

Java means DURGA SOFT..

# CORE JAVA

## Material



India's No.1 Software Training Institute  
**DURGASOFT**  
www.durgasoft.com Ph: 9246212143 ,8096969696

JAVA Means Durga Soft

**Index Page**

<b>1) Introduction</b>	<b>1-18</b>
a. Flow control statements	19
b. Variables	32
c. Methods	41
d. Constructors	57
<b>2) Oops</b>	<b>74</b>
a. Inheritance	
b. Polymorphism	97
c. Garbage Collector	112
d. Abstraction	114
e. Main method	121
f. Encapsulation	
<b>3) Packages</b>	<b>126-137</b>
<b>4) Interfaces</b>	<b>138-148</b>
<b>5) String manipulations</b>	<b>149-161</b>
<b>6) Wrapper classes</b>	<b>162-169</b>
<b>7) Java.io</b>	<b>170-175</b>
<b>8) Exception handling</b>	<b>176-199</b>
<b>9) Multi Threading</b>	<b>200 – 217</b>
<b>10) Nested classes</b>	<b>218 – 228</b>
<b>11) Enumeration</b>	<b>229- 231</b>
<b>12) Collections &amp; generics</b>	<b>232-279</b>
<b>13) Networking</b>	<b>280 – 285</b>
<b>14) Java.awt</b>	<b>286-311</b>
<b>15) Swings</b>	<b>312-318</b>
<b>16) i18n</b>	<b>319-333</b>
<b>17) Arrays</b>	<b>334-336</b>
<b>18) Java interview questions</b>	<b>337-341</b>
<b>19) Core java classroom schedule</b>	<b>357</b>

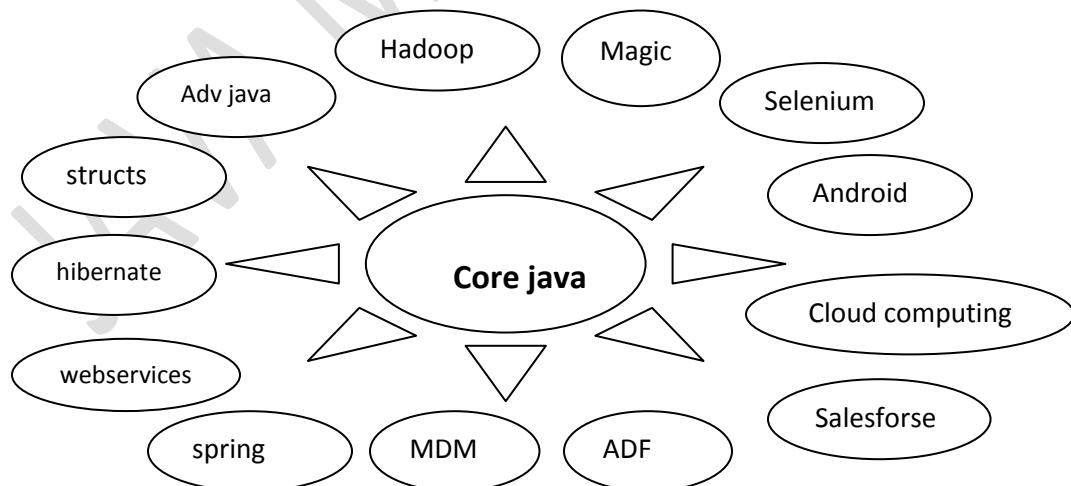
**JAVA introduction:-**

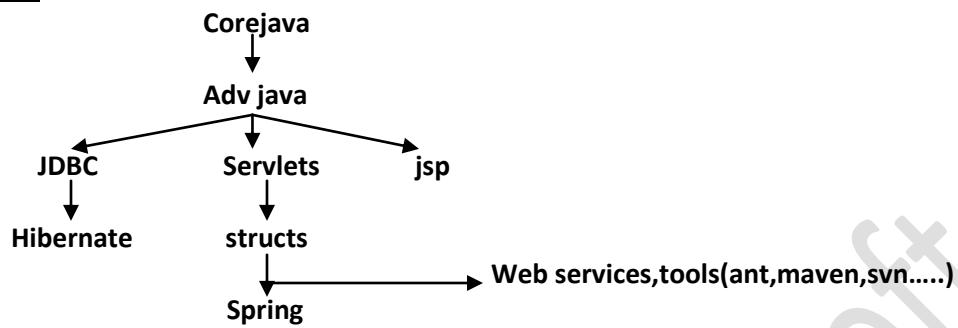
Author	:	<b>James Gosling</b>
Vendor	:	<b>Sun Micro System</b> (which has since merged into Oracle Corporation)
Project name	:	<b>Green Project</b>
Type	:	<b>open source &amp; free software</b>
Initial Name	:	<i>OAK language</i>
Present Name	:	<b>java</b>
Extensions	:	<i>.java &amp; .class &amp; .jar</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 srujanjobs.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:-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:-Data Types

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

Flow-Control:-

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

method-level:-

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

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

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)**

**JAVA VERSIONS:-**

<b>VERSION</b>	<b>YEAR</b>
Java Alpha & beta	1995
JDK 1.0	1996
JDK1.1	1997
J2SE 1.2	1998
J2SE 1.3	2000
J2SE 1.4	2002
J2SE 1.5	2004
JAVA SE 6	2006
JAVA SE 7	2011
JAVA SE 8	2014

applications use #include statement.

**Ex:- #include<stdio.h>**

7) To print some statements into output console use “printf” function.

**Printf(“hi ratan ”);**

8) extensions used :- .c ,.obj , .h

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

**Ex:- iostream.h**

5) The header files contains predefined functions.

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

6) To make available predefined support into our application use #include statement.

**Ex:- #include<iostream>**

7) To print the statements use “cout” function.

**Cout<<“hi ratan”;**

8) extensions used :- .cpp ,.h

**Cpp-lang**

1) Author : Bjarne Stroustrup

2) implementation languages are c ,ada,ALGOL68.....

3) program execution starts from main method called by **operating system**.

5) The header files contain predefined functions.

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

6) To make available predefined support into our

**Java -lang**

1) Author : James Gosling

2) implementation languages are C,CPP,ObjectiveC.....

3) program execution starts from main method called by **JVM(java virtual machine)**.

4) In java predefined support available in the form of packages.

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

[\*] mean all

5) The packages contains predefined classes and class contains predefined funtions.

**Ex:- String, System**

6) To make available predefined support into our application use import statement.

**Ex:- import java.lang.\*;**

7)To print the statements we have to use

**System.out.println("hi ratan");**

8)extensions used :-  
**.java, .class**

#### C –sample application:-

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

#### Java sample application:-

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

#### c-language:-

c-language  
↓  
headerfiles  
↓  
functions

Dennis Ritchie  
↓  
stdio.h,conio.h  
↓  
printf,scanf.....

**void main()** (execution startsfrom main )

**printf("ratan");** (used to print the output)

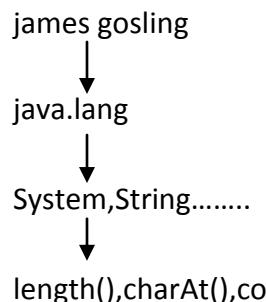
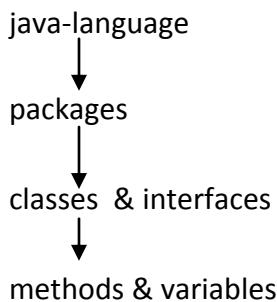
#### cpp-language:-

cpp-language  
↓  
headerfiles  
↓  
functions

Bjarne Stroustrup  
↓  
iostream.h  
↓  
cout,cin....

**void main()** (execution startsfrom main)

**cout<<"ratan";** (used to print the output)

**java-language:-**

**public static void main(String[] args)**  
(execution starts from main)  
**System.out.println("ratan");**  
(used to print the output)

**JAVA Features:-**

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

**1. Simple:-**

Java is a simple programming language because:

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

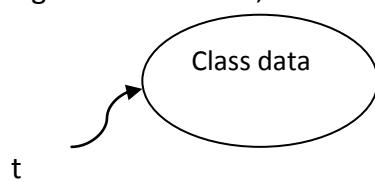
**2. Object Oriented:-**

Java is object oriented technology because to represent total data in the form of object.

By using object reference we are calling all the methods, variables which is present in that class.

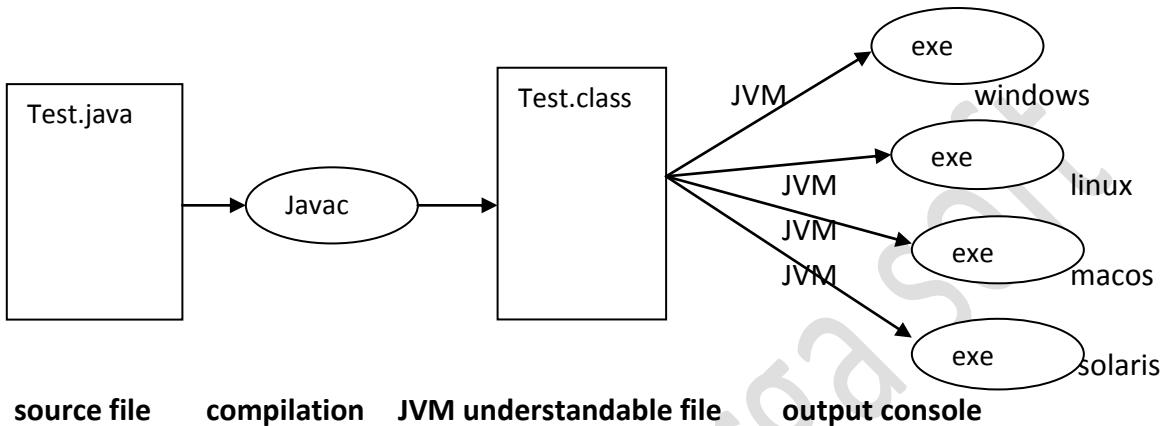
***Class Test***

{logics }      **Test t=new Test();**

**3. Platform Independent :-**

- Compile the Java program on one OS (operating system) that compiled file can execute in any OS (operating system) is called Platform Independent Nature.

- The java is platform independent language. The java applications allow its applications compilation one operating system that compiled (.class) files can be executed in any operating system.



#### **4. Architectural Neutral:-**

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

#### **5. Portable:-**

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

#### **6. Robust:-**

Any technology if it is good at two main areas it is said to be ROBUST

1. Exception Handling
2. Memory Allocation

JAVA is Robust because

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

#### **7. Secure:-**

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

**8. Dynamic:-**

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

**9. Distributed:-**

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

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

**web applications** are java applications it need client server architecture.

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

**10. Multithreaded: -**

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

**11. Interpretive:-**

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

**12. High Performance:-**

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

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

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

Download the software from internet based on your operating system. The software is different from 32-bit operating and 64-bit operating system.

To download the software open the fallowing web site.

<http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html>

for 32-bit operating system please click on  
Windows x86 :- 32- bit operating system

for 64-bit operating system please click on  
Windows x64 :- 64-bit operating system

After installing the software the java folder is available in the following location

Local Disk c: -----→program Files-----→java---→jdk(java development kit),jre(java runtime environment)

To check whether the java is installed in your system or not go to the command prompt. To open the command prompt

Start -----→run-----→open: cmd----→ok

Command prompt is opened.

