*MOUSELISTENER INTERFACE\*\*\*\*\***

```
import java.awt.*;
import java.awt.event.*;
class myframe extends Frame implements MouseListener
{
    String[] msg=new String[5];
    myframe()
    {
        this.setSize(500,500);
        this.setVisible(true);
        this.addMouseListener(this);
    }
    public void mouseClicked(MouseEvent e)
    {
        msg[0]="mouse clicked.....("+e.getX()+","+e.getY()+")";
        repaint();
    }
    public void mousePressed(MouseEvent e)
    {
        msg[1]="mouse pressed.....("+e.getX()+","+e.getY()+")";
        repaint();
    }
    public void mouseReleased(MouseEvent e)
    {
        msg[2]="mouse released.....("+e.getX()+","+e.getY()+")";
        repaint();
    }
    public void mouseEntered(MouseEvent e)
    {
        msg[3]="mouse entered.....("+e.getX()+","+e.getY()+")";
        repaint();
    }
    public void mouseExited(MouseEvent e)
    {
        msg[4]="mouse exited.....("+e.getX()+","+e.getY()+")";
        repaint();
    }
    public void paint(Graphics g)
    {
        int X=50;
        int Y=100;
```

```
        for(int i=0;i<msg.length;i++)
        {
            if (msg[i]!=null)
            {
                g.drawString(msg[i],X,Y);
                Y=Y+50;
            }
        }
    };
class Demo
{
    public static void main(String[] args)
    {
        myframe f=new myframe();
    }
};
```

## \*\*\*\*\*ITEMLISTENER INTERFACE\*\*\*\*\*

```
import java.awt.*;
import java.awt.event.*;
class myframe extends Frame implements ItemListener
{
    String qual="",gen="";
    Label l1,l2;
    CheckboxGroup cg;
    Checkbox c1,c2,c3,c4,c5;
    Font f;
    myframe()
    {
        this.setSize(300,400);
        this.setVisible(true);
        this.setLayout(new FlowLayout());

        l1=new Label("Qualification: ");
        l2=new Label("Gender: ");

        c1=new Checkbox("BSC");
        c2=new Checkbox("BTECH");
        c3=new Checkbox("MCA");

        cg=new CheckboxGroup();
        c4=new Checkbox("Male",cg,false);
        c5=new Checkbox("Female",cg,true);

        c1.addItemListener(this);
        c2.addItemListener(this);
        c3.addItemListener(this);
        c4.addItemListener(this);
        c5.addItemListener(this);
    }
}
```

```

        this.add(l1);           this.add(c1);           this.add(c2);
        this.add(c3);           this.add(l2);           this.add(c4);
        this.add(c5);          

    }

    public void itemStateChanged(ItemEvent ie)
    {
        if(c1.getState()==true)
        {
            qual=qual+c1.getLabel()+",";
        }
        if(c2.getState()==true)
        {
            qual=qual+c2.getLabel()+",";
        }
        if(c3.getState()==true)
        {
            qual=qual+c3.getLabel()+",";
        }
        if(c4.getState()==true)
        {
            gen=c4.getLabel();
        }
        if(c5.getState()==true)
        {
            gen=c5.getLabel();
        }
        repaint();
    }

    public void paint(Graphics g)
    {
        Font f=new Font("arial",Font.BOLD,20);
        g.setFont(f);
        this.setForeground(Color.green);
        g.drawString("qualification----->" + qual,50,100);
        g.drawString("gender----->" + gen,50,150);
        qual="";
        gen="";
    }
}

class rc
{
    public static void main(String[] args)
    {
        myframe f=new myframe();
    }
};

*****KEYLISTENER INTERFACE*****
import java.awt.*;
import java.awt.event.*;
class myframe extends Frame
{
    myframe()
    {
        this.setSize(400,400);
        this.setVisible(true);
        this.setBackground(Color.green);
        this.addKeyListener(new keyboardimpl());
    }
}

```

```

};

class keyboardimpl implements KeyListener
{
    public void keyTyped(KeyEvent e)
    {
        System.out.println("key typed "+e.getKeyChar());
    }
    public void keyPressed(KeyEvent e)
    {
        System.out.println("key pressed "+e.getKeyChar());
    }
    public void keyReleased(KeyEvent e)
    {
        System.out.println("key released "+e.getKeyChar());
    }
}
class Demo
{
    public static void main(String[] args)
    {
        myframe f=new myframe();
    }
};
*****CHECK LIST AND CHOICE*****
import java.awt.*;
import java.awt.event.*;
class myframe extends Frame implements ItemListener
{
    Label l1,l2;
    List l;
    Choice ch;
    String[] tech;
    String city="";
    myframe()
    {
        this.setSize(300,400);
        this.setVisible(true);
        this.setLayout(new FlowLayout());

        l1=new Label("Technologies: ");
        l2=new Label("City: ");

        l=new List(3,true);
        l.add("c");           l.add("c++");          l.add("java");
        l.addItemListener(this);

        ch=new Choice();
        ch.add("hyd");         ch.add("chenni");       ch.add("Banglore");
        ch.addItemListener(this);

        this.add(l1);          this.add(l);            this.add(l2);          this.add(ch);
    }
    public void itemStateChanged(ItemEvent ie)
    {
        tech=l.getSelectedItems();
        city=ch.getSelectedItem();
        repaint();
    }
}

```

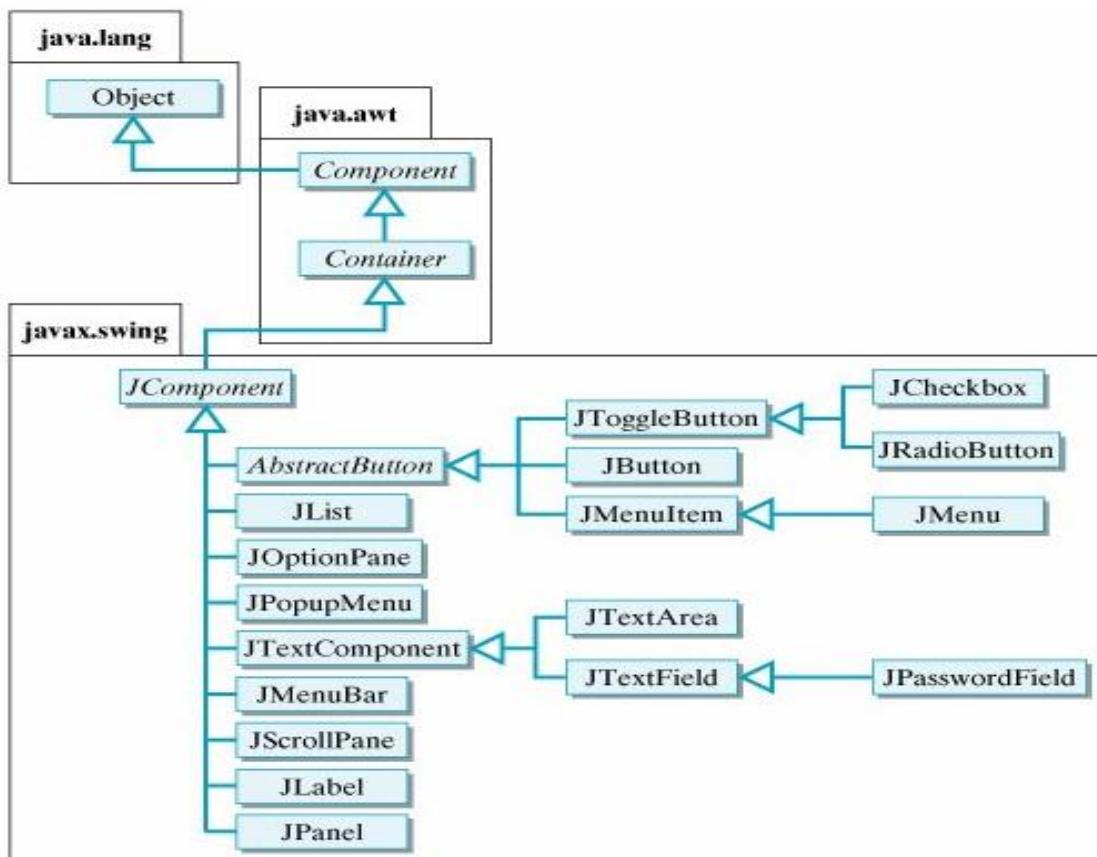
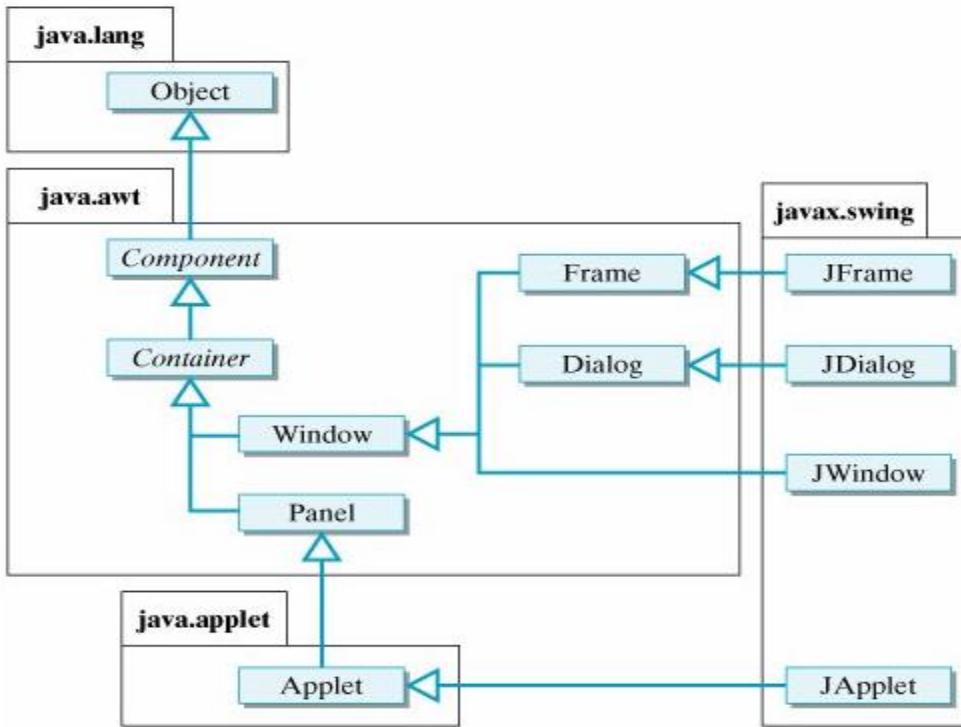
```
        }
    public void paint(Graphics g)
    {
        Font f=new Font("arial",Font.BOLD,20);
        g.setFont(f);
        String utech="";
        for(int i=0;i<tech.length ;i++ )
        {
            utech=utech+tech[i]+" ";
        }
        g.drawString("tech:-----"+utech,50,200);
        g.drawString("city-----"+city,50,300);
        utech="";
    }
}
class Demo
{
    public static void main(String[] args)
    {
        myframe f=new myframe();
    }
};
*****AdjustmentListener*****
import java.awt.*;
import java.awt.event.*;
class myframe extends Frame implements AdjustmentListener
{
    Scrollbar sb;
    int position;
    myframe()
    {
        this.setSize(400,400);
        this.setVisible(true);
        this.setLayout(new BorderLayout());
        sb=new Scrollbar(Scrollbar.VERTICAL);
        this.add("East",sb);
        sb.addAdjustmentListener(this);
    }
    public void adjustmentValueChanged(AdjustmentEvent e)
    {
        position=sb.getValue();
    }
    public void paint(Graphics g)
    {
        g.drawString("position:"+position,100,200);
        repaint();
    }
}
class scrollbarex
{
    public static void main(String[] args)
    {
        myframe f=new myframe();
    }
};
```