**In the command prompt type :- javac**

'javac' is not recognized is an internal or external command, operable program or batch file.

Whenever we are getting above information at that moment the java is installed but the java is not working properly.

C:/>javac

Whenever we are typing javac command on the command prompt

- 1) Operating system will pickup javac command search it in the internal operating system calls. The javac not available in the internal command list .
- 2) Then operating system goes to environmental variables and check is there any path is sets or not. up to now we are not setting any path. So operating system don't know anything about javac command Because of this reason we are getting error message.

**Hence we have to environmental variables. The main aim of the setting environmental variable is to make available the following commands javac,java,javap (softwares) to the operating system.**

**To set the environmental variable:-**

My Computer (right click on that) ---->properties---->Advanced--->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 good in your system. open the command prompt to check once  
C:/>javac-----→now list of commands will be displayed

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

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

**Step1:- Select Editor**

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

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

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

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

IDE is providing very good environment to develop the application and it is real-time standard but don't use IDE to develop core java applications.

**Editor vs. IDE:-**

If we are using IDE to develop core java application then 75% work is done by IDE like

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

And remaining 25% work is down by developer

If we are using EditPlus software to develop application then 100% work done by user only.

**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 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.

To import the classes into our application we are having two approaches,

- 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.**

#### **Step3:- save the application.**

- After writing the application must save the application by using (`.java`) extension.
- While saving the application must follow two rules
  - If the source file contains public class then public class and the name and Source file 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:
|--ratan
    |--Sravya.java
```

#### **example 2:- valid**

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


```

#### **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>           intial 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
```

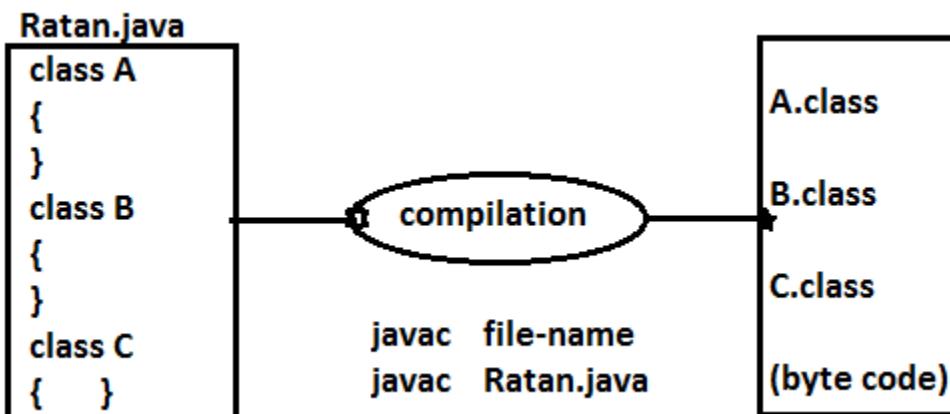
Whenever we are trying to perform compilation compiler perform following actions.

- Compiler checks the syntax error, if syntax error are there compiler generate error message.
- If syntax errors are not present then compiler generate `.class` files.

**Note:-** in java .class file 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 generate .class file and .class file contains byte code instructions it is intermediate code.



#### Process of compiling different files:-

D:

```

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.

- It will loads corresponding .class file byte code into memory. If the .class is not available JVM generate error message like "**Could not find main class**".
- After loading .class file byte into memory JVM calling main method to start the execution process. If the main method is not available compiler generate error message like "**Main method not found in class A, please define the main method**".

**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  
Executing all generated .class files:-

D:\ratan>java Test

Hi Ratan

D:\ratan>java A

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

public static void main(String[] args)

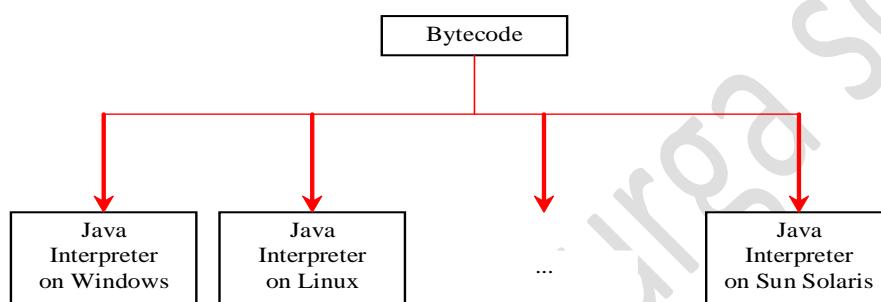
D:\ratan>java B

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

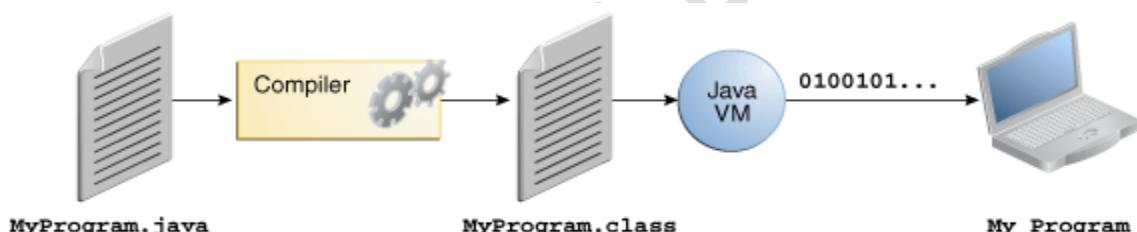
public static void main(String[] args)

D:\ratan>java XXX

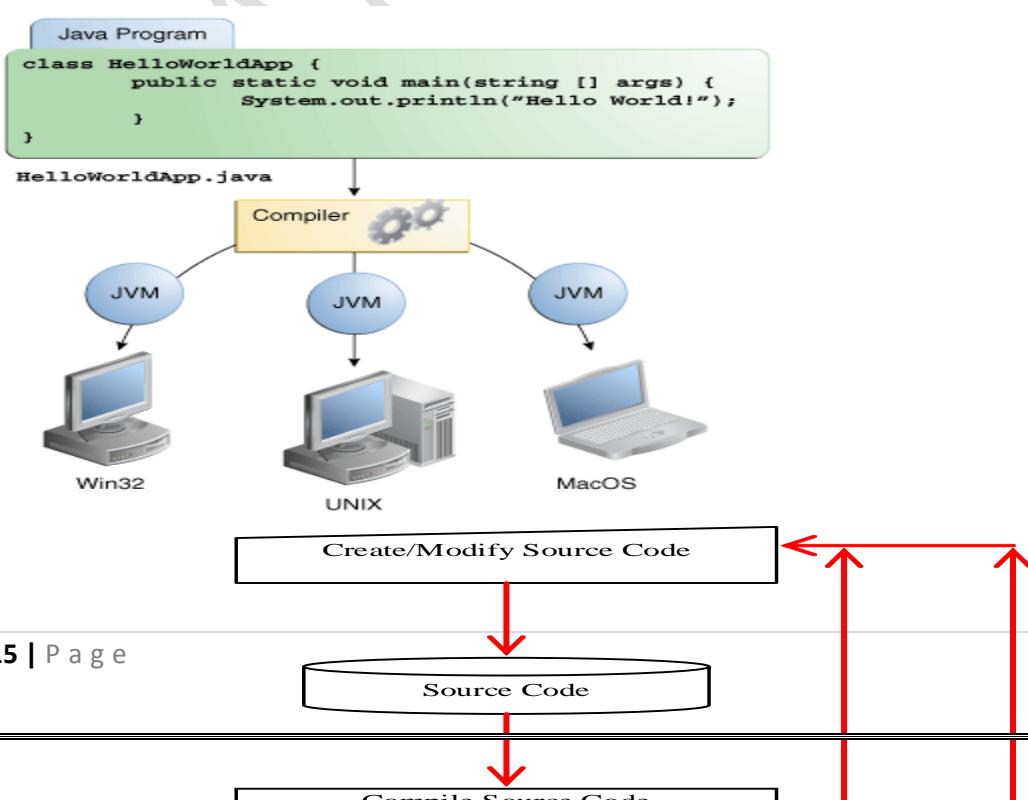
Error: Could not find or load main class XXX



#### Environment of the java programming development:-



#### First program development :-



**Example application :-**

- The default package in java is **java.lang** package it means if we are importing or not by default that package is imported.
- In below example importing classes are optional.

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

**Example Application:-**

The class contains main method is called **Mainclass** and java allowes 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!

**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 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 applicable for predefined library not for user defined library .But it is recommended to follow the coding standards for user defined library also.

### 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

```

### Java Comments :-

- Comments are used to provide detailed description about application.
- comments are non executable code.
- 

There are 3 types of comments.

#### 1) Single line Comments:-

By using single line comments we are providing description about our program within a single line.

Starts with.....>// (double slash)

Syntax:- //description

#### 2) Multi line Comments:-

This comment is used to provide description about our program in more than one line.

Syntax: - /\* .....line-1  
.....line-2  
\*/

#### 3) Documentation Comments:-

al we are using document comment to prepare API(Application programming interface) documents.. We will discuss later chapter.

Syntax: - /\*\* .....line-1  
\* .....line-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

```

**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*

**Java Identifiers:-**

any name in the java program like variable name, classname, methodname, interface name is called identifier.

```

class Test
{
    void add()
    {
        int a=10;
        int b=20;
    }
};

```

Test-----→identifier  
add-----→identifier  
a-----→identifiers  
b-----→identifiers

**Rules to declare identifiers:-**

1. the java identifiers should not start with numbers, it may start with alphabet symbol and underscore symbol and dollar symbol.
  - a. Int abc=10;----→valid
  - b. Int 2abc=20;----→not valid
  - c. Int \_abc=30;----→valid
  - d. Int \$abc=40;----→valid
  - e. Int @abc=50;---→not valid
2. The identifier will not contains symbols like + , - , . , @ , # , \* .....
3. The identifier should not duplicated.