**SWINGS**

1. *Sun Micro Systems introduced AWT to prepare GUI applications but awt components not satisfy the client requirement.*
2. *An alternative to AWT Netscape Communication Corporation has provided set of GUI components in the form of IFC(Internet Foundation Class) but IFC also provide less performance and it is not satisfy the client requirement.*
3. *In the above context[sun & Netscape] combine and introduced common product to design GUI applications is called JFS(java foundation classes) then it is renamed as swings.*

**Differences between awt and Swings:**

- ✓ AWT components are platform dependent but Swings components are platform independent because these components are completely written in java .
- ✓ AWT components are heavyweight component but swing components are light weight component.
- ✓ AWT components consume more number of system resources Swings consume less number of system resources.
- ✓ AWT is provided less number of components whereas a swing provides more number of components.
- ✓ AWT doesn't provide Tooltip Test support but swing components have provided Tooltip test support.
- ✓ In awt to close the window :      windowListener      windowAdaptor  
In case of swing use small piece of code.  
`f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`
- ✓ 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.
- ✓ The awt classes & interfaces are present in java.awt packages & swing classes are present in javax.swing package.



**Example :-**

```
package swingss;

import java.awt.Color;
import java.awt.FlowLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
public class MyFrame extends JFrame{
    public MyFrame() {
        setVisible(true);
        setSize(300, 300);
        setTitle("MyFrame");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        JLabel label = new JLabel("Hello World:");
        JButton button = new JButton("click me");
        add(label);
        add(button);
    }
    public static void main(String[] args) {
        MyFrame f = new MyFrame();
    }
}
```

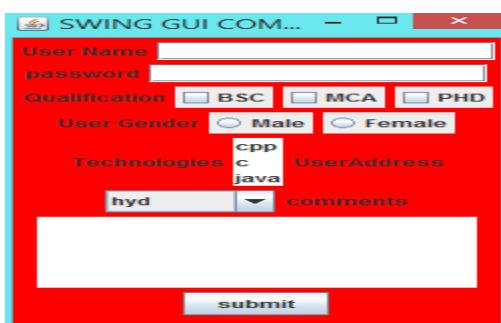
**Example 2:-**

```
package swingss;
import java.awt.*;
import javax.swing.*;
class Test2 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;
    Test2()
    {
        setVisible(true);
        setSize(150,500);
        setTitle("SWING GUI COMPONENTS EXAMPLE");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        getContentPane().setBackground(Color.red);
    }
}
```

```

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");
add(l1);
add(tf);      add(l2);      add(pf);
add(l3);      add(cb1);      add(cb2);      add(cb3);
add(l4);      add(rb1);      add(rb2);      add(l5);
add(l);       add(l6);       add(cb);       add(l7);
add(ta);      add(b);
}
public static void main(String[] args)
{
    Test2 f=new Test2();
}
};


```



**Example 3:**

```
package swingss;

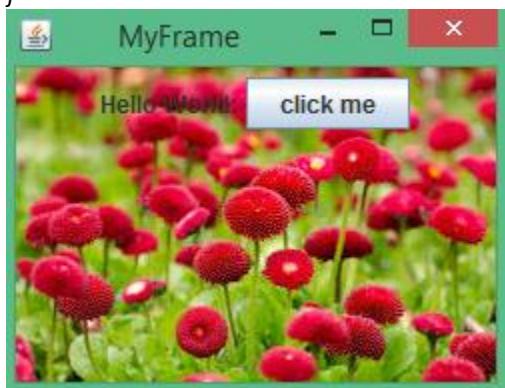
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;

import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MyFrame extends JFrame{
    public MyFrame() {
        setVisible(true);
        setSize(200, 200);
        setTitle("MyFrame");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new BorderLayout()); //setting layout to frame

        JLabel label = new JLabel(new ImageIcon("F:\\corejava images\\download.jpg"));
        add(label);
        label.setLayout(new FlowLayout()); // setting layout to back ground image

        JLabel label2 = new JLabel("Hello World:");
        JButton button = new JButton("click me");
        label.add(label2);
        label.add(button);
    }
    public static void main(String[] args) {
        MyFrame f = new MyFrame();
    }
}
```



**Example 4:-**

```
package swingss;

import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JRadioButton;
public class MyFrame2 extends JFrame implements ActionListener {
    JRadioButton button1,button2;
    JButton button;
    public MyFrame2() {
        setVisible(true);
        setTitle("MyFrame");
        setSize(300, 300);
        setLayout(new FlowLayout());

        button1 = new JRadioButton("Male");
        button2 = new JRadioButton("Female");
        button = new JButton("click");

        ButtonGroup group = new ButtonGroup();
        group.add(button1);
        group.add(button2);
        add(button1);           add(button2);           add(button);
        button.addActionListener(this);
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        if(button1.isSelected())
        {   JOptionPane.showMessageDialog(this, "U r selected Male option thankyou");}
        if(button2.isSelected())
        {   JOptionPane.showMessageDialog(this, "U r selected Female option thankyou");}
    }
    public static void main(String[] args) {
        MyFrame2 frame2 = new MyFrame2();
    }
}
```



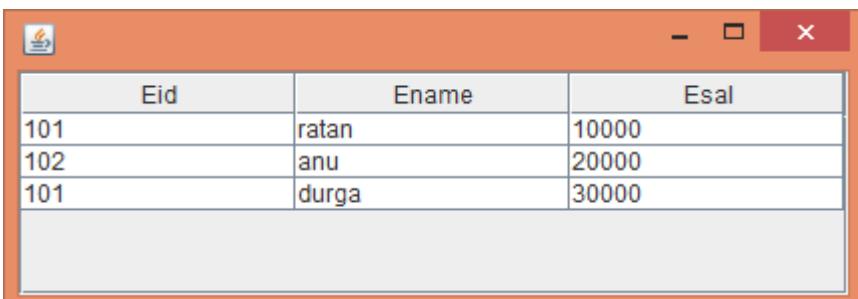
*Example 5:*

```
package swingss;

import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
public class MyFrame3 extends JFrame {
    public MyFrame3() {
        setVisible(true);
        setSize(400, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        String[][] data={{"101","ratan","10000"}, {"102","anu","20000"}, {"101","durga","30000"}};
        String[] col={"Eid", "Ename", "Esal"};
        JTable jTable = new JTable(data,col);
        add(jTable);

        JScrollPane jScrollPane = new JScrollPane(jTable);
        add(jScrollPane);
    }
    public static void main(String[] args)
    {
        MyFrame3 frame3 = new MyFrame3();
    }
}
```



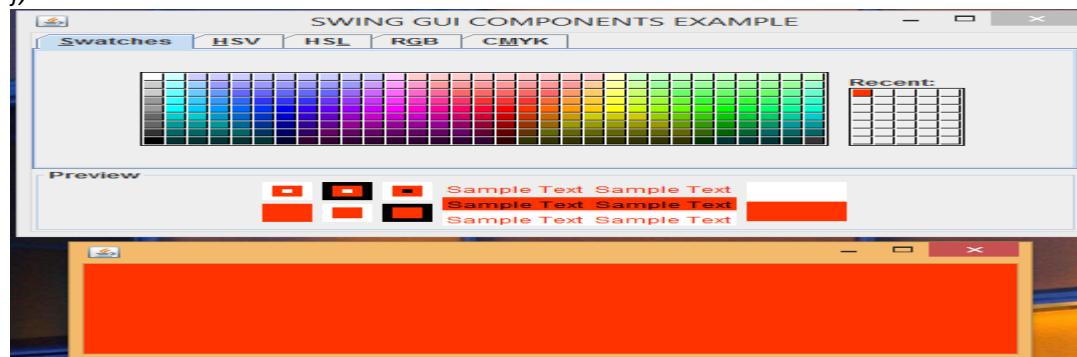
## ***Application 6: - JCOLORCHOOSEN***

```
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();
    }
}

```



#### **Application 7: JFILECHOOSEN**

```

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=getContentPane();
        l=new JLabel("Select File:");

```

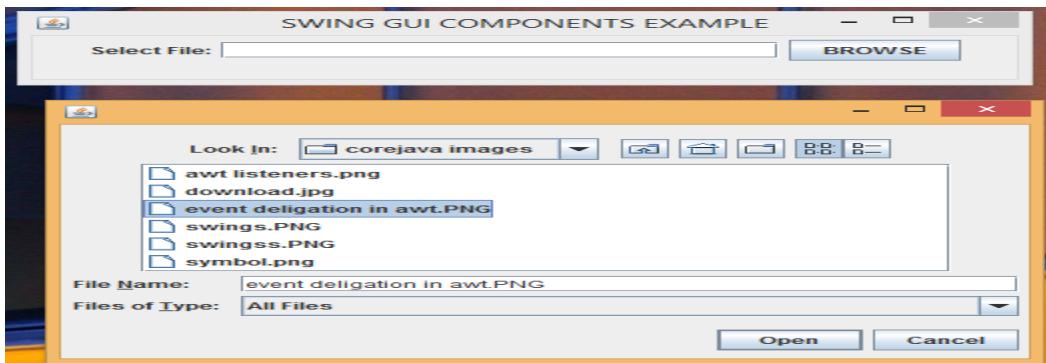
```
tf=new JTextField(25);
b=new JButton("BROWSE");
this.setLayout(new FlowLayout());
b.addActionListener(this);
c.add(l);           c.add(tf);           c.add(b);
}

public void actionPerformed(ActionEvent ae)
{
    class FileChooserDemo extends JFrame implements ActionListener
    {
        FileChooserDemo()
        {
            Container c=getContentPane();
            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();
    }
}
```



### Java.awt.Applet

- ✓ The applet is runs on browser window to display the dynamic content on browser window.
- ✓ The applet does not contains main methods to start the execution but it contains life cycle methods these methods are automatically called by web browser.
- ✓ To run the applet in browser window we need to install plugin in.