`class Test`

```

    {
        void add()
        {
            int a=10;
            int a=20; 
        }
    }

```

the identifier should not be duplicated.

```

        }
    };
4. In the java applications it is possible to declare all the predefined class names and predefined
   interfaces names as a identifier. But it is not recommended to use.
class Test
{
    public static void main(String[] args)
    {
        int String=10;           //predefind String class
        int Serializable=20;     //predifed Seriaiable Interface
        float Exception=10.2f;   //predefined Exception class
        System.out.println(String);
        System.out.println(Serializable);
        System.out.println(Exception);
    }
};

```

**java flow control Statements:-**

There are three types of flow control statements in java

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

**1. Selection Statements**

- a. If
- b. If-else
- c. switch

**If syntax:-**

```

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

```

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

**Ex-1:-**

```

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

```

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

```

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

```

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

```

#### **Switch statement:-**

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

#### **Syntax:-**

```

switch(argument)
{
    case label1 :    sop(" ");break;
    case label2 :    sop(" ");break;
    |
    |
    default   :    sop(" ");      break;
}

```

#### **Eg-1:Normal input and normal output.**

```

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

```

```

        case 30:System.out.println("samantha");      break;
    default:System.out.println("ubanu");           break;
}
}
}

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

Unicode values a-97 A-65

```

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

```

```

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

```

**Ex -10: internal conversion of integer to character.**

```

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

```

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

```

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

```

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

```

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

```

}

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

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

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

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

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

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

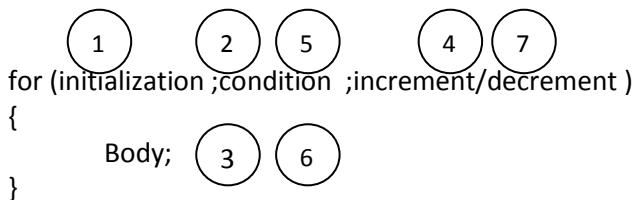
**Iteration Statements:-**

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

- 1) For      2) while    3) do-while

**for syntax:-**

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

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

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

**Initialization part:-**

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

**With out for loop**

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

**Initialization:-****Ex1: Inside the for loop initialization part is optional.**

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

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

```
class Test
```

```

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

```

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

```

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

```

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

```

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

```

#### Conditional part:-

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

```

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

```

#### Increment/decrement:-

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

```

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

```

```

        {
            System.out.println("Rattaiah");
        }
    }
}

```

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

```

class Test
{
    public static void main(String[] args)
    {
        for (int i=0;i<10;System.out.println("aruna"),System.out.println("nagalakshmi"))
        {
            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.**

**Example :-**

```

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

```

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

```

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

```

**ex:- compiler able to identify the unreachable Statement.**

```

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

```

**While loop:-****Syntax:-**

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

**Ex :-**

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

**Ex :- compilation error unreachable statement**

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

**Do-While:-**

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

**Syntax:-do**

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

**Example :-****class Test**

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

```

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

```

**Example :- 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 :-**

```

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

```

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

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

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

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

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

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

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

```

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

```

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

```

class Test
{
    public static void main(String[] args)
    {
        if (true)
        {
            System.out.println("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 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>
<b>byte</b>	<b>1</b>	<b>-128 to 127</b>	<b>0</b>
<b>short</b>	<b>2</b>	<b>-32768 to 32767</b>	<b>0</b>
<b>int</b>	<b>4</b>	<b>-2147483648 to 2147483647</b>	<b>0</b>
<b>long</b>	<b>8</b>	<b>-9,223,372,036,854,775,808 to 9,223,372,036,854,775,8070</b>	
<b>float</b>	<b>4</b>	<b>-3.4e38 to 3.4e</b>	<b>0.0</b>
<b>double</b>	<b>8</b>		<b>0.0</b>
<b>char</b>	<b>2</b>	<b>0 to 6553</b>	<b>single space</b>
<b>Boolean</b>	<b>no-size</b>	<b>no-range</b>	<b>false</b>

**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
System.out.println(10);        //invalid

```

**Note :-**

- To represent numeric values (10,20,30...etc) use **byte,short,int,long**.
- To represent point 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**.

**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
```

**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

**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.

**To overcome above problem use "f" constant to represent float value.**

```
float f=10.5f; //valid
System.out.println(f);
```

**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 :-**

```

class Test
{
    public static void main(String[] args)
    {
        System.out.println("information about byte data type");
        System.out.println("byte size="+Byte.SIZE);
        System.out.println("byte min value="+Byte.MIN_VALUE);
        System.out.println("byte max value="+Byte.MAX_VALUE);

        System.out.println("information about int data type");
        System.out.println("int size="+Integer.SIZE);
        System.out.println("int min value="+Integer.MIN_VALUE);
        System.out.println("int max value="+Integer.MAX_VALUE);
    }
}

```

**Java Variables:-**

- Variables are used to hold the constant values by using these values we are achieving project requirements/functionality.
- While declaring variable must specify the type of the variable by using data types.
- Variables are also known as **fields** of a class or **properties** of a class.

**Note :- All variables must have a type. You can use primitive types such as int, float, boolean, etc. Or you can use reference types, such as strings, arrays, or objects.**

*Variable declaration is composed of three components in order,*

- 1) Zero or more modifiers.
- 2) The variable type.
- 3) The variable name.

**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);
    }
}

```

- }
- ❖ We are able to access local variable 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.

```
void m1() //memory allocated when method starts
{
    //local variable
    int a=10;
    int b=20;
} //memory released when method completed
```

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

- ❖ The variables which are declare inside a class and outside of methods those variables are called instance variables.
- ❖ Instance variables are visible in all **methods and constructors** of a particular class.
- ❖ Instance variables are also known as **non-static** fields. And it is having global visibility.
- ❖ For instance variables memory allocated during object creation time and memory released when object is destroyed.
- ❖ Instance variables are stored in heap memory.

#### Areas of java language:-

There are two types are in java.

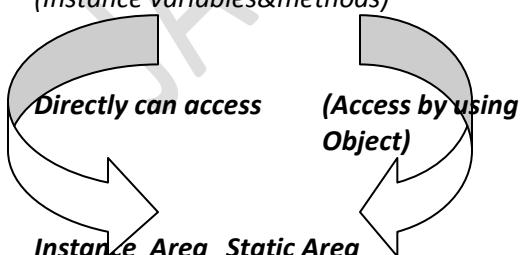
- 1) *Instance Area.*
- 2) *Static Area.*

#### Instance Area:-

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

#### Instance variable accessing:-

(Instance variables & methods)



#### Static Area:-

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

#### 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 in main method
    {
        //instance area
        System.out.println(a);
        System.out.println(b);
    }
}//main ends
}//class ends

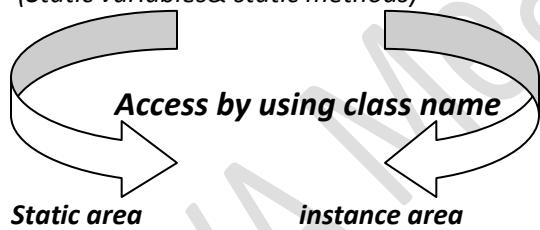
```

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

- ❖ The variables which are declared inside the class and outside of the methods with static modifier is called static variables.
- ❖ Static variables are visible all methods and constructors of a particular class.
- ❖ Static variables memory allocated at the time of .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 in main method
    {
        System.out.println(Test.a);
    }
}

```

```

        System.out.println(Test.b);
    }
}

```

**Calling of static variables:-**

We are able to access the static members inside the static area in three ways.

- ✓ Directly possible.
- ✓ 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();             //3-way(By using reference variable)
        System.out.println(t.a);
    }
}

```

**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
{
    // instance variables
    int a=10;
    int b=20;
    static int c=30; //static variables
    static int d=40;
    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)
    {
        //access the instance variables by using object
        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)
    {
        //access the static variable by using class Names
        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 localvariables 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)
    {
        //local variables (access directly)
        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);
```

#### Class Vs Object:-

- Class is a logical entity it contains logics where as 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.
- Civil engineer based on blue print of house it is possible to create multiple houses in different places but every house required some area.
- 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 factory method.
  - By using deserialization....etc

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

- We will discuss object creation in detailed in constructor concept.

#### Instance vs. Static variables:-

- ❖ For the instance variables the JVM will create separate memory for each and every object it means separate instance variable value for each and every object.
- ❖ For the 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;
}

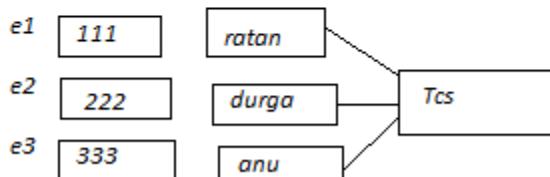
```



```

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

```



**Note 1 :-** The variables which are declared inside the method or constructor or blocks those variables are called local variables and we are able to access(usage) local variables only inside the method or constructor or blocks , for the local variables memory is allocated when method starts and memory is destroyed when method ends and these variables are stored in stack memory & call the local variables directly .

**Note 2:-** The variables which are declared inside the class and outside of the method those variables are called instance variables and we are able to access (usage) instance members inside the class,for the instance variables memory is allocated during object creation and memory destroyed when object is destroyed & these variables are stored in heap memory & call the instance members (variables & methods) by using object.

**Note 3 :-** The variables which are declared inside the class and outside of the method with static modifier those variables are called static variables & we are able to access (usage) static members inside the class& for the static variables memory is allocated during .class loading and memory destroyed .class file unloading & access the static members(variables & methods) by using class Name.

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

```

class Test
{
    int s=10;
    int a,b,c;
    int x,y,z=10;
    int i=10,j=20,k;
    static int p=100,q=200,r=300;
    public static void main(String[] args)
    {
        Test t = new Test();
        System.out.println(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(Test.p+" "+Test.q+" "+Test.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><u>where declared</u></b>	inside method or Constructor or block.	inside the class outside Of methods	inside the class outside of methods.
<b><u>Use</u></b>	within the method	inside the class all the methods and constructors.	inside the class all Methods& constructors.
<b><u>When memory allocated</u></b>	when method starts	when object created	when .class file loading
<b><u>When memory destroyed</u></b>	when method ends.	When object destroyed	when .class unloading.
<b><u>Initial values</u></b>	none, must initialize the value before first use.	default values are Assigned by JVM.	default values are Assigned by JVM.
<b><u>Relation with Object</u></b>	no way related to object.	for every object one copy Of instance variable created	for all objects one copy is created. It means memory.
<b><u>Accessing</u></b>	directly possible.	By using object name. <b>Test t = new Test();</b> <b>System.out.println(t.a);</b>	by using class name. <b>System.out.println(Test.a);</b>
<b><u>Memory</u></b>	stored in stack memory.	Stored in heap memory	non-heap memory.

**+ operator:-**

- ✓ One operator with multiple behaviors is called operator over loading but java is not supporting operator overloading concept and only one overloaded operator in java is +
- If two operands are integers then + perform addition.

- If at least one operand is String then + 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;
        int b=20;
        int 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);
    }
}

```

#### Java.util.Scanner(Dynamic Input):-

1. Scanner class present in **java.util** package and it is introduced in 1.5 version.
2. Scanner class is used to take dynamic input from the keyboard.

```

Scanner s = new Scanner(System.in);
to get int value      --->  s.nextInt()
to get float value   --->  s.nextFloat()
to get byte value    --->  s.nextByte()
to get String value  --->  s.next()
to get single line   --->  s.nextLine()
to close the input stream ---> s.close()

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Scanner s=new Scanner(System.in);      //used to take dynamic input from keyboard
        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
    }
}

```

### **Java Methods (behaviors):-**

- ❖ Methods are used to write the business logics of the project.
- ❖ Coding convention of method is method name starts with lower case letter if method contains more than one word then every inner word starts with uppercase letter.