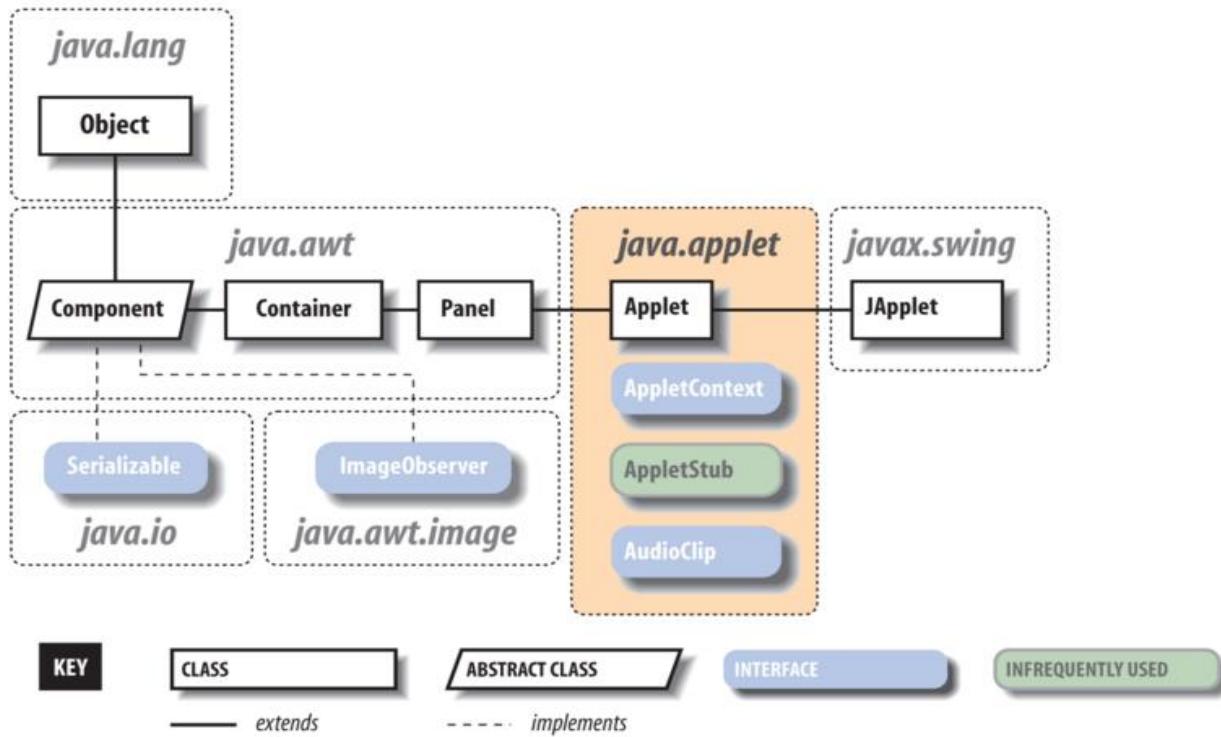


Figure 23-1. The `java.applet` package

The applet contains four life cycle methods

- a. `Init()`
- b. `Start()`
- c. `Stop()`
- d. `Destroy()`

These methods are called by applet viewer or web browser

`Init()` : used to initialize the applet and it called only once.

`Start()` : it invoked after `init()` method to start the applet then the applet become visible.

`Stop()` : it is used to stop the applet then the applet become invisible.

`Destroy()`: it called after `stop` method it gives to applet last chance to cleanup.

`Java.awt.Component` class provide one life cycle method of applet

`Public void paint(Graphics g)`

The above method used to paint the applet to print the data in applet.

**Steps to design the application:-**

1. Create the applet by extending **Applet** class.
2. Then configure the applet in html code.

**Applet First Application:-****Test.java:-**

```
import java.applet.Applet;
import java.awt.*;
public class Test extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello, world this is applet first example!", 150, 150);
    }
}
```

**Firstapplet.html**

```
<html>
<body>
    <applet code="Test.class" height="300" width="300"/>
</body>
</html>
```

**Execution : appletviewer Firstapplet.html**

**Running applet:-**

It is possible to run the applet in two ways

- 1) By using html file

Configure the applet in html file then open the html file in browser window.  
Click on **Firstapplet.html** then the applet is displayed on browser window.

- 2) By using applet viewer tool

Run the application by using below command

**Appletviewer Firstapplet.html**

**Application 2:***Test.java:-*

```

import java.awt.*;
import java.applet.*;
public class Test extends Applet
{
    String msg="";
    public void paint(Graphics g)
    {
        Font f=new Font("arial",Font.BOLD,20);
        g.setFont(f);
        g.drawString("Durga Software Solutions "+msg,100,200);
    }
    public void init()
    {
        msg=msg+"initialization"+ " ";
        System.out.println("init()");
    }
    public void start()
    {
        msg=msg+"starting"+ " ";
        System.out.println("start()");
    }
    public void stop()
    {
        msg=msg+"stoping";
        System.out.println("stop()");
    }
    public void destroyed()
    {
        msg=msg+"destroyed";
        System.out.println("destroy()");
    }
};

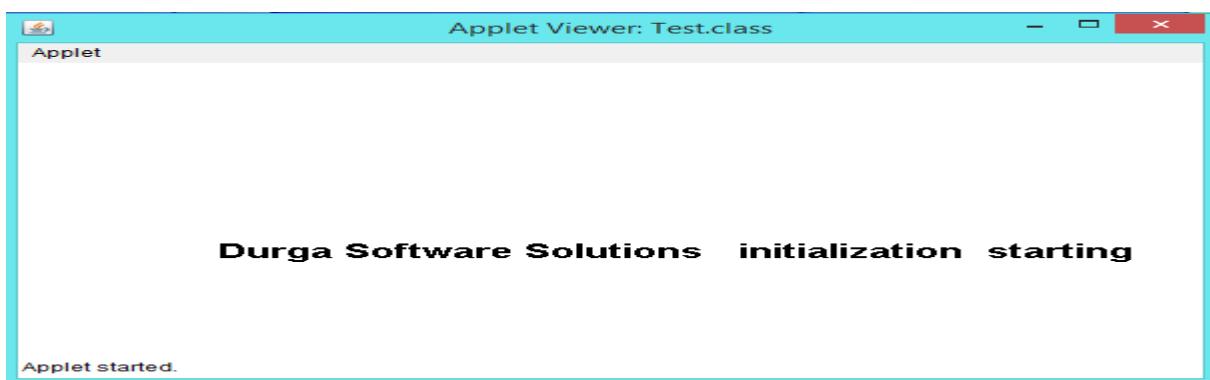
```

**Configuration of Applet:-**

```

<html>
<applet code="Test.class" width="500" height="500">
</applet>
</html>

```

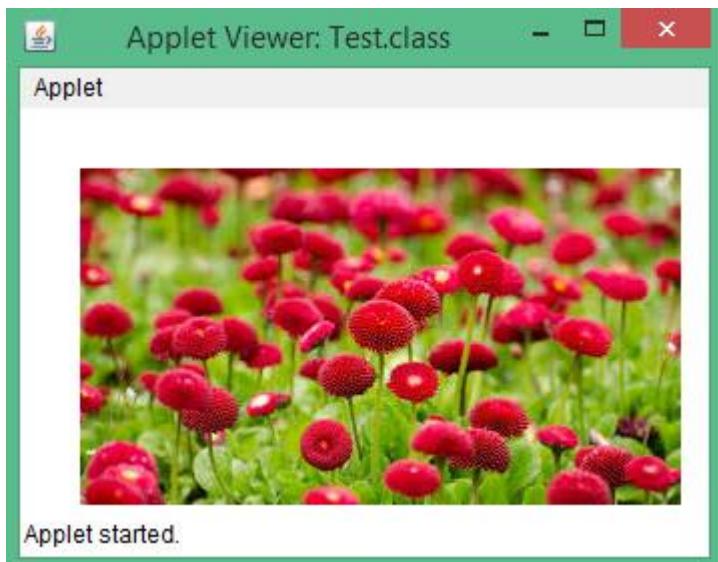
**Execution : appletviewer Firstapplet.html**

**Application 3: displaying image on applet.****Test.java:**

```
import java.awt.*;
import java.applet.*;
public class Test extends Applet {
    Image picture;
    public void init() {
        picture = getImage(getDocumentBase(),"flower.jpg");
    }
    public void paint(Graphics g) {
        g.drawImage(picture, 30,30, this);
    }
}
```

**Firstapplet.html**

```
<html>
<body>
<applet code="Test.class" height="300" width="300"></applet>
</body>
</html>
```

**Execution : appletviewer Firstapplet.html**

## INTERNATIONALIZATION (i18N)

**i18N enables the application to support in different languages.**

- Internationalization is also called as i18n because in between I & n 18 characters are present.
- By using Locale class and ResourceBundle class we are enable I18n on the application.
- Local is nothing but language + country.
- For making your application to support I18n we need to prepare local specific properties file it means for English one properties file & hindi one properties file ...etc.
- The property file format is key = value
- The properties file name followed pattern bundlename with language code and country code.
  - ApplicationMessages\_en\_US.properties.
- In single web application contains different properties file all the properties files key must be same and values are changed local to Locale.

### Java.util.Locale:-

- Locale Object is decide properties file based on argument you passed and then it display locale specific details based on Properties file entry.

```
Locale l = new Locale(args[0],args[1]);
```

```
Locale l = new Locale(en,US);
```

D:\5batch>javap java.util.Locale

Compiled from "Locale.java"

```
public final class java.util.Locale extends java.lang.Object {
    public static final java.util.Locale ENGLISH;
    public static final java.util.Locale FRENCH;
    public static final java.util.Locale GERMAN;
    public static final java.util.Locale ITALIAN;
    public static final java.util.Locale JAPANESE;
    public static final java.util.Locale KOREAN;
    public static final java.util.Locale CHINESE;
    public static final java.util.Locale SIMPLIFIED_CHINESE;
    public static final java.util.Locale TRADITIONAL_CHINESE;
    public static final java.util.Locale FRANCE;
    public static final java.util.Locale GERMANY;
    public static final java.util.Locale ITALY;
    public static final java.util.Locale JAPAN;
    public static final java.util.Locale KOREA;
    public static final java.util.Locale CHINA;
    public static final java.util.Locale PRC;
    public static final java.util.Locale TAIWAN;
    public static final java.util.Locale UK;
    public static final java.util.Locale US;
```

```
public static final java.util.Locale CANADA;
public static final java.util.Locale CANADA_FRENCH;
```

#### Sample Language Codes

Language Code	Description
de	German
en	English
fr	French
ru	Russian
ja	Japanese
јv	Javanese
ko	Korean
zh	Chinese