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

*There are two types of methods*

1. Instance method
2. Static method

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

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

### **Instance method :-**

```

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

```

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

### **Syntax:-**

```

Void m1() { } //instance method
Test t = new Test();

```

```
Objectnameinstancemethod(); //calling instance method
t.m1();
```

### **static method:-**

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

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

### **Syntax:-**

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

### **Syntax:-**

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

<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()**  
**Private int m2(int a,int b)**

### **Method Signature:-**

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

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

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

**Every method contains two parts.**

1. **Method declaration**
2. **Method implementation (logic)**

**Ex:-**      **void m1()**                  ----->   **method declaration**  
{         **Body (Business logic);**   ----->   **method implementation**  
}

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

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

```

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

```

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

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

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

```

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

```

**Example-3 :- static methods without parameters.**

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

```

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

```

```

    {
        Test.m1();      //call the static method by using class name
        Test.m2();      //call the static method by using class name
    }
}

```

**Example -4 :-static methods with parameters.**

```

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

```

**Example 6:-For java methods it is possible to provide Objects as a parameters(in real time project level).****Case 1:- project code at student level.**

```

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 using object
        Y y = new Y();
        Test.m2(10,y);              //calling of static method by using class-name
    }
}

```

**Case 2: project code at realtime project level**

```

class X{}
class Emp{}
class Y{}
class Test
{
    void m1(X x ,Emp e)//taking objects as a parameter
    {
        System.out.println("m1 method");
    }
    static void m2(int a,Y y) //taking objects as a parameter
    {
        System.out.println("m2 method");
    }
    public static void main(String[] args)
    {
        new Test().m1(new X(),new Emp());
        Test.m2(10,new Y());
    }
}

```

**Example-7 :- method vs. 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);           //by default 10 is int value
        t.m2((short)20);          //by default 20 is int value
        t.m3(30);
        t.m4(40);
        t.m5(10.6f);             //by default 10.6 value is double
        t.m6(20.5);               t.m7('a');                      t.m8(true);
    }
}

```

```

        }
    }

Example-8:-method calling
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-9:-**

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

```

class Test
{
    void m1() { System.out.println("hi m1-method"); }
    m2() { System.out.println("hi m2-method"); } //return type is mandatory
    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1();
        t.m2();
    }
}

```

**Example-10 :-**

- Inside the java 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 “m1() is already defined in Test ”
- Java class not allowed Duplicate methods.
- It is possible to write,

```

void m1()
Void m1(int a)

```

But the above method signatures are different it is method overloading concept.

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

#### Example-11 :-

- Declaring the class inside another class is called inner classes, java supports inner classes.
- Declaring the methods inside another 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-12 :- methods vs return type.

1. Every functionality is able to return some functionality return value just like acknowledgement.  
Ex : when we applied for driving license then after one month we will receive ID card.
2. In java every method is able to return some return value (int , char , String.....) if we are not interested then return nothing by using **void** return type.
3. If the method is having return type other than void at that situation must return the value by using **return** keyword otherwise compiler will generate error message “missing return statement”

*Ex : below syntax invalid because method must return int value by using return statement.*

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

*Ex :- the below example is valid because it is returning int value by using return statement.*

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

4. Inside the method we are able to use only one return statement(except flow control statement) that must be last statement of the method otherwise compiler will generate error message “**unreachable statement**”.

*Ex : the below example is invalid because return statement is must be last statement.*

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

*Ex : the below example valid because return statement is last statement .*

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

5. Every method is able to return value but holding (storing) that return value is optional ,but it is recommended to hold the value.

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

**Example-13:- 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

**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

**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

```

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

```

class Person
{
    void eat(){System.out.println("person takes 4-idles");}
}

class Ratan
{
    void eat(){System.out.println("ratan takes 10-idles");}
}

class RatanKid
{
    void eat(){System.out.println("person takes 2-idles");}
}

class Test
{
    Person m1()
    {
        System.out.println("m1 method");
        Person p = new Person();
        return p;
    }

    Ratan m2()
    {
        System.out.println("m2 method");
        Ratan r = new Ratan();
        return r;
    }

    RatanKid m3()
    {
        System.out.println("m3 method");
        RatanKid k = new RatanKid();
        return k;
    }

    public static void main(String[] args)
    {
        Test t =new Test();
        Person p = t.m1();
        p.eat();
        Ratan r = t.m2();
        r.eat();
        RatanKid k = t.m3();
        k.eat();
    }
};

```

**The project level used code:-**

```

Person m1()
{
    System.out.println("m1 method");
    return new Person();
}
Ratan m2()
{
    System.out.println("m2 method");
    return new Ratan();
}
RatanKid m3()
{
    System.out.println("m3 method");
    return new RatanKid();
}

```

**Example 15:**

Note :- **This keyword representing current class objects.**

In java method is able to return current class object return value in two ways.

- 1) By creation of object.
- 2) Return **this** keyword because it is representing current class object.

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

```

class Test
{
    Test m1()      //first approach to return same class(Test) object
    {
        return new Test();
    }
    Test m2()      //second approach to return same class(Test) object
    {
        return this;
    }
    public static void main(String[] args)
    {
        Test t = new Test();      //it creates object of Test class
        System.out.println(t.getClass());
        Test t1 = t.m1();        //m1() method return Object of Test class
        System.out.println(t1.getClass());
        Test t2 = t.m2();        //m2() method return Object of Test class
        System.out.println(t1.getClass());
    }
};

```

**Example 16 :- Template method:-**

- Let Assume to complete your task you must call four methods at that situation you must remember number of methods and order of calling.

- To overcome above limitation take one 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 part");}
    void product()       { System.out.println("product part"); }
    void selection()     { System.out.println("selection part"); }
    void billing()        { System.out.println("billing part"); }

    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)
    {
        //normal approach
        Test t = new Test();
        t.customer();          t.product();           t.selection();         t.billing();

        //by using template method
        Test t1 = new Test();
        t1.deliveryManager(); //this method is calling four methods to complete our task.
    }
};

```

**Example 17:- Method recursion** A method is calling itself during execution is called recursion.

**Example 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);
    }
};

```

**Example 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);
    }
};

```

```

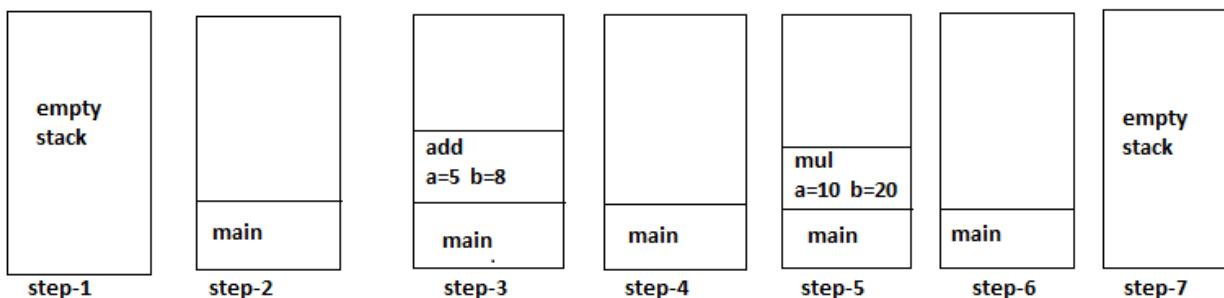
    }
};
```

**Example 18 :- Stack Mechanism:-**

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

```

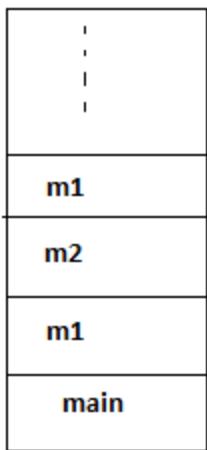
class Test
{
    void add(int a,int b)
    {
        System.out.println(a+b);
    }
    void mul(int a,int b)
    {
        System.out.println(a+b);
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.add(5,8);
        t.mul(10,20);
    }
}
```



**Example 18:-we are getting 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();
    }
}
```

**this keyword:-**

this keyword is holding current class reference variable and it is used to represent,

- a. Current class variables.
- b. Current class methods.
- c. Current class constructors.

**Current class variables:-****This keyword not required:-**

```
class Test
{
    //instance variables
    int a=100;
    int b=200;
    void add(int i,int j)//local variables
    {
        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);
    }
}

```

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

#### This keyword required:-

```

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

```

In bove example intstance variables and local variables having same name at that situation we are able to print local variables directly but to represent instance variables use **this** keyword.

#### Conversion of local variables to instacevariables:-

##### This keyword not Required:-

```

class Test
{
    int i, j; //instance variables
    void values(int val1,int val2)//local variables
    {
        //conversion of local variables to instance variables (passing local variable
        //values to instance variables)
        i=val1;
        j=val2;
    }
    void add(){System.out.println(i+j);}
    void mul(){System.out.println(i*j);}
}

```

```

public static void main(String[] args)
{
    Test t=new Test();
    t.values(100,200);
    t.add();      t.mul();
}
//end main
}//end class

```

In above example local variables and instance variables having different names hence this keyword not required.

**This keyword Required:-**

```

class Test
{
    //instance variables
    int val1;
    int val2;
    void values(int val1,int val2)//local variables
    {
        //printing local variables
        System.out.println(val1);
        System.out.println(val2);
        //conversion of localvariables to instance variables (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();
    }
}
//end main

```

In above example local variables and instance variables having same names so while conversion to represent instance variable use this keyword.

**Current class method calling:-**

Ex:- to call the current class methods this keyword optional hence the both examples are same.

```

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

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

```

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

#### CONSTRUCTORS:-

##### Object creation syntax:-

Test t = new Test();  
Test ---> class Name  
t -----> Reference variables  
= -----> assignment operator  
New -----> keyword used to create object  
Test () -----> constructor  
; -----> statement terminator

1. When we create new instance (Object) of a class using new keyword, a constructor for that class is called.
2. Constructors are used to initialize the instance variables of a class

**New :-**

- new keyword is used to create object in java.
- Different approaches are there to create objects like
  - By using instance factory method.
  - By using static factory method
  - By using pattern factory method
  - By using new operator.
  - By using Deserialization .
  - By using newInstance() method.
  - By using clone() method.....etc
- 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) Constructor is able to take parameters.
- 3) Constructor not allowed explicit return type (return type declaration not possible).

There are two types of constructors,

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

**Default Constructor:-**

- 1) If we are not writing constructor for a class then compiler generates one constructor for you that constructor is called default constructor. And it is not visible in code.
- 2) Compiler generates Default constructor inside the class when we are not providing any type of constructor (0-arg or parameterized).
- 3) The compiler generated default constructor is always **0-argument** constructor with **empty implementation (empty body)**.

**Application before compilation :-**

```
class Test
{
    void m1() { System.out.println("m1 method"); }
    public static void main(String[] args)
    {
        //at object creation time 0-arg constructor executed
        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 "**Test()**" constructor inside the class since not there hence compiler generate default constructor at the time of compilation.

**Application after compilation :-**

```

class Test
{
    void m1()      { System.out.println("m1 method"); }
    //default constructor generated by compiler
    Test()
    {
    }
    public static void main(String[] args)
    {//object creation time 0-arg constructor executed
        Test t = new Test();
        t.m1();
    }
}

```

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

**Example-3:-**

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

```

class Test
{
    Test(int i)
    {
        System.out.println(i);
    }
    Test(int i, String str)
    {
        System.out.println(i);
        System.out.println(str);
    }
    public static void main(String[] args)
    {Test t1=new Test(); // in this line compiler is searching 0-arg cons but not available
     Test t2=new Test(10);
     Test t3=new Test(100, "rattaiah");
    }
}

```

**Example-2:- default constructor execution vs. user defined constructor execution****Case 1:- 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)
    {
        // during object creation 0-arg cons executed then values are assigned
        Employee e1 = new Employee();
        e1.display();
    }
}
D:\morn11>javac Employee.java
D:\morn11>java Employee
****Employee details****
Employee name :-->null
Employee eid :-->0
Employee sal :-->0.0

```

**Note:** - in above example during object creation time default constructor is executed with empty implementation and initial values of instance variables (default values) are printed .

#### Case 2:- user defined o-argument constructor execution process.

```

class Employee
{
    //instance variables
    int eid;
    String ename;
    double esal;
    Employee()//user defined 0-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)
    {
        // during object creation 0-arg cons executed then values are assigned
        Employee e1 = new Employee();
        e1.display(); //calling display method
    }
}

```

#### Compilation & execution process:-

```
D:\morn11>javac Employee.java
D:\morn11>java Employee
****Employee details****
Employee name :-->ratan
Employee name :-->111
Employee name :-->60000.0
```

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

#### The problem with above approach:-

When we create two employee objects but every object same values are initialized.

```
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 just use parameterized constructor to initialize different values.

#### Case 3:- user defined parameterized constructor execution.

##### User defined parameterized constructors:-

- Inside the class if the default constructor is executed means the 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 that constructor is able to initialize the values only for single object.
- To overcome above limitation declare the parameterized constructor and pass the different values during different objects creation.
- Parameterized constructor is nothing but the constructor is able to parameters.

##### 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();
        Employee e3 = new Employee(333, "Sravya", 80000);
        e3.display();
    }
}

```

**Note:** - by using parameterized constructor we are able to initialize values to instance variables during object creation and it is possible to initialize different values to different objects during object creation time.

**Note :-** the main objective of constructor is initialize some values to instance variables during object creation time.

#### Example :-

- Constructors are performing following operations
  - Constructors are useful to initialize some user provided values to instance variables during object creation.
  - Constructors are used to write the functionality of project that functionality is executed during object creation.

- Inside the class it is possible to declare multiple constructors.

```

class Test
{
    Test() //user defined 0-arg constructor
    {
        System.out.println("0-arg cons logics");
    }
    Test(int a, int b) //user defined parameterized constructor

```

```

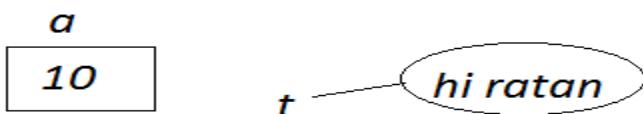
{
    System.out.println("2-arg cons logics");
}
void m1() { System.out.println("m1 method"); }
public static void main(String[] args)
{
    Test t1 = new Test();           t1.m1();
    Test t2 = new Test(10,20);      t2.m1();
}
}

Example-4:- constructors vs all data-types
class Test
{
    Test(byte a) { System.out.println("Byte value-->" + a); }
    Test(short a) { System.out.println("short value-->" + a); }
    Test(int a) { System.out.println("int value-->" + a); }
    Test(long a) { System.out.println("long value is-->" + a); }
    Test(float f) { System.out.println("float value is-->" + f); }
    Test(double d) { System.out.println("double value is-->" + d); }
    Test(char ch) { System.out.println("character value is-->" + ch); }
    Test(boolean b) { System.out.println("boolean value is-->" + b); }
    public static void main(String[] args)
    {
        Test t1=new Test((byte)10);
        Test t2=new Test((short)20);
        Test t3=new Test(30);
        Test t4=new Test(40);
        Test t5=new Test(10.5);
        Test t6=new Test(20.5f);
        Test t7=new Test('a');
        Test t8=new Test(true);
    }
}

```

**Example :-**

- *a* is variable of primitive type such as int,char,double,boolean...etc
- *t* is reference variable & it is the memory address of object.
- Reference variable used to hold particular object & class decides object creation.



```

class Test
{
    Test(String str){}
    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("hi ratan");
    System.out.println(t);
}
}

```

**This keyword :-****To call Current class constructor use this keyword**

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

**Example-1:-****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)
    {
        Test t = new Test();    //at object creation time 0-arg constructor executed
    }
}

```

**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**”.

Constructor calling 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");
}

Test(int a)
{
    this(10,20); //current class 2-argument constructor calling
    System.out.println(a);
}

```

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

```

Test()
{
    System.out.println("0 arg");
}

```

```

        this(10); //current class 1-argument constructor calling
    }
    Test(int a)
    {
        System.out.println(a);
        this(10,20); //current class 2-argument constructor calling
    }
}

```

**Example-3:-**

1. Constructor calling must be first statement in constructor it means this keyword must be first statement in constructor.
2. In java One constructor is able to call only one constructor at a time it is 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");
}

```

**Note :-**

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 :-** in java object creation done in 2-ways.

- 1) Named object (having reference variable)      **Test t = new Test();**
- 2) Nameless object (without reference variable)      **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();
}
}

```

**Example :** Two 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;
        ::::::::::::::::::::
        t1=new Test();
        t1.m1();
    }
}

```

**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)    //local variables
    {
        //conversion [passing local variable values to instance variables]
        this.sid=sid;           this.sname=sname;           this.smarks=smarks;
    }
    //method assigning values to instance variables
    void assign(int sid,String sname,int smarks) //local variable
    {
        //conversion
        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();
}
}

Example :- By using constructors copy the values of one object to another object.
class Student
{
    //instance variables
    int sid;
    String sname;
    int smarks;
    Student(int sid,String sname,int smarks)
    {
        //conversion [converting localvariable values to instance variables]
        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();
    }
}

```

**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>	<i>It is initializing variable when We call that method.</i>	<i>It is initializing variable during object creation.</i>
<b>3) Return type</b>	Return type not allowed Even void.	<i>It allows all valid return Types(void,int,Boolean...etc)</i>

<b>4)Name</b>	<b>Method name starts with lower Case &amp; every inner word starts With upper case.</b> Ex: <code>charAt()</code> , <code>toUpperCase()</code> ....	<b>Class name and constructor name must be matched.</b>
<b>5)types</b>	<b>a) instance method b)static method</b>	<b>a)default constructor b)user defined constructor</b>
<b>6)inheritance</b>	<b>methods are inherited</b>	<b>constructors are not inherited.</b>
<b>7)how to call</b>	<b>To call the methods use method Name.</b>	<b>to call the constructor use <code>this</code> keyword.</b>
<b>8)able to call how many Methods or constructors</b>	<b>one method is able to call multiple methods at a time.</b>	<b>one constructors able to Call only one constructor at a time.</b>
<b>9)this</b>	<b>to call instance method use this Keyword but It is not possible to call static method.</b>	<b>To call constructor use this keyword but inside constructor use only one this statement.</b>
<b>10)Super</b>	<b>used to call super class methods.</b>	<b>Used to call super class constructor</b>
<b>11)Overloading</b>	<b>it is possible to overload methods</b>	<b>it is possible to overload cons.</b>
<b>12)compiler generate Default cons or not</b>	<b>yes</b>	<b>does not apply</b>
<b>13)compiler generate Super keyword.</b>	<b>yes</b>	<b>does not apply.</b>

**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();
    }
}

```

**Instance Blocks:-**

- Instance blocks 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 instance blocks are executed.

**Example 1:-**

```

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

```

**Example 2:-**

```

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)
    {
        Test t1 = new Test();
        Test t2 = new Test();
        Test t3 = new Test(10);
    }
}

```

**Example 3:-**

```

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)
{
    Test t1 = new Test();
}
}

```

**Example 1:-**

```

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)
    {
        Test t = new Test();
    }
}

```

D:\morn11>java Test

**instance block**

**m1() method called by variable**

**example :-**

```

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

```

D:\morn11>java Test

**m1() method called by variable**

**instance block**

**example :-**

- instance blocks are used to initialize instance variables during object creation but before constructor execution.

```
class Emp
{
    //instance blocks
    int eid;           //0
    String ename;      //null
    double esal;       //0.0
    //instance block used to initialize instance variables
    {
        ename="ratan";
        eid=111;
        esal=20000;
    }
    void disp()
    {System.out.println("emp name="+ename);
     System.out.println("emp id="+eid);
     System.out.println("emp sal="+esal);
    }
    public static void main(String[] args)
    {
        Emp e1 = new Emp();           //default constructor & instance block is executed
        e1.disp();
    }
};
```

**static block:-**

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

**Example :-**

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

**Example :-**

```
class Test
{
    static{System.out.println("static block-1");} //static block
    static{System.out.println("static block-2");} //static block
    {System.out.println("instance block-1");}      //instance block
    {System.out.println("instance block-2");}      //instance block
    Test(){ System.out.println("0-arg constructor"); } //0-arg constructor
    Test(int a){ System.out.println("1-arg constructor"); } //1-arg constructor
```

```

public static void main(String[] args)
{
    Test t1 = new Test(); //instance block & constructor executed
    Test t2 = new Test(10); //instance blocks & constructor executed
}
}

D:\morn11>java Test
static block-1
static block-2
instance block-1
instance block-2
0-arg constructor
instance block-1
instance block-2
1-arg constructor

```

**Example:-**

```

class Test
{
    //instance variables
    int a=10,b=20;
    //static variables
    static int c=30,d=40;
    //instance method
    int m1(int a,int b)//local variables
    {
        System.out.println(a+"---"+b);
        return 10;
    }
    //static method
    static String m2(boolean b) //local variables
    {
        System.out.println(b);
        return "ratan";
    }
    Test(int a) //constructor with 1-arg
    {
        System.out.println("1-arg constructor");
    }
    Test(int a,int b) //constructor with 2-arg
    {
        System.out.println("2-arg constructor");
    }
    {System.out.println("instance block-1");} //instance block
    {System.out.println("instance block-2");} //instance block
    static {System.out.println("static block-1");} //static block
    static {System.out.println("static block-2");} //static block
    public static void main(String[] args)
    {
        //Test object created with 1-arg constructor
        Test t1 = new Test(10); //1-arg constructor & instance blocks executed
    }
}

```

```

//Test object created with 2-arg constructor
Test t2 = new Test(100,200);    //2-arg constructor & instance blocks executed
//printing instance variables by using Object name
System.out.println(t1.a);
System.out.println(t1.b);
//printing static variables by using class name
System.out.println(Test.c);
System.out.println(Test.d);
//instnace method calling by using object name
int x = t1.m1(1000,2000);
System.out.println("m1() method return value:-"+x);           //printing return value
//static method calling by using class name
String y = Test.m2(true);
System.out.println("m2() method return value:-"+y);           //printing return value
}
;

```

**Practical example:-**

```

class A
{
}
class B
{
}
class Test
{
    2-instance varaibles
    2-static variables
    2-instance methods
        1-method --->2-arg(int,char) --->return-type-->String
        2 method ---->1-arg(boolean) ---->Test
    2-static methods
        1static method--->2-arg(Aobj,Bobj) ---->A
        2 static mentod --->1-arg(double)-->int
    public static void main(String[] args)
    {
        print instance var
        print static var
        call instance methods
        call static methods
    }
};
class A
{};
class B
{};
class Test