#### To get particular language and country code use following example:-

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Locale l = Locale.FRANCE;
        System.out.println(l.getLanguage());
        System.out.println(l.getCountry());
    }
}
```

D:\5batch>java Test

fr

FR

#### Java.util.ResourceBundle:-

Creates ResourceBundle object by passing Local object then by using ResourceBoundle we are able to get data form properties file that is decide by Locale.

- It is possible to create ResourceBundle Object without specifying Locale it will take default properties file with default language.

**ResourceBundle bundle1 = ResourceBundle.getBundle("Application");**

- It is possible to create ResourceBundle Object by specifying default Locale object.

**ResourceBundle bundle2 = ResourceBundle.getBundle("Application",Locale.FRANCE);**

- It is possible to create ResourceBundle object by creating new user Locale Object

**ResourceBundle bundle3 = ResourceBundle.getBundle("Application",new Locale("ratan","RATAN"));**

**Application 1:-****Steps to design application:-****Step-1:- prepare properties files to support different languages and countries.**

Application.properties	default properties file(base properties file)
Application_fr_FR.properties	French properties file
Alliction_ratan_RATAN.properties	Ratan country properties file

**Step 2:- create locale object it identified particular language and country and it decides execution of properties file.**`Locale l = new Locale("en","US");`**The above statement specify language is English and country united states**`Locale l = new Locale("fr","CA");``Locale x = new Locale("fr","FR");`**The above two locales specifies France language in Canada & France***Instead of hard coding language name and country name get the values from command prompt at runtime.*

```
Public static void main(String[ ] args)
{
    Locale l = new Locale(args[0],args[1]);
}
```

`D:\5batch>java Test fr FR`**Step 3:-create ResourceBundle by passing Locale object.****//if no local is Matched this property file is executed [default property file]**`ResourceBundle bundle1 = ResourceBundle.getBundle("Application");`**//it create ResourceBundle with local that is already defined [France properties file ]**`ResourceBundle bundle2 = ResourceBundle.getBundle("Application",Locale.FRANCE);`**Step 4:- fetch the text form ResourceBundle**

```
String msg = Bundle.getString("wish");
System.out.println(msg);
```

**Application.properties:-**`countryname = USA``lang = eng`**Application\_fr\_FR.properties:-**`countryname = canada``lang = france`**Alliction\_ratan\_RATAN.properties:-**`countryname=Ratan``lang= ratan`

**Test.java:-**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
//if no local is Matched this property file is executed
        ResourceBundle bundle1 = ResourceBundle.getBundle("Application");
//it create ResourceBundle with local that is already defined
        Locale l1 = Locale.FRANCE;
        ResourceBundle bundle2 = ResourceBundle.getBundle("Application",l1);
//it creates ResourceBundle with new user created Locale
        Locale l2 = new Locale("ratan", "RATAN")
        ResourceBundle bundle3 = ResourceBundle.getBundle("Application",l2);
        System.out.println(bundle1.getString("countryname")+"--"+bundle1.getString("lang"));
        System.out.println(bundle2.getString("countryname")+"--"+bundle2.getString("lang"));
        System.out.println(bundle3.getString("countryname")+"--"+bundle3.getString("lang"));
    }
}

```

**Output:-**

D:\5batch>java Test

USA--eng

Canada--france

Ratan--Ratan

**APPLICATION 2:-**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
//creates local object with the help of arguments
        Locale l = new Locale(args[0],args[1]);
//it creates resource bundle with local passed from as command line arguments
        ResourceBundle bundle = ResourceBundle.getBundle("Application",l);
        System.out.println(bundle.getString("countryname"));
        System.out.println(bundle.getString("lang"));
    }
}

```

D:\5batch>java Test x y

USA

eng

D:\5batch>java Test fr FR

canada

france

D:\5batch>java Test ratan RATAN

Ratan

ratan

### Application before internationalization:-

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        System.out.println("hello");
        System.out.println("i like you");
        System.out.println("i hate you");
    }
}
```

We are decide to print this messages in different languages like Germany, French.....etc then we must translate the code in different languages by moving the message out of source code to text file it looks the program need to be internationalized(supporting different languages).

### Application.properties:-

```
wish = hello
lovely = i love you
angry = i hate you
```

### Application\_fr\_FR.properties:-

```
wish = hlloe
lovely = i evol you
angry = i etah you
```

### Application\_hi\_IN.properties:-

```
wish=\u0c39\u0c46\u0c32\u0c4d\u0c32\u0c4a
lovely=\u0c07 \u0c32\u0c4a\u0c35\u0c46 \u0c2f\u0c4a\u0c09
angry=\u0c07 \u0c39\u0c24\u0c46 \u0c09
```

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Locale l = new Locale(args[0],args[1]);
        ResourceBundle rb = ResourceBundle.getBundle("Application",l);
        System.out.println(rb.getString("wish"));
        System.out.println(rb.getString("lovely"));
        System.out.println(rb.getString("angry"));
    }
}
```

D:\5batch>java Test x y  
 hello  
 i love you  
 i hate you

D:\5batch>java Test fr FR  
 hlloe  
 i evol you  
 i etah you  
 D:\5batch>java Test hi IN  
 ??????

? ???? ???

? ??? ?

### Conversion of any language to Unicode values:-

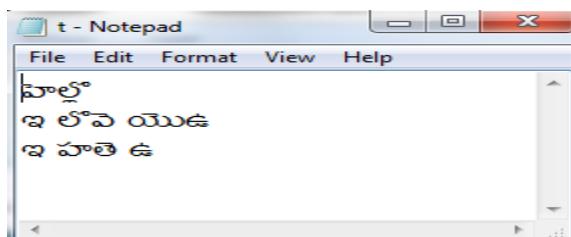
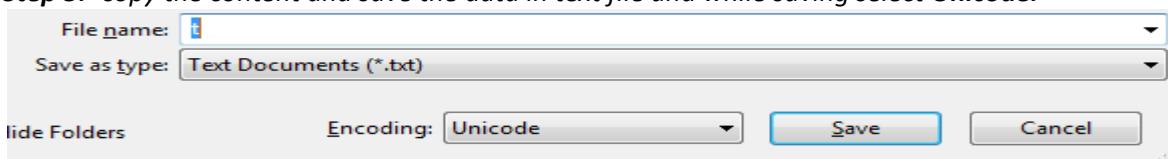
Step 1:- download Unicode editor from internet [www.higopi.com](http://www.higopi.com)

#### [Converters Link](#)

Above converter can also be downloaded and used offline from [here](#)

Step 2:- unzip the file and click on index.html page select language and type the words.

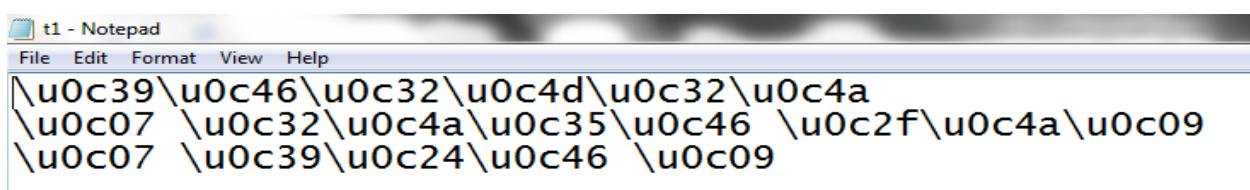
Step 3:- copy the content and save the data in text file and while saving select **Unicode**.



Step 4:- convert the above language to Unicode character format.

Syntax:- **native2ascii -encoding encoding-name source-file destination-file**

D:\>native2ascii -encoding unicode t.txt output.txt



**Application :-****Application.properties:-**

```
wish = hello
lovely = i love you
angry = i hate you
```

**Application\_fr\_FR.properties:-**

```
wish = hlloie
lovely = i evol you
angry = i etah you
```

**Application\_t1\_IN.properties:-**

```
wish=\u0c39\u0c46\u0c32\u0c4d\u0c32\u0c4a
lovely=\u0c07 \u0c32\u0c4a\u0c35\u0c46 \u0c2f\u0c4a\u0c09
angry=\u0c07 \u0c39\u0c24\u0c46 \u0c09
```

**Test.java:-**

```
import java.util.*;
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Locale l = new Locale(args[0],args[1]);
        ResourceBundle b = ResourceBundle.getBundle("Application",l);
        Frame f = new Frame();          //to create frame
        f.setVisible(true);            //to provide visibility to frame
        f.setSize(300,75);           //to align the frame set bounds
        f.setLayout(new FlowLayout()); //to set the frame proper format
        //creation of buttons with labels
        Button b1 = new Button(b.getString("wish"));
        Button b2 = new Button(b.getString("lovely"));
        Button b3 = new Button(b.getString("angry"));
        //adding buttons into frame
        f.add(b1);
        f.add(b2);
        f.add(b3);
    }
}
```

D:\5batch>java Test t1 IN





Test.java:- example

```
import java.util.*;
public class Test {
    static public void main(String[] args) {
        String language;
        String country;
        Locale currentLocale;
        ResourceBundle messages;
        if (args.length != 2)
            {   language = new String("en");
                country = new String("US");
            }
            else
            {   language = new String(args[0]);
                country = new String(args[1]);
            }
        currentLocale = new Locale(language, country);
        messages = ResourceBundle.getBundle("Application", currentLocale);
        System.out.println(messages.getString("wish"));
        System.out.println(messages.getString("lovely"));
        System.out.println(messages.getString("angry"));
    }
}
```

D:\5batch>java Test

hello

i love you

i hate you

D:\5batch>java Test x y

hello

i love you

*i hate you  
D:\5batch>java Test tl IN*

*??????*

*? ???? ???*

*? ??? ?*

*D:\5batch>java Test fr FR*

*hlooe*

*i evol you*

*i etah you*

**Example :- display Date in different Locale.**

***DateFormat.DEFAULT,***

***DateFormat.SHORT,***

***DateFormat.MEDIUM,***

***DateFormat.LONG,***

***DateFormat.FULL***

Sample Date Formats

Style	U.S. Locale	French Locale
DEFAULT	Jun 30, 2009	30 juin 2009
SHORT	6/30/09	30/06/09
MEDIUM	Jun 30, 2009	30 juin 2009
LONG	June 30, 2009	30 juin 2009
FULL	Tuesday, June 30, 2009	mardi 30 juin 2009

**Test.java:-**

```
import java.util.*;
import java.text.DateFormat;
class Test
{
    public static void main(String[] args)
    {
        Date d = new Date();
        //default locale en US
        DateFormat df1 = DateFormat.getDateInstance(DateFormat.DEFAULT,Locale.getDefault());
        System.out.println(df1.format(d));
        //date of fresh
        DateFormat df2 = DateFormat.getDateInstance(DateFormat.MEDIUM,Locale.FRENCH);
        System.out.println(df2.format(d));
        //date of Italy
        DateFormat df3 = DateFormat.getDateInstance(DateFormat.SHORT,Locale.ITALY);
        System.out.println(df3.format(d));
    }
};
```

**Example on time format:-**

## Sample Time Formats

Style	U.S. Locale	German Locale
DEFAULT	7:03:47 AM	7:03:47
SHORT	7:03 AM	07:03
MEDIUM	7:03:47 AM	07:03:07
LONG	7:03:47 AM PDT	07:03:45 PDT
FULL	7:03:47 AM PDT	7.03 Uhr PDT

```
import java.util.*;
import java.text.*;
class Test
{
    public static void main(String[] args)
    {Date d = new Date();
    DateFormat df1 = DateFormat.getTimeInstance(DateFormat.DEFAULT,Locale.getDefault());
    System.out.println(df1.format(d));
    DateFormat df2 = DateFormat.getTimeInstance(DateFormat.MEDIUM,Locale.FRENCH);
    System.out.println(df2.format(d));
    DateFormat df3 = DateFormat.getTimeInstance(DateFormat.SHORT,Locale.ITALY);
    System.out.println(df3.format(d));
    }
};


```

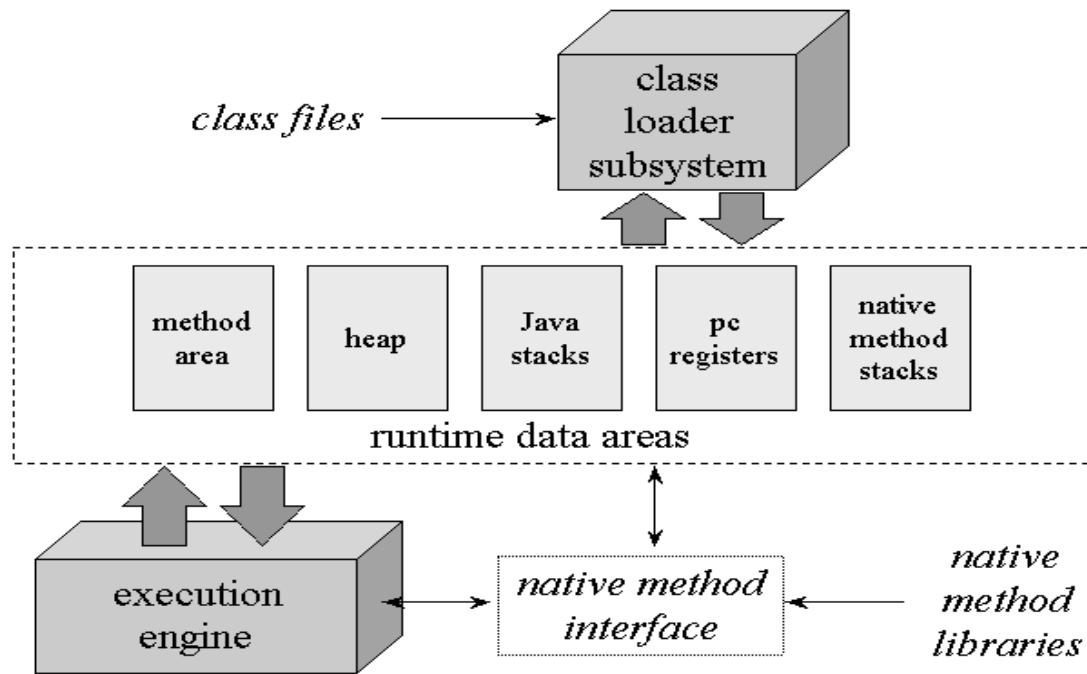
Example on both data and Time format:-

## Sample Date and Time Formats

Style	U.S. Locale	French Locale
DEFAULT	Jun 30, 2009 7:03:47 AM	30 juin 2009 07:03:47
SHORT	6/30/09 7:03 AM	30/06/09 07:03
MEDIUM	Jun 30, 2009 7:03:47 AM	30 juin 2009 07:03:47
LONG	June 30, 2009 7:03:47 AM PDT	30 juin 2009 07:03:47 PDT
FULL	Tuesday, June 30, 2009 7:03:47 AM PDT	mardi 30 juin 2009 07 h 03 PDT