```

```

{
    //instance variables
    int a=10;
    int b=20;
    //static variables
    static int c=30;
    static int d=40;
    String m1(int a,int b)
    {
        System.out.println("m1 method");
        return "ratan"; //returning String value
    }
    Test m2(int a)
    {
        System.out.println("m2 method");
        return this; //returning current class object
    }
    static A m3(A a,B b) //method is taking objects as a input values
    {
        System.out.println("m3 method");
        A a1 = new A();
        return a1; //returning A class object
    }
    static int m4(int a)
    {
        System.out.println("m4 method");
        return 100; //returning integer value
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        //printing instance variables by using object name
        System.out.println(t.a);
        System.out.println(t.b);
        //printing static variables by using class name
        System.out.println(Test.c);
        System.out.println(Test.d);
        String str=t.m1(1,2); //holding m1() method return value(String)
        System.out.println(str); //printing return value
        Test t1 = t.m2(3); //holding m2() method return value(Test)
        A a = new A(); B b = new B();
        //calling m3() method by passing two objects (A,B)
        A a1 = Test.m3(a,b); //holding m3() method return value(A)
        int x = Test.m4(4); //holding m4() method return value(int)
        System.out.println(x); //printing return value
    }
}

```

**Oops concepts:-**

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

**Inheritance:-**

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

**Application code before inheritance**

```
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(){}
};
```

**Application code after inheritance**

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

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

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

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

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

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

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

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

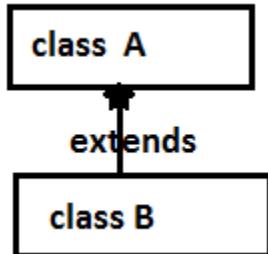
### Types of inheritance :-

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



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

### Example:-

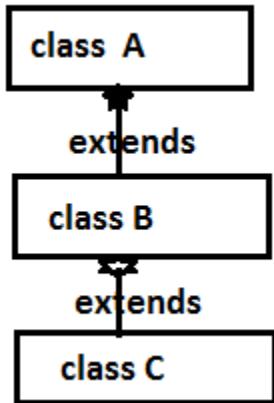
```

class Parent
{
    void property(){System.out.println("money");}
}

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

**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:-**

```

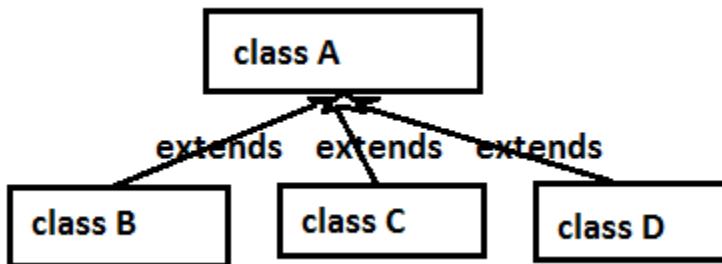
class A
{
    void m1(){System.out.println("m1 method");}
}

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

class B extends A
{
    void m2(){System.out.println("B class");}
}

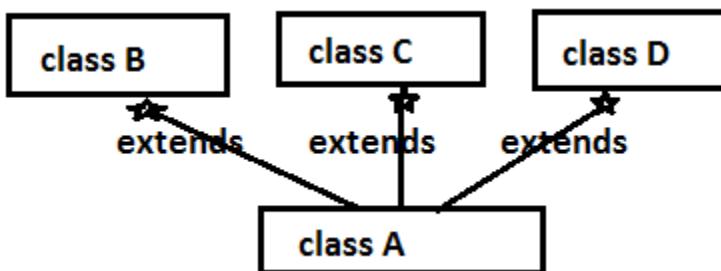
class C extends A
{
    void m2(){System.out.println("C class");}
}

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

**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 inheritance 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 A
{
    void money(){System.out.println("A class money");}
}

class B
{
    void money(){System.out.println("B class money");}
}

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

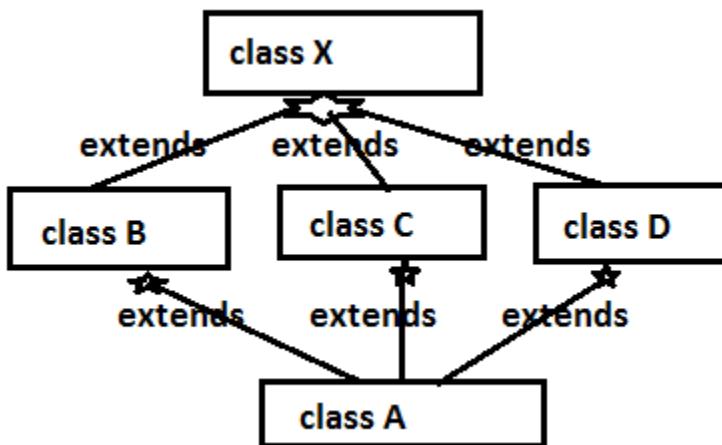
```

    }
};


```

### 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 keyword in the parent class declaration.

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

```

{
};

class Child extends Parent
{
};


```

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

### Note:-

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

### Java.lang.Object class methods :-

```

public class java.lang.Object {
    1) public final native java/lang/Class<?> getClass();
    2) public native int hashCode();
    3) public boolean equals(java.lang.Object);
    4) protected native java.lang.Object clone() throws java.lang.CloneNotSupportedException;
    5) public java.lang.String toString();
    6) public final native void notify();
    7) public final native void notifyAll();
    8) public final native void wait(long) throws java.lang.InterruptedException;
    9) public final void wait(long, int) throws java.lang.InterruptedException;
    10) public final void wait() throws java.lang.InterruptedException;
    11. protected void finalize() throws java.lang.Throwable;
}


```

**Example :-**

In java if we are extending java class that extended class will become Parent class , if we are not extending Object class will become Parent class.

In below example

A class Parent is ----> Object

B class Parent is ---->A

C class Parent is ---->B

class A

```
{     void m1(){}; }
```

class B extends A

```
{     void m2(){}; }
```

class C extends B

```
{     void m3(){}; }
```

In above example A class Parent is Object class

Object class contains ---->**11 methods**

A class contains ---->**12 methods**

B class contains ---->**13 methods**

C class contains ---->**14 methods**

**Instanceof operator:-**

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

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

**Example:-**      **Test t=new Test();**  
**t instanceof Test**

- Whenever we are using instanceof operator the reference variable and class-name must have some relationship either [parent to child] or [child-parent] otherwise compiler generates error message "**inconvertible types**"
- If the relationship is
  - Child – parent returns **true**
  - Parent - child returns **false**

**Example :-**

```
class Animal{ };  

class Dog extends Animal{ };  

class Test  

{   public static void main(String[] args)  

    {  

     //syntax: (ref-ver instanceof class-name);  

     String str="ratan";  

     Animal a = new Animal();  

     Dog d = new Dog();  

     Object o = new Object();  

     System.out.println(a instanceof Object);           //true [child-parent]  

     System.out.println(d instanceof Animal);          //true [child-parent]  

     System.out.println(str instanceof Object);         //true [child-parent]  

     System.out.println(o instanceof Animal);           //false [parent-child]  

     System.out.println(a instanceof Dog);              //false [parent-child]  

//no relationship compilation error :inconvertible types  

//System.out.println(str instanceof Animal);
```

```

    }
}

```

**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:-**

```

class Student
{
    int sid;
    String sname;
    Student(int sid, String sname) //local variables
    {
        //conversion
        this.sid = sid;
        this.sname = sname;
    }
    void disp()
    {
        System.out.println("****student details****");
        System.out.println("student name-->" + sname);
        System.out.println("student name-->" + sid);
    }
};

Class RatanTeacher //teacher uses Student class "association"
{
    public static void main(String[] args)
    {
        Student s1 = new Student(111, "ratan");
        Student s2 = new Student(222, "anu");
        s1.disp();           s2.disp();
    }
};

```

**Example-2:-**

```

class Ratan
{
    void disp(){System.out.println("ratan : corejava");}
};

class Anu
{
    void disp(){System.out.println("anu : advjava");}
};

class Sravya
{
    void disp(){System.out.println("Sravya : ocjp");}
};

class Student //student uses different teachers "association"
{
    public static void main(String[] args)
    {
        Ratan r = new Ratan(); r.disp();
        Anu a = new Anu();      a.disp();
        Sravya d = new Sravya(); d.disp();
    }
};

```

**Aggregation:-**

- Class A has instance of class B.
- 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 -1:-****//Teacher.java**

```
class Teacher
{
    //instance variables
    String tname,sub;
    Teacher(String tname,String sub)//local variables
    {
        //conversion
        this.tname=tname;           this.sub=sub;
    }
};
```

**//Department.java:-**

```
class Department //if we delete department teacher can exists is called aggregation
{
    //instance variables
    int did;
    Teacher t;
    Department(int did ,Teacher t) //local variables
    {
        //conversion
        this.did = did;          this.t = t;
    }
    void disp()
    {
        System.out.println("Department id :--->" + did);
        System.out.println("Teacher details :--->" + t.tname + " --- " + t.sub);
    }
    public static void main(String[] args)
    {
        Teacher x1 = new Teacher("ratan","corejava");
        Department d = new Department(100,x1);
        d.disp();
    }
}
```

**Example -2:****Address.java**

```
class Address
{
    //instance variables
    String country, state;
    int hno;
    Address(String country,String state,int hno)//local variables
    {//passing local variable values to instance variables (conversion)
        this.country = country;      this.state= state;      this.hno = hno;
    }
};
```

**Heroin.java:**

```

class Heroin
{
    //instance variables
    String hname; int hage;
    Address addr;//reference of address class [address class can exists without Heroin class]
    Heroin(String hname,int hage,Address addr)//localvariables
    {
        //conversion of local variables to instance variables
        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);
        //printing address values
        System.out.println("heroin address-->" + addr.country + " " + addr.state + " " + addr.hno)
    }
    public static void main(String[] args)
    {
        //object creation of Address class
        Address a1 = new Address("india", "banglore", 111);
        Address a2 = new Address("india", "mumbai", 222);
        Address a3 = new Address("US", "california", 333);
        //Object creation of Heroin class by passing address object name
        Heroin h1 = new Heroin("anushka", 30, a1);
        Heroin h2 = new Heroin("KF", 30, a2);
        Heroin h3 = new Heroin("AJ", 40, a3);
        h1.display();          h2.display();          h3.display();
    }
}

```

**Example-3:-****Test1.java:-**

```

class Test1
{
    //instance variables
    int a;
    int b;
    Test1(int a,int b)
    {
        this.a=a;
        this.b=b;
    }
};

```

**Test2.java:-**

```

class Test2
{
    //instance variables
    boolean b1;
    boolean b2;
    Test2(boolean b1,boolean b2)
    {
        this.b1=b1;
        this.b2=b2;
    }
};