```
import java.util.*;
import java.text.*;
class Test
{
    public static void main(String[] args)
    {Date d = new Date();
    DateFormat df1 = DateFormat.getDateTimeInstance(DateFormat.FULL,DateFormat.FULL,Locale.getDefault());
    System.out.println(df1.format(d));
    DateFormat df2 = DateFormat.getDateTimeInstance(DateFormat.FULL,DateFormat.FULL,Locale.FRENCH);
    System.out.println(df2.format(d));
    }
};


```

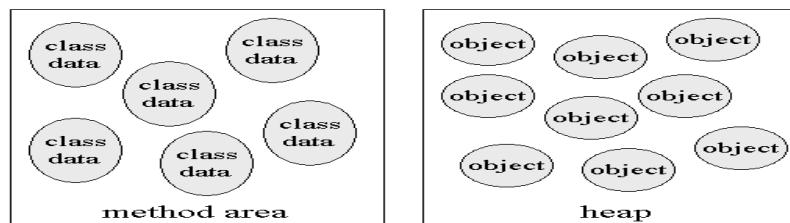
JVM Architecture:-Class loader subsystem:-

1. It is used to load the classes and interfaces.
2. It verifies the byte code instructions.
3. It allots the memory required for the program.

**Runtime data area:-this is the memory resource used by the JVM and it is 5 types**

**Method Area:-**It is used to store the class data and method data.

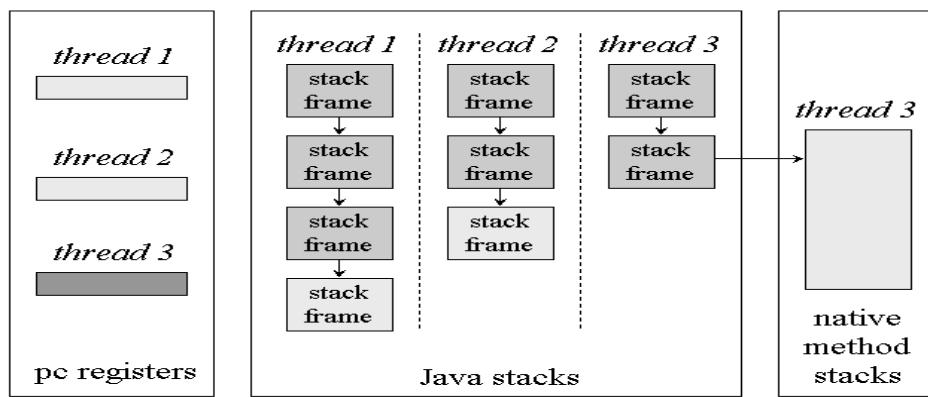
**Heap area:-**It is used to store the Objects.



**Runtime data areas shared among all threads.**

Java stacks:-

- Whenever new thread is created for each and every new thread the JVM will creates PC(program counter) register and stack.
- If a thread executing java method the value of pc register indicates the next instruction to execute.
- Stack will stores method invocations of every thread. The java method invocation includes local variables and return values and intermediate calculations.
- The each and every method entry will be stored in stack. And the stack contains group of entries and each and every entry stored in one stack frame hence stack is group of stack frames.
- Whenever the method completes the entry is automatically deleted from the stack so whatever the functionalities declared in method it is applicable only for respective methods.
- Java native method stack is used to store the native methods invocations.



**Runtime data areas exclusive to each thread.**

#### **Native method interface:-**

Native method interface is a program that connects native methods libraries (C header files) with JVM for executing native methods.

**Native method library:** It contains native libraries information.

#### **Execution engine:-**

It is used to execute the instructions available in the methods of loaded classes. It contains JIT(just in time compiler) and interpreter used to convert byte code instructions into machine understandable code.

**Java Class declaration interview questions**

- 1) What is present version of java and initial version of java?
- 2) How many modifiers in java and how many keywords in java?
- 3) What is initial name of java and present name of java?
- 4) What do you mean by open source software & java is open source software or not?
- 5) What are dependent languages and technologies on java?
- 6) What do you mean by licensed version and what are the licensed softwares?
- 7) Can we have multiple public classes in single source file?
- 8) What do you mean by platform dependent & platform independent ? java is which type?
- 9) What do you mean by main class & it is possible to declare multiple main classes in single source file or not?
- 10) What do you mean by token and literal?
- 11) What do you mean by identifier?
- 12) Can we create multiple objects for single class?
- 13) Is it possible to declare multiple public classes in single source file?
- 14) What is the difference between editor and IDE(integrated development environment)
- 15) Write the examples of editor and IDE?
- 16) Define a class & object?
- 17) What are the coding conventions of classes and interfaces?
- 18) What are the coding conventions of methods and variables?
- 19) Is java object oriented programming language?
- 20) In java program starts from which method and who is calling that method?
- 21) What are the commands required for compilation and execution?
- 22) Is it possible to compile how many files at a time & is it possible to execute how many classes at a time?
- 23) The compiler understandable file format & JVM understandable file format?
- 24) What is the difference between JRE and JDK?
- 25) What is the difference between path and class path?
- 26) What is the purpose of environmental variables setup?
- 27) What are operations done at compilation time and execution time?
- 28) What is the purpose of JVM?
- 29) JVM platform dependent or independent?
- 30) Is it possible to provide multiple spaces in between two tokens?
- 31) Class contains how many elements what are those?
- 32) Who is generating .class file and .class files generation is based on?
- 33) What is .class file contains & .java contains?
- 34) What is the purpose of data types and how many data types are present in java?
- 35) Who is assigning default values to variables?
- 36) What is the default value of int, char, Boolean, double?
- 37) Is null a keyword or not?
- 38) What is the default values for objects
- 39) Can I have multiple main methods in single class?
- 40) What is the default package in java?
- 41) Can I import same class twice yes→what happened no →why ?
- 42) Is empty java source file is valid or not?
- 43) What is the purpose of variables in java?
- 44) How many types of variables in java and what are those variables?
- 45) What is the life time of static variables and where these variables are stored?

- 46) What is the life time of instance variables and where these variables are stored?
- 47) What is the life time of local variables and where these variables are stored?
- 48) For the static members when memory is allocated?
- 49) Where we declared local variables & instance variables & static variables
- 50) For the instance members when memory is allocated?
- 51) For the local variables when memory is allocated?
- 52) What is the difference between instance variables and static variables?
- 53) Can we declare instance variables inside the instance methods & static variables inside the static method?
- 54) If the local variables of methods and class instance variables having same names at that situation how we are represent local variables and how are representing instance variable?
- 55) What is the purpose of method & how many types of methods in java?
- 56) What do you mean by method signature?
- 57) What do you mean by method declaration & implementation?
- 58) What is the purpose of template method?
- 59) Can we have inner methods in java?
- 60) One method is able to call how many methods at time?
- 61) For java methods return type is mandatory or optional?
- 62) Who will create and destroy stack memory in java?
- 63) When we will get StackOverflowError?
- 64) Is it possible to declare return statement any statement of the method or any specific rule is there?
- 65) When we will get "variable might not have been initialized" error message?
- 66) What are the different ways to create a object?
- 67) By using which keyword we are creating object in java?
- 68) Object creation syntax contains how many parts?
- 69) How many types of constructors in java?
- 70) What are the advantages of constructors in java?
- 71) How one constructor is calling another constructor? One constructor is able to call how many constructors at time?
- 72) What do you mean by instantiation?
- 73) What is the difference between named Object & nameless object?
- 74) What do you mean by eager object creation & lazy object creation?
- 75) What is the difference between object instantiation and object initialization?
- 76) What is the purpose of this keyword?
- 77) Is it possible to use this keyword inside static area?
- 78) What is the need of converting local variables to instance variables?
- 79) Is it possible to convert instance variables to local variables yes → how no → why?
- 80) When we will get compilation error like "call to this must be first statement in constructor"?
- 81) When we will get compilation error line "cannot find symbol"?
- 82) What do u mean by operator overloading, is it java supporting operator overloading concept?
- 83) What is the purpose of scanner class and it is present in which package and introduced in which version?
- 84) What are the applicable modifiers for constructors?
- 85) Who is generating default constructor and at what time?
- 86) What is object and what is relationship between class and Object?
- 87) Is it possible to execute default constructor and user defined constructor time?
- 88) What is the purpose of instance block & what is the syntax?

- 89) What is the difference between instance block & constructor?
- 90) What do you mean by object delegation?
- 91) What is the purpose of instance blocks when it will execute?
- 92) Inside class it is possible to declare how many instance blocks & constructors ?
- 93) What is the purpose of static block & what is the execution process?
- 94) For a class I am creating ten objects so how many times instance blocks are executed & how many times static blocks are executed?
- 95) How to load the .class file into memory programmatically?
- 96) How to create the object of loaded class in java?
- 97) To execute the static block inside the class main method mandatory or optional?
- 98) Is it possible to print some statements in output console without using main method or not?
- 99) What is execution flow of method VS constructor Vs instance blocks Vs static blocks?
- 100) When instance blocks and static blocks are executed?

### **Flow control statement**

- 1) How many flow control statements in java?
- 2) What is the purpose of conditional statements?
- 3) What is the purpose of looping statements?
- 4) What are the allowed arguments of switch?
- 5) When we will get compilation error like “possible loss of precision”?
- 6) Inside the switch case vs. default vs. Break is optional or mandatory?
- 7) While declaring switch braces are mandatory or optional?
- 8) Switch is allowed String argument or not?
- 9) Switch allows float,double,long arguments or not?
- 10) Inside the switch how many cases are possible and how many default declarations are possible?
- 11) What are the advantages of fall through inside the switch.
- 12) What is difference between if & if-else & switch?
- 13) What is the default condition of for loop?
- 14) Inside for initialization & condition & increment/decrement parts optional or mandatory?
- 15) What is the difference between for loop & for-each loop?
- 16) What is the difference between iterable & Iterator?
- 17) When we will get compilation error like “incompatible types”?
- 18) We are able to use break statements how many places and what are the places?
- 19) What is the difference between break& continue?
- 20) What do you mean by transfer statements and what are transfer statements present in java?
- 21) for (;) representing?
- 22) When we will get compilation error like “unreachable statement ”?
- 23) Is it possible to declare while without condition yes - →what is default condition no →what is error?
- 24) What is the difference between while and do-while?
- 25) While declaring if , if-else , switch curly braces are optional or mandatory?

**Oops**

- 1) What are the main building blocks of oops?
- 2) What do you mean by inheritance?
- 3) How to achieve inheritance concept and inheritance is also known as?
- 4) How many types of inheritance in java and how many types of inheritance not supported by java?
- 5) How to prevent inheritance concept?
- 6) If we are extending the class then your class will become parent class but if we are not extending what is the parent class?
- 7) One class able to extends how many classes at a time?
- 8) What is the purpose of extends keyword?
- 9) What do you mean by cyclic inheritance java supporting or not?
- 10) What is the difference between child class and parent class?
- 11) Which approach is recommended to create object either parent class object or child class object?
- 12) Except one class all class contains parent class in java what is that except class?
- 13) What is the purpose of instanceof keyword in java?
- 14) What is the root class for all java classes?
- 15) How to call super class constructors?
- 16) Is it possible to use both super and this keyword inside the method?
- 17) Is it possible to use both super and this keyword inside the constructor?
- 18) Inside the constructor if we are not providing this() and super() keyword the compiler generated which type of super keyword?
- 19) What is the execution process of constructors if two classes are there in inheritance relationship?
- 20) What is the execution process of instance blocks if two classes are there in inheritance relationship?
- 21) What is the execution process of static blocks if two classes are there in inheritance relationship?
- 22) What is the purpose of instanceof operator in java & what is the return-type?
- 23) If we are using instanceof both reference-variable & class-name must have some relationship otherwise compiler generated error message is what?
- 24) If the child class and parent class contains same variable name that situation how to call parent class variable in child class?
- 25) What do you mean by aggregation and what is the difference between aggregation and inheritance?
- 26) What do you mean by aggregation and composition and Association?
- 27) Aggregation is also known as?
- 28) How many objects are created ?
  - a. MyClassHero c1,c2;
- 29) What do you mean by polymorphism?
- 30) How many types of polymorphism in java?
- 31) What do you mean by method overloading and method overriding?
- 32) How many types of overloading in java?
- 33) Java supports operator overloading or not?
- 34) Is it possible to overload the constructors are not?
- 35) What do you mean by constructor overloading?

- 36) What are the implicit overloaded operators in java explain it?
- 37) What do you mean by overriding? What are rules must follow while performing method overriding?
- 38) What do you mean by overridden method & overriding method?
- 39) To achieve overriding how many java classes are required?
- 40) Is it possible to override variable in java?
- 41) When we will get compilation error like “overridden method is final”?
- 42) What is the purpose of final modifier java?
- 43) Is it possible to override static methods yes → how no → why?
- 44) Parent class reference variable is able to hold child class object or not?
- 45) What do you mean by dynamic method dispatch?
- 46) The applicable modifiers only on local variables?
- 47) What do you mean by type casting & how many types?
- 48) Is it all methods present in final class is always final and all variables present final class is always final or not ?
- 49) If Parent class is holding child class object then by using that we are able to call only overridden methods of child class but how to call direct methods of child class?
- 50) Object class present in which package & it contains how many methods?
- 51) When we will get compilation error like “con not inherit from final parent”?
- 52) What do you mean by co-variant return types?
- 53) What do you mean by method signature?
- 54) What do u mean by method hiding and how to prevent method hiding concept?
- 55) What do you mean by abstraction?
- 56) How many types of classes in java generally?
- 57) Normal class is also known as ?
- 58) What is the difference between normal method and abstract method?
- 59) What is the difference between normal class and abstract class?
- 60) Is it possible to create a object for abstract class?
- 61) What do you mean by abstract variable?
- 62) Is it possible to override non-abstract method as a abstract method?
- 63) Is it possible to declare main method inside the abstract class or not?
- 64) What is the purpose of abstract modifier in java?
- 65) How to prevent object creation in java?
- 66) What is the definition of abstract class?
- 67) In java is it abstract class reference variable is able to hold child class object or not?
- 68) What do you mean by encapsulation & what is the examples encapsulation?
- 69) What do you mean by tightly encapsulated class?