```

**Test3.java:-**

```

class Test3
{
    //instance variables
    char ch1;
    char ch2;
    Test3(char ch1,char ch2)
    {
        this.ch1=ch1;
        this.ch2=ch2;
    }
};

```

**MainTest.java:-**

```

class Test
{
    //instance variables
    Test1 t1;
    Test2 t2;
    Test3 t3;
    Test(Test1 t1 ,Test2 t2,Test3 t3)//constructor [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');
        Test main = new Test(t,tt,ttt);
        main.display();
    }
};

```

**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
    {
        //conversions
        this.m1=m1;
        this.m2=m2;
        this.m3=m3;
    }
}

```

```

};

//student.java
class Student
{
    //instance variables
    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);
        Marks m2 = new Marks(100, 200, 300);
        Student s1 = new Student(m1, "ratan", 111);
        Student s2 = new Student(m2, "anu", 222);
        s1.display();
        s2.display();
    }
}

```

**Object delegation:-**

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

**Example-1:-**

```

class Test1
{
    //instance variables
    int a=10;
    int b=20;
    static void add() //static method
    {
        Test1 t = new Test1();
        System.out.println(t.a+t.b);
    }
    static void mul() //static method
    {
        Test1 t = new Test1();
        System.out.println(t.a*t.b);
    }
    public static void main(String[] args)
    {
        Test1.add(); //calling static method add()
        Test1.mul(); //calling static method mul()
    }
}

```

**Example-2 :-**

```

class Test1
{ //instance variables
    int a=10;      int b=20;
    static Test1 t = new Test1(); // t is a variable of Test1 type (instance variable)
    static void add()      //static method
    {      System.out.println(t.a+t.b);  }
    static void mul()      //static method
    {      System.out.println(t.a*t.b);  }
    public static void main(String[] args)
    {      Test1.add(); //calling static method add()
          Test1.mul(); //calling static method mul()
    }
}

```

**Example-3:-**

```

class Developer
{
    void task1(){System.out.println("task-1");}
    void task2(){System.out.println("task-2");}
};

class TeamLead
{
    Developer d = new Developer(); //instance variable
    void display1()
    {      d.task1(); d.task2();  }
    void display2()
    {      d.task1(); d.task2();  }
    public static void main(String[] args)
    {      TeamLead t = new TeamLead();
          t.display1(); t.display2();
    }
}

```

**Example -4:-**

```

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:-

Super keyword is holding super class object. And it is representing

1. Super class variables
2. Super class methods
3. Super class constructors

### super class variables calling:-

#### Super keyword not required:-

```
class Parent
{
    int x=10, y=20;//instance variables
}
class Child extends Parent
{
    int a=100,b=200;//instance variables
    void m1(int i,int j)//local variables
    {
        System.out.println(i+j);           //local variables addition
        System.out.println(a+b);           //current class variables addition
        System.out.println(x+y);           //super class variables addition
    }
    public static void main(String[] args)
    {
        Child c = new Child();
        c.m1(1000,2000);
    }//end main
}//end class
```

In above example current class and super class variables having different names so this keyword and super keyword not required.

#### Super keyword required:-

```
class Parent
{
    int a=10,b=20;//instance variables
}
class Child extends Parent
{
    //instance variables
    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)
    {
        Child c = new Child();
        c.m1(1000,2000);
    }
}
```

In bove example sub class and super class having same variable names hence to represent.

- a. sub class variables use this keyword.
- b. Super class variables use super keyword.

**super class methods calling:-****super keyword not required:-**

```

class Parent
{
    void m1(int a) { System.out.println("parent m1()-->" + a); }
}

class Child extends Parent
{
    void m2(int a) { System.out.println("child m1()-->" + a); }

    void m3()
    {
        m1(10);           // parent class m1(int) method calling
        System.out.println("child m2()");
        m2(100);         // child class m2(int) method calling
    }

    public static void main(String[] args)
    {
        Child c = new Child();
        c.m3();
    }
}

```

In above example sub class and super class contains different methods so super keyword not required.

**Superkeyword required:-**

```

class Parent
{
    void m1(int a){ System.out.println("parent m1()-->" + a); }
}

class Child extends Parent
{
    void m1(int a){ System.out.println("child m1()-->" + a); }

    void m2()
    {
        this.m1(10);           //child class m1(int) method calling
        System.out.println("child m2()");
        super.m1(100);         // parent class m1(int) method calling
    }

    public static void main(String[] args)
    {
        Child c = new Child();
        c.m2();
    }
}

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();           //current class method is executed
        super.m1();          //super class method will be executed
    }

    public static void main(String[] args)
    {
        new Child().m3();
    }
}

```

```

    }
};
```

**In above example super class and sub class contains methods with same names( m1() ) at that situation to represent.**

- Super class methods use super keyword.**
- Sub class methods use this keyword.**

#### **super class constructors calling:-**

```

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

#### **Example-1:-To call super class constructor use super keyword.**

```

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)
    {
        Child c = new Child();
    } //end class
} //end main
```

#### **Example-2:-**

**Inside the constructor super keyword must be first statement otherwise compiler generate error message "call to super must be first line in constructor".**

#### **No compilation error:-**

```

Child()
{
    this(10); //current class 1-arg constructor calling(must be first line)
    System.out.println("Child 0-arg constructor");
}

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

**Compilation Error:-**

```

Child()
{
    System.out.println("Child 0-arg constructor");
    this(10); //current class 1-arg constructor calling
}
Child(int a)
{
    System.out.println("child 1-arg constructor--->" + a);
    super(); //super class 0-arg constructor calling (compilition Error)
}

```

**Example-3:-**

Inside the constructor **this** keyword must be first statement and **super** keyword must be first statement hence inside the constructor it is possible to use either **this** keyword or **super** keyword but both at a time not possible.

**No compilation Error:-**

```

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

```

**Compilation Error:-**

```

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

```

**Example-4:-**

1. Inside the constructor (whether it is default or parameterized) if we are not declaring **super** or **this** keyword at that situation compiler generate **super()** keyword at first line of the constructor.
2. If we are declaring **super** keyword at least one constructor compiler is not responsible to generate **super()** keyword.
3. The compiler generated **super** keyword is always 0-argument 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)
    {
        Child c = new Child();
    }
}

```

D:\>java Child

**parent 0-arg constructor**

**Child 0-arg constructor**

**Example-5:-**

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

```

class Parent
{
    //default constructor Parent() {} generated by compiler at compilation time
}
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)
    {
        Child c = new Child();
    }
}

```

**Example-6:-**

*By using below example we are assigning values to instance variable at the time of object creation either the help of parameterized constructor.*

```

class Parent
{
    int a; //instance variable
    Parent(int a)//local variable
    {
        //conversion of local variable to instance variable
        this.a=a;
    }
}
class Child extends Parent
{
    boolean x; //instance variable
    Child(boolean x) //local variable
    {
        super(10); //super class constructor calling
        this.x=x; //conversion of local variable to instance variable
                    //passing local variable value to instance variable
    }
}

```

```

    }
    void display()
    {
        System.out.println(a);
        System.out.println(x);
    }
    public static void main(String[] args)
    {
        Child c = new Child(true);
        c.display();
    }
}

```

**Example-7:-**

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 at compilation time
        System.out.println("Child 0-arg constructor");
    }
    public static void main(String[] args)
    {
        Child c = new Child();
    }
}

```

**Example-8:-**

In below example in child class 1-argument constructor compiler generate super keyword hence parent class 0-argument constructor is executed.

```

class Parent
{
    Parent(){System.out.println("parent 0-arg cons"); }
}
class Child extends Parent
{
    Child()
    {
        this(10); //current class 1-argument constructor calling
        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)
{
    Child c = new Child();
}
};

D:\>java Child
parent 0-arg cons
child 1-arg cons
Child 0-arg constructor
```

**Example-9:-**

Inside the constructor either it is zero argument or parameterized if we are not providing super or this keyword at that situation compiler generate super keyword at first line of constructor.

```
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)
    {
        Child c = new Child();
        Child c1 = new Child(10);
    }
};

D:\>java Child
parent 0-arg cons
Child 0-arg constructor
parent 0-arg cons
child 1-arg cons
```

**Example-10:-**

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

**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)
    {
        Child c = new Child();
    }
}
```

```
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)
    {
        Child c = new Child();
    }
}
```

**Application code after compilation:- ( .class )****Example-11:-**

In below example inside the 1-argument constructor compiler generate super() keyword hence it is executing super class(**Object**)0-argument constructor is executed.

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

```
class Test
{
    Test(int a){
        System.out.println("Test 1-arg cons");
    }
    public static void main(String[] args)
    {
        Test t = new Test(10);
    }
}
```

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

(**Object** class 0-arg constructor executed)

```
class Test extends Object
{
    Test(int a)
    {
        super(); //generated by compiler
        System.out.println("Test 1-arg cons");
    }
    public static void main(String[] args)
    {
        Test t = new Test(10);
    }
}
```

Note 1:- in java if we are extending class that extended class will become super class

Ex :- **class B{ }**  
**class A extends B //B class is Parent of A class**  
**{ }**

Note 2 :- in java if we are not extending any class then **Object** class will become parent of that class.

Ex :- **class A { } //in this Object class is Parent of A class**

**Note:-**

1. Every class in the java programming either directly or indirectly child class of **Object**.
2. Root class for all java classes is **Object** class.
3. The object class present in **java.lang** package

**Example : assignment**

```

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 because first parent class object constructors 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"); } //constructor
    public static void main(String[] args){
        Child c = new Child();
    }
};

```

```
};
```

**Example-2:-**

In below example just before child class instance blocks first parent class instance blocks are executed.

```
class Parent
{
    {System.out.println("parent instance block");}//instance block
    Parent(){System.out.println("parent cons");}//constructor
};

class Child extends Parent
{
    {System.out.println("Child instance block");}//instance block
    Child()
    {
        //super(); generated by compiler
        System.out.println("child 0-arg cons");
    }
    Child(int a)
    {
        //super(); generated by compiler
        System.out.println("child 1-arg cons");
    }
    public static void main(String[] args)
    {
        Child c = new Child();
        Child c1 = new Child(10);
    }
};

D:\>java Child
parent instance block
parent cons
Child instance block
child 0-arg cons
parent instance block
parent cons
Child instance block
child 1-arg cons
class Parent
{
    {System.out.println("parent class ins block");}
    Parent()
    {
        System.out.println("parent class 0-arg cons");
    }
};
class Child extends Parent
{
    {System.out.println("Child class ins block");}
    Child()
    {
        //super(); generated by compiler
        System.out.println("child class 0-arg cons");
```

```

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

```

```

E:\>java Child
parent class static block
child class static block
parent class ins block
parent class 0-arg cons
Child class ins block
child class 0-arg cons
parent class ins block
parent class 0-arg cons
Child class ins block
child class 0-arg cons

```

**Parent class static block:-**

**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.

```

class Parent
{
    static{System.out.println("parent static block");}//static block
}
class Child extends Parent
{
    static{System.out.println("child static block");}//static block
    public static void main(String[] args)
    {
    }
};

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

class Child extends Parent
{
    {System.out.println("child class instance block");}
    Child()
    {
        //super(); generated by compiler
        System.out.println("child class 0-arg cons");
    }
    static {System.out.println("child class static block");}
    public static void main(String[] args)

```

```

    {
        new Child();
    }
};


```

**Example-2:-**

**Note 1:-** instance blocks execution depends on number of object creations but not number of constructor executions. If we are creating 10 objects 10 times constructors are executed just before constructor execution 10 times instance blocks are executed.

**Note 2:-** Static blocks execution depends on .class file loading hence the static blocks are executed only one time for single class.

```

class Parent
{
    static {System.out.println("parent static block");}//static block
    {System.out.println("parent instance block");}//instance block
    Parent(){System.out.println("parent 0-arg cons");}//constructor
};

class Child extends Parent
{
    static {System.out.println("Child static block");}//static block
    {System.out.println("child instance block");} //instance block
    Child()
    {
        //super(); generated by compiler
        System.out.println("Child 0-arg cons");
    }
    Child(int a){
        this(10,20);//current class 2-argument constructor calling
        System.out.println("Child 1-arg cons");
    }
    Child(int a,int b)
    {
        //super(); generated by compiler
        System.out.println("Child 2-arg cons");
    }
    public static void main(String[] args)
    {
        Parent p = new Parent(); //creates object of Parent class
        Child c = new Child(); //creates object of Child class
        Child c1 = new Child(100); //creates object of child class
    }
};

D:\>java Child
parent static block
Child static block
parent instance block
parent 0-arg cons
parent instance block
parent 0-arg cons
child instance block
Child 0-arg cons
parent instance block
parent 0-arg cons

```

*child instance block*  
*Child 2-arg cons*  
*Child 1-arg cons*

### **Polymorphism:-**

- One thing can exhibits more than one form is called polymorphism.
- Polymorphism shows some functionality(method name same) with different logics execution.
- The ability to appear in more forms 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

**[method execution decided at compilation time]**

**Example :- method overloading.**

- 2) Runtime polymorphism /dynamic binding /late binding.

**[Method execution decided at runtime].**

**Example :- method overriding.**

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

- 1) If java class allows two methods with same name but different number of arguments such type of methods are called overloaded methods.
- 2) We can overload the methods in two ways in java language
  - a. By passing different number of arguments to the same methods.  
`void m1(int a){}`  
`void m1(int a,int b){}`
  - b. Provide the same number of arguments with different data types.  
`void m1(int a){}`  
`void m1(char ch){}`
- 3) If we want achieve overloading concept one class is enough.
- 4) It is possible to overload any number of methods in single java class.

### **Types of overloading:-**

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

### **Method overloading:-**

**Example:-**

```

class Test
{
    //below three methods are overloaded methods.
    void m1(char ch)      {System.out.println(" char-arg constructor ");}
    void m1(int i)         {System.out.println("int-arg constructor ");}
    void m1(int i,int j)   {System.out.println(i+j);}
    public static void main(String[] args)
    {Test t=new Test();
     //three methods execution decided at compilation time
     t.m1('a');t.m1(10);t.m1(10,20);
    }
}

```

#### Example :- overloaded methods vs. all data types

```

class Test
{
    void m1(byte a)  { System.out.println("Byte value-->" + a); }
    void m1(short a) { System.out.println("short value-->" + a); }
    void m1(int a)   { System.out.println("int value-->" + a); }
    void m1(long a)  { System.out.println("long value is-->" + a); }
    void m1(float f) { System.out.println("float value is-->" + f); }
    void m1(double d) { System.out.println("double value is-->" + d); }
    void m1(char ch) { System.out.println("character value is-->" + ch); }
    void m1(boolean b) { System.out.println("boolean value is-->" + b); }
    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);
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1((byte)10);           t.m1((short)20);           t.m1(30);           t.m1(40);
        t.m1(10.6f);            t.m1(20.5);             t.m1('a');          t.m1(true);
        t.sum(10,20);           t.sum(100L,200L);
    }
}

```

#### Constructor Overloading:-

The class contains more than one constructors with same name but different arguments is called constructor overloading.

```

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(ch+"----"+i); }
    public static void main(String[] args)
    {   Test t1=new Test(); //zero argument constructor executed.
}

```

```

        Test t2=new Test(10); //one argument constructor executed.
        Test t3=new Test('a',100);//two argument constructor executed.
    }
}

```

**Operator overloading:-**

- One operator with different behavior is called Operator overloading .
- Java is not supporting operator overloading but only one overloaded in java language is '+'.
  - If both operands are integer + perform 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]
    }
}

```

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

- 1) If we want to achieve method overriding we need two class with parent and child relationship.
- 2) The parent class method contains some implementation (logics).
  - a. If child is satisfied use parent class method.
  - b. If the child class not satisfied (required own implementation) then override the method in Child class.
- 3) A subclass has the same method as declared in the super class it is known as method overriding.

The parent class method is called      ==> **overridden method**

The child class method is called      ==> **overriding method**

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

- 1) **While overriding child class method signature & parent class method signatures must be same otherwise we are getting compilation error.**
- 2) **The return types of overridden method & overriding method must be same.**
- 3) **While overriding the methods it is possible to maintains same level permission or increasing order but not decreasing order, if you are trying to reduce the permission compiler generates error message "attempting to assign weaker access privileges ".**
- 4) **You are unable to override final methods. (Final methods are preventing overriding).**
- 5) **While overriding check the covariant-return types.**
- 6) **Static methods are bounded with class hence we are unable to override static methods.**
- 7) **It is not possible to override private methods because these methods are specific to class.**

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.instanceMethod();
    }
};

```

- 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-1 :-method Overriding

```

class Parent //parent class
{
    void property() {System.out.println("money+land+house");}
    void marry() {System.out.println("black girl");} //overridden method
};

class Child extends Parent//child class
{
    void marry() {System.out.println("white girl/red girl");} //overriding method
    public static void main(String[] args)
    {
        Child c=new Child();
        c.property();
        c.marry();
        Parent p=new Parent();
        p.property();
        p.marry();
    }
};

```

#### Covariant return types :-

##### Example 1:-

in below example overriding is not possible because overridden method return type & overriding method return types are not matched.

```

class Parent
{
    void m1(){}
};

class Child extends Parent
{int m1(){}
};

Compilation error:- m1() in Child cannot override m1() in Parent
return type int is not compatible with void

```

##### Example-2:-

- 1) Before java 5 version it is not possible to override the methods by changing its return types .

- 2) From java5 versions onwards java supports support covariant return types it means while overriding it is possible to change the return types of parent class method(overridden method) & child class method(Overriding).
- 3) The return type of overriding method is must be sub-type of overridden method return type this is called covariant return types.

```
class Animal
{
    void m2(){System.out.println("Animal class m2() method");}
    Animal m1()
    {
        return new Animal();
    }
}
class Dog extends Animal
{
    Dog m1()
    {
        return new Dog();
    }
    public static void main(String[] args)
    {
        Dog d = new Dog();      d.m2();
        Dog d1 = d.m1();        // [d.m1() returns Dog object]
        d1.m2();
        Animal a = new Animal();
        a.m2();
        Animal a1 = a.m1();    // [a.m1() returns Animal object]
        a1.m2();
    }
};
```

#### Type-casting:-

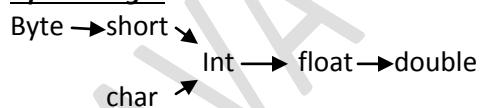
The process of converting data one type to another type is called type casting.

There are two types of type casting

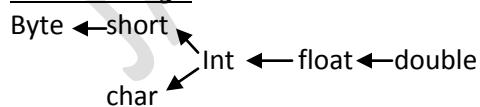
1. Implicit typecasting /widening/up casting
2. Explicit type-casting (narrowing/do)

#### 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");}    //overridden 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 :- importance of converting parent class reference variable into child class object**

```
//let assume predefined class
class ActionForm
{
    void xxx(){}/>predefined method
    void yyy(){}/>predefined method
};
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 :-[ overloading vs. overriding]**

```
class Parent
{
    //overloaded methods
    void m1(int a){System.out.println("parent int m1()-->" + a);}/>overridden method
    void m1(char ch){System.out.println("parent char m1()-->" + ch);}/>overridden method
};

class Child extends Parent
{
    //overloaded methods
    void m1(int a){System.out.println("Child int m1()-->" + a);}/>overriding method
    void m1(char ch){System.out.println("child char m1()-->" + ch);}/>overriding method
    public static void main(String[] args)
    {
        Parent p = new Parent();//it creates object of Parent class
        p.m1(10); p.m1('s'); //10 s [parent class methods executed]
        Child c = new Child();//it creates object of Child class
        c.m1(100); c.m1('a'); //100 a Child class methods executed
        Parent p1 = new Child();//it creates object of Child class
        p1.m1(1000); p1.m1('z'); //1000 z child class methods executed
    }
};
```

```

    }
};
```

**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("Person eat");}
};

class Ratan extends Person
{    void eat(){System.out.println("Ratan eat");}
};

class RatanKid extends Ratan
{    void eat(){System.out.println("RatanKid eat");}
};

class Test
{    public static void main(String[] args)
    {        Person pp = new Person();  //#[creates object of Person class]
        pp.eat();
        Person p = new Ratan();//#[creates object of Ratan class]
        p.eat();                //compile time: Person runtime:Ratan
        Person p1 = new RatanKid();//#[creates object of RatanKid class]
        p1.eat();                //compile time: Person runtime:RatanKid
        Ratan r = new RatanKid();//#[creates object of RatanKid class]
        r.eat();                //compile time: Ratan runtime:RatanKid
    }
};

```

***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
    Child c = (Child)p;
    System.out.println("a values is :--->" + c.a); //1000
}
};

```

**Method overloading:-**

- 1) Method name same & parameters must be different.
  - a. Void m1 (int a) { }
  - b. Void m1(int a,int b) { }
- 2) To achieve overloading one java class sufficient.
- 3) It is also known as Compile time polymorphism/static binding/early binding.

**Method overriding :-**

- 1) Method name same & parameters must be same.
  - a. Void m1(int a){ } //parent class method
  - b. Void m1(int a){ } //child class method
- 2) To achieve overriding we required two java classes with parent and child relationship.
- 3) It is also known as runtime polymorphism/dynamic binding/late binding.

**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.

- The `toString()` method is overridden some classes check the below implementation.
  - In `String` class `toString()` is overridden to return content of `String` object.
  - In `StringBuffer` class `toString()` is overridden to returns content of `StringBuffer` class.
  - In `Wrapper` classes(`Integer`,`Byte`,`Character`...etc) `toString` is overridden to returns content of `Wrapper` classes.

**internal implementation:-**

```
class Object
{
    public String toString()
    {
        return getClass().getName() + '@' + Integer.toHexString(hashCode());
    }
};
```

**Example:-**

**Note :- whenever you are printing reference variable internally `toString()` method is called.**

`Test t = new Test(); //creates object of Test class reference variable is "t"`

**//the below two lines are same.**

```
System.out.println(t);
System.out.println(t.toString());
```

```
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 -2:-**

`toString()` method present in `Object` class but in our `Test` class we are overriding `toString()` method hence our class `toString()` method is executed.

```
class Test
{
    //overriding toString() method