- 70) What do you mean accessor method and mutator method ?
- 71) How many ways area there to set some values to class properties (variables)?
- 72) What do you mean by javaBean class?
- 73) The javabean class is also known as?
- 74) In java program execution starts from which method & who is calling that method?
- 75) Can we inherit main method in child class?
- 76) The applicable modifiers on main method?
- 77) While declaring main method public static modifiers order mandatory or optional?
- 78) What is the argument of main method?
- 79) What is the return type of main method?

- 80) What are the mandatory modifiers for main method and optional modifiers of main method?
- 81) Why main method is public & static?
- 82) What do you by command line arguments & command line arguments are stored in which format(type)?
- 83) Is it possible to pass command line arguments with space symbol no→ why yes→how ?
- 84) What is the purpose of strictfp modifier?
- 85) What is the purpose of native modifier?
- 86) What do you mean by native method and it also known as?
- 87) Is it possible to overload the main method or not yes=how no=why?
- 88) Is it possible to override the main method or not yes=how no=why?
- 89) What is the purpose of variable argument method & what is the syntax?
- 90) If the application contains both normal argument & variable argument then which one executed first?
- 91) The java method allows both variable argument & normal argument in single method?
- 92) Is it possible to overload the variable argument methods are not?
- 93) What is the difference between method overloading & variable argument method.
- 94) What are the modification are allowed on main method?
- 95)

## **Packages**

1. What do you mean by package and what it contains?
2. How many pre-defined packages in java?
3. What is the default package in java?
4. Is it possible to declare package statement any statement of the source file?
5. What is the difference between user's defined package and predefined package?
6. What are coding conventions must follow while declaring user defined package names?
7. Is it possible to declare multiple packages in single source file?
8. What do you mean by import?
9. What is the location of predefined packages in our system?
10. How many types of imports present in java explain it?
11. How to import individual class and all classes of packages and which one is recommended?
12. What do you mean by static import?
13. What is the difference between normal and static import?
14. I am importing two packages, both packages contains one class with same name at that situation how to create object of two package classes?
15. If we are importing root package at that situation is it possible to use sub package classes in our applications?
16. What is difference between main package and sub package?
17. If source file contains package statement then by using which command we are compiling that source file?
18. What do you mean by fully qualified name of class?
19. What is the default modifier in java?
20. What is the public access and default access?
21. The public class members(variables,methods,constructors) are by default public or not?
22. What is private access and protected access?
23. What is most restricted modifier in java?
24. What is most accessible modifier in java?
25. Is it possible to declare pre-defined package names as a user defined package names or not?
26. What are the applicable modifiers for constructors?
27. Is it possible to override private methods or not yes=how no=why?
28. When we will get compilation error like "attempting to assign weaker access privileges" how to rectify?

### Exception handling

1. What do you mean by Exception?
2. How many types of exceptions in java?
3. What is the difference between Exception and error?
4. What is the difference between checked Exception and un-checked Exception?
5. Is it possible to handle Errors in java?
6. Explain exception handling hierarchy?
7. What the difference is between partially checked and fully checked Exception?
8. What do you mean by exception handling?
9. How many ways are there to handle the exception?
10. What is the root class of Exception handling?
11. Can you please write some of checked and un-checked exceptions in java?
12. What are the keywords present in Exception handling?
13. What is the purpose of try block?
14. In java is it possible to write try without catch or not?
15. What is the purpose catch block?
16. What is the difference between try-catch?
17. Is it possible to write normal code in between try-catch blocks?
18. What are the methods used to print exception messages?
19. What is the purpose of printStackTrace( ) method?
20. What is the purpose of finally block?
21. If the exception raised in catch block what happened?
22. Independent try blocks are allowed or not allowed?
23. Once the control is out of try , is it remaining statements of try block is executed or not?
24. Try-catch , try-catch-catch , catch-catch , catch-try how many combinations are valid?
25. Try-catch-finally , try-finally ,catch-finally , catch-catch-finally how many combinations are valid?
26. Is possible to write code in between try-catch-finally blocks?
27. Is it possible to write independent catch blocks?
28. Is it possible to write independent finally block?
29. What is the difference between try-catch –finally?
30. Is it allows to use nested try-catch in java?
31. For the method argument it us possible to provide exceptions?
32. What do you mean by exception propagation?
33. is the checked Exceptions are propagated or not?
34. If the exception raised in finally block what happened?
35. What are the situations finally block is not executed?
36. What is the relation of finally & return statement.
37. Try,catch,finally three blocks are returning value then which one taken as a final value.
38. What is the purpose of throws keyword?
39. What is the difference between try-catch blocks and throws keyword?

40. What do you mean by default exception handler and what is the purpose of default exception handler?
41. What is the purpose of throw keyword?
42. If we are writing the code after throw keyword usage then what happened?
43. What is the difference between throw and throws keyword?
44. How to create user defined checked exceptions?
45. How to create user defined un-checked exceptions?
46. Where we placed clean-up code like resource release, database closeting inside the try or catch or finally and why ?
47. Write the code of ArithmeticException?
48. Write the code of NullPointerException?
49. Write the code of ArrayIndexOutOfBoundsException & StringIndexOutOfBoundsException?
50. Write the code of IllegalThreadStateException?
51. When we will get InputMismatchException?
52. When we will get IllegalArgumentException?
53. When we will get ClassCastException?
54. When we will get OutOfMemoryError?
55. What is the difference between ClassNotFoundException & NoClassDefFoundError?
56. When we will get compilation error like “unreportedException must be catch”?
57. When we will get compilation error like “Exception XXXException has already been caught”?
58. When we will get compilation error like “try without catch or finally”?
59. How many approaches are there to create user defined unchecked exceptions and un-checked exceptions?
60. How to create object of user defined exceptions?
61. How to handover user created exception objects to JVM?
62. Is it possible to handle different exceptions by using single catch block yes-->how no → why?
63. What is the purpose of try with resources?
64. Relation with Exception handling & overriding?
65. How to progate checked & unchecked Exceptions?

### Interfaces

- a. What do you mean by interface how to declare interfaces in java?
- b. Interfaces allows normal methods or abstract methods or both?
- c. For the interfaces compiler generates .class files or not?
- d. Interface is also known as?
- e. What is the abstract method?
- f. By default modifiers of interface methods?
- g. What is the purpose of implements keyword?
- h. Is it possible to declare variables in interface ?
- i. Can abstract class have constructor ? can interface have constructor?
- j. What must a class do to implement interface?
- k. What do you by implementation class?
- l. Is it possible to create object of interfaces?
- m. What do you mean by abstract class?
- n. When we will get compilation error like “attempting to assign weaker access privileges”?
- o. What is the difference between abstract class and interface?
- p. What do you mean by helper class?
- q. Which of the fallowing declarations are valid & invalid?
  - a. ***class A implements it1***
  - b. ***class A implements it1,it2,it3***
  - c. ***interface it1 extends it2***
  - d. ***interface it1 extends it2,it3***
  - e. ***interface it1 extends A***
  - f. ***interface it1 implements A***
- r. what is the difference between classes and interfaces?
- s. The interface reference variable is able to hold implementation class objects or not?
  - a. Interface-name reference-variable = new implementation class object(); valid or invalid
- t. What is the real-time usage of interfaces?
- u. what is the limitation of interfaces how to overcome that limitation?
- v. What do you mean by adaptor class?
- w. What is the difference between adaptor class interfaces?
- x. Is it possible to create user defined adaptor classes?
- y. Tell me some of the adaptor classes?
- z. What do you mean by marker interface and it is also known as?
- aa. Define marker interfaces?
- bb. What are the advantages of marker interfaces?
- cc. Is it possible to create user defined marker interfaces or not?
- dd. Is it possible to declare nested interfaces or not?

Different types of methods in java (must know information about all methods)

- 1) Instance method
- 2) Static method
- 3) Normal method
- 4) Abstract method
- 5) Accessor methods
- 6) Mutator methods
- 7) Inline methods
- 8) Call back methods
- 9) Synchronized methods
- 10) Non-synchronized methods
- 11) Overriding method
- 12) Overridden method
- 13) Instance Factory method
- 14) Static factory method
- 15) Pattern factory method
- 16) Template method
- 17) Default method
- 18) Public method
- 19) Private method
- 20) Protected method
- 21) Final method
- 22) Strictfp method
- 23) Native method

Different types of classes in java (must know information about all classes)

- 1) Normal class /concrete class /component class
- 2) Abstract class /helper class
- 3) Tightly encapsulated class
- 4) Public class
- 5) Default class
- 6) Adaptor class
- 7) Final class
- 8) Strictfp class
- 9) JavaBean class /DTO(Data Transfer Object) /VO (value Object)/BO(Business Object)
- 10) Singleton class
- 11) Child class
- 12) Parent class
- 13) Implementation class

Different types of variables in java (must know information about all variables)

- 1) Local variables
- 2) Instance variables
- 3) Static variables
- 4) Final variables
- 5) Private variables
- 6) Protected variables
- 7) Volatile variables
- 8) Transient variables
- 9) Public variables

***String manipulation***

- 1) How many ways to create a String object & StringBuffer object?
- 2) What is the difference between
  - a. String str="ratan";
  - b. String str = new String("ratan");
- 3) equals() method present in which class?
- 4) What is purpose of String class equals() method.
- 5) What is the difference between equals() and == operator?
- 6) What is the difference between by immutability & immutability?
- 7) Can you please tell me some of the immutable classes and mutable classes?
- 8) String & StringBuffer & StringBuilder & StringTokenizer presented package names?
- 9) What is the purpose of String class equals() & StringBuffer class equals()?
- 10) What is the purpose of StringTokenizer and this class functionality replaced method name?
- 11) How to reverse String class content?
- 12) What is the purpose of trim?
- 13) Is it possible to create StringBuffer object by passing String object as a argument?
- 14) What is the difference between concat() method & append()?
- 15) What is the purpose of concat() and toString()?
- 16) What is the difference between StringBuffer and StringBuilder?
- 17) What is the difference between String and StringBuffer?
- 18) What is the difference between compareTo() vs equals()?
- 19) What is the purpose of contains() method?
- 20) What is the difference between length vs length()?
- 21) What is the default capacity of StringBuffer?
- 22) What do you mean by factory method?
- 23) Concat() method is a factory method or not?
- 24) What is the difference between heap memory and String constant pool memory?
- 25) String is a final class or not?
- 26) StringBuilder and StringTokenizer introduced in which versions?
- 27) What do you mean by legacy class & can you please give me one example of legacy class?
- 28) How to apply StringBuffer class methods on String class Object content?
- 29) When we use String & StringBuffer & String
- 30) What do you mean by cloning and use of cloning?
- 31) Who many types of cloning in java?
- 32) What do you mean by cloneable interface present in which package and what is the purpose?
- 33) What do you mean by marker interface and Cloneable is a marker interface or not?
- 34) How to create duplicate object in java(by using which method)?

**Wrapper classes**

1. What is the purpose of wrapper classes?
2. How many Wrapper classes present in java what are those?
3. How many ways are there to create wrapper objects?
4. When we will get NumberFormatException?
5. How many constructors are there to create Character Wrapper class Object ?
6. How many constructors are there to create Integer Wrapper class?
7. How many constructors are there to create Float Wrapper class?
8. What do you mean by factory method?
9. What is the purpose of valueOf() method is it factory method or not?
10. How to convert wrapper objects into corresponding primitive values?
11. What is the implementation of toString() in all wrapper classes?
12. How to convert String into corresponding primitive?
13. What do you mean by Autoboxing and Autounboxng & introduced in which version?
14. Purpose of parseXXX() & xxxValue() method?
15. Which Wrapper classes are direct child class of Object class?
16. which Wrapper classes are direct child class of Number class?
17. How to convert primitive to String?
18. When we will get compilation error like "int cannot be dereferenced"?
19. Wrapper classes are immutable classes or mutable classes?
20. Perform following conversions int-->String String-->int Integer-->int int-->Integer ?

**Garbage Collector**

1. What is the functionality of Garbage collector?
2. How many ways are there to make eligible our objects to Garbage collector?
3. How to call Garbage collector explicitly?
4. What is the purpose of gc( ) method?
5. What is the purpose of finalize() method?
6. If the exception raised in finalize block what happened **error or output?**
7. What is the purpose of RunTime class?
8. How to create object of RunTime class?
9. What is singleton class?
10. What is the algorithm followed by GC?
11. What is the difference between final , finally , finalize()?
12. When GarbageCollector calls finalize()?
13. Finalize method present in which class?
14. Which part of the memory involved in garbage collector Heap or Stack?
15. Who creates stack memory and who destroy that memory?
16. What do you mean by demon thread? Is Garbage collector is DemonThread?
17. How many times Garbage collector does call finalize() method for object?
18. What are the different ways to call Garbage collector ?
19. How to enable/disable call of finalize()?
20. Is it possible to call finalize() method explicitly by the programmer?

**Collections**

- 1) What is collection? What is Collection framework?
- 2) What is the main objective of collections?
- 3) What are the advantages of collections over arrays?
- 4) Collection frame work classes are present in which package?
- 5) By using collection framework classes is it possible to store primitive data?
- 6) What is the root interface of collection framework?
- 7) List out implementation classes of List interface?
- 8) List out implementation classes of set interface?
- 9) List out implementation classes of map interface?
- 10) What is the parent interface of Collection interface?
- 11) What is the difference between heterogeneous and homogeneous data?
- 12) What do you mean by legacy class can you please tell me some of the legacy classes present in collection framework?
- 13) What are the characteristics of collection classes?
- 14) What is the purpose of generic version of collection classes?
- 15) How to provide type safety to the collection?
- 16) What is the difference between general version of ArrayList and generic version of ArrayList?
- 17) What is purpose of generic version of ArrayList & arrays?
- 18) How to convert Collection data to arrays & Arrays data to collection?
- 19) How to get Array by using ArrayList?
- 20) What is the difference betweenArrayList and LinkedList?
- 21) How to decide when to use ArrayList and when to use LinkedList?
- 22) What is the difference between ArrayList & vector?
- 23) Which collection classes are synchronized or Threadsafe?
- 24) Which collection classes are non-synchronized or not Threadsafe?
- 25) How can ArrayList be synchronized without using vector?
- 26) Arrays are already used to hold homogeneous data but what is the purpose of generic version of Collection classes because generic version also used to store homogeneous data?
- 27) What is the purpose of RandomAccess interface and it is marker interface or not?
- 28) What do you mean by cursor and how many cursors present in java?
- 29) How many ways are there to retrieve objects from collections classes what are those?
- 30) What is the purpose of Enumeration cursor and how to get that cursor object?
- 31) By using how many cursors we are able to retrieve the objects both forward backward direction and what are the cursors?
- 32) What is the purpose of Iterator and how to get Iterator Object?
- 33) What is the purpose of ListIterator and how to get that object?
- 34) What is the difference between Enumeration vs Iterator Vs ListIterator?
- 35) We are able to retrieve objects from collection classes by using cursors and for-each loop what is the difference?
- 36) By using which cursor it is possible to read the data from forward and backward directions?
- 37) All collection classes are commonly implemented some interfaces what are those interfaces?
- 38) What is the difference between HashSet & linkedHashSet?
- 39) all most all collection classes are allowed heterogeneous data but some collection classes are not allowed can you please list out the classes?
- 40) What is the purpose of TreeSet class?
- 41) What is the difference between Set & List interface?
- 42) What is the purpose of Map interface?

- 43) What do you mean by entry.
- 44) What is the difference between HashMap & LinkedHashMap?
- 45) What is the difference between comparable vs Comparator interface?
- 46) What is the difference between TreeSet and TreeMap?
- 47) What is the difference between HashTable and Properties file key=value pairs?
- 48) What do you mean by properties file and what are the advantages of properties file?
- 49) Properties class present in which package?
- 50) What is the difference between collection & collections?

#### Enumeration

- 1) What is the purpose enum in java?
- 2) How to declare enum & compiler will generate .class files or not?
- 3) enum constants are by default modifiers are?
- 4) What is the difference enum & Enum?
- 5) What is the difference between java enum & other language enums.
- 6) Is it possible to declare main method & constructor inside the enum or not?
- 7) Is it possible to provide parameterized constructor inside the enum?
- 8) Inside the enum group of constants ends with semicolon is optional or mandatory?
- 9) What is the difference between java enum and java class?
- 10) What is the purpose of values() & ordinal() methods?
- 11) Is it possible to create object for enum or not?
- 12) For enum inheritance concept is applicable or not?
- 13) When enum constants are loaded?
- 14) Enums are able to implement interfaces or not?
- 15) One enum is able extends other enum or not?
- 16) What is the difference between **enum & Enumeration & Enum**?
- 17) Can you use enum constants switch case in java?
- 18) What is the modifier applicable for enum constructor?
- 19) Is it possible to declare the enum inside the class or not & inside the method or not&outside of the class or not?
- 20) When we access enum constants outside of the package directly without using enum name then normal import required or static import required?

**Nested classes**

- 1) What are the advantages of inner classes?
- 2) How many types of nested class?
- 3) How many types of inner classes?
- 4) What do you by static inner classes?
- 5) What is the relation between inner classes & nested classes?
- 6) The inner class is able to access outer class private properties or not?
- 7) The outer class is able to access inner classes properties& methods or not?
- 8) How to create object inner class and outer class?
- 9) .class file names of Inner & outer classes?
- 10) The outer class object is able to call inner class properties & methods or not?
- 11) The inner class object is able to call outer class properties and methods or not?
- 12) What is the difference between normal inner classes and static inner classes?
- 13) How to prevent the object creation of inner classes?
- 14) One inner class able to extends another inner class or not?
- 15) What do you mean by anonymous inner classes?
- 16) What do you mean by method local inner classes?
- 17) Is it possible to create inner class object without outer class object?
- 18) Java supports inner methods concept or not?
- 19) Is it possible to declare main method inside inner classes & outer class?
- 20) Is it possible to declare constructors inside inner classes?
- 21) If outer class variables and inner class variables are having same name then how to represent outer class variables and how to represent inner class variables?
- 22) Is it possible to declare same method in both inner class and outer class?
- 23) What are the applicable modifiers on Outer classes?
- 24) What are the applicable modifiers on inner classes?
- 25) What are the applicable modifiers on method local inner classes?

**File IO**

1. What is the purpose of java.io package?
2. What do you mean by stream?
3. What do you mean by channel and how many types of channels present in java?
4. What is the difference between normal stream & buffered Streams?
5. What is the difference between FileInputStream & BufferedReader?
6. What is the difference between FileOutputStream & printwriter?
7. Println() method present in which class?
8. Out is which type of variable(instance /static ) present in which class?
9. To create byte oriented channel we required two class what are those classes?
10. To create character oriented channel we required two class what are those classes?
11. What is the difference between byte oriented channel and character oriented channel?
12. What is the difference between read() & readLine() method?
13. What is the difference between normal Streams & bufferd streams?
14. Wat is the purpose of write() & println() ?
15. Example classes normal Streams & bufferd streams?
16. What do you mean by serialization?
17. What is the purpose of Serializable interface& it is marker interface or not ?
18. How to prevent serialization concept?
19. What do you mean deserialization?
20. To perform deserialization we required two classes what are those classes?

21. To perform serialization we required two classes what are those classes?
22. What is the purpose of transient modifier?
23. What are advantage of serialization?
24. Serializable interface present in which package?
25. When we will get IOException how many ways are there to handle the exceptions?
26. IOException is checked Exception or unchecked Exception?

### Multithreading

1. What do you mean by Thread?
2. What do you mean by single threaded model?
3. What is the difference single threaded model and multithreaded model?
4. What do you mean by main thread and what is the importance?
5. What is the difference between process and thread?
6. How many ways are there to create thread which one prefer?
7. Thread class & Runnable interface present in which package?
8. Runnable interface is marker interface or not?
9. What is the difference between t.start() & t.run() methods where t is object of Thread class?
10. How to start the thread?
11. What are the life cycle methods of thread?
12. Run() method present in class/interface ? Is it possible to override run() method or not?
13. Is it possible to override start method or not?
14. What is the purpose of thread scheduler?
15. Thread Scheduler follows which algorithm?
16. What is purpose of thread priority?
17. What is purpose of sleep() & isAlive() & isDaemon() & join() & getId() & activeCount() methods?
18. Jvm creates stack memory one per Thread or all threads only one stack?
19. What is the thread priority range & how to set priority and how to get priority?
20. What is the default name of user defined thread and main thread? And how to set the name and how to get the name?
21. What is the default priority of main thread?
22. Which approach is best approach to create a thread?
23. What is the difference between synchronized method and non-synchronized method?
24. What is the purpose of synchronized modifier?
25. What is the difference between synchronized method and non synchronized method?
26. What do you mean by demon thread tell me some examples?
27. what is the purpose of volatile modifier?
28. What is the difference between synchronized method and synchronized block?
29. Wait() notify() notifyAll() methods are present in which class?
30. When we will get Exception like "IllegalThreadStateException" ?
31. When we will get Exception like "IllegalArgumentException" ?
32. If two threads are having same priority then who decides thread execution?
33. How two threads are communicate each other?
34. What is race condition?
35. How to check whether the thread is demon or not? Main thread is demon or not?
36. How a thread can interrupt another thread?
37. Explain about wait() notify() notifyAll()?
38. Once we create thread what is the default priority?
39. What is the max priority & min priority & norm priority?
40. What is the difference between preemptive scheduling vs time slicing?

**Internationalization**

- 1) What is the main importance of I18n?
- 2) What is the purpose of locale class?
- 3) What is the format of the properties file?
- 4) Local class present in which package?
- 5) What do you mean by properties file and what it contains?
- 6) What is the purpose of ResourceBundle class and how to create object?
- 7) How to convert different languages characters into Unicode characters?
- 8) What is the command used to convert different language characters into Unicode characters?
- 9) Who decides properties file executions?
- 10) What is the method used to get values from properties file?
- 11) By using which classes we are achieving i18n?
- 12) What is the default Locale and how to get it?
- 13) Is it possible to create your own locale?
- 14) What is purpose of DateFormat class and it is preset in which package?
- 15) What are the DateFormat Constantans' to print Date & time?
- 16) How to print date in different Locales?
- 17) How to print time in different locales?
- 18) How to print both date & time by using single method?
- 19) What do you mean by factory method? getBundle() is factory method or not?
- 20) How to get particular locale language & country?

**Java Versions, Features and History:-**

**Released on 23 January 1996, JDK 1.0 version.**

**Released on 19 February 1997 JDK 1.1 version.**

**New features in JDK 1.1**

- ✓ JDBC (Java Database Connectivity)
- ✓ Inner Classes
- ✓ Java Beans
- ✓ RMI (Remote Method Invocation)
- ✓ Reflection (introspection only)

**Released on 8 December 1998 J2SE 1.2 version.**

**New features in J2SE 1.2**

- ✓ Collections framework.
- ✓ Java String memory map for constants.
- ✓ Just In Time (JIT) compiler.
- ✓ Jar Signer for signing Java ARchive (JAR) files.
- ✓ Policy Tool for granting access to system resources.
- ✓ Java Foundation Classes (JFC) which consists of Swing 1.0, Drag and Drop, and Java 2D class libraries.
- ✓ Java Plug-in
- ✓ Scrollable result sets, BLOB, CLOB, batch update, user-defined types in JDBC.
- ✓ Audio support in Applets.

**Released on 8 May 2000 J2SE 1.3 version.**

**New features in J2SE 1.3**

- ✓ Java Sound
- ✓ Jar Indexing
- ✓ A huge list of enhancements in almost all the java area.

**Released on 6 February 2002 J2SE 1.4 version.**

**New features in J2SE 1.4**

- |                                                                                                                                                                                                 |                                                                                                                                                                                  |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>✓ XML Processing</li> <li>✓ Java Print Service</li> <li>✓ Logging API</li> <li>✓ Java Web Start</li> <li>✓ JDBC 3.0 API</li> <li>✓ Assertions</li> </ul> | <ul style="list-style-type: none"> <li>✓ Preferences API</li> <li>✓ Chained Exception</li> <li>✓ IPv6 Support</li> <li>✓ Regular Expressions</li> <li>✓ Image I/O API</li> </ul> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**Released on 30 September 2004 J2SE 1.5 version.**

**New features in J2SE 1.5**

- |                                                                                                                                          |                                                                                                                                                   |
|------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>✓ Generics</li> <li>✓ Enhanced for Loop</li> <li>✓ Autoboxing/Unboxing</li> <li>✓ Enum</li> </ul> | <ul style="list-style-type: none"> <li>✓ Varargs</li> <li>✓ Static Import</li> <li>✓ Metadata (Annotations)</li> <li>✓ Instrumentation</li> </ul> |
|------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|

**Released on 11 December 2006 J2SE 1.6 version.**

**New features in J2SE 1.6**

- ✓ Scripting Language Support
- ✓ JDBC 4.0 API
- ✓ Java Compiler API
- ✓ Pluggable Annotations
- ✓ Native PKI, Java GSS, Kerberos and LDAP support.
- ✓ Integrated Web Services.
- ✓ Lot more enhancements.

**Released on 28 July 2011 J2SE 1.7 version.**

**New features in J2SE 1.7**

- ✓ Strings in switch Statement
- ✓ Type Inference for Generic Instance Creation
- ✓ Multiple Exception Handling
- ✓ Try with Resources
- ✓ Java nio Package
- ✓ Binary Literals, underscore in literals

**Released on 18th march 2014 JDK 1.8 version.**

**New features in JDK 1.8**

- |                                                                                                                                                                               |                                                                                                                   |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>✓ Default and Static methods in Interface</li> <li>✓ Lambda Expressions</li> <li>✓ Functional interface</li> <li>✓ Optional</li> </ul> | <ul style="list-style-type: none"> <li>✓ Streams</li> <li>✓ Method References</li> <li>✓ Data Time API</li> </ul> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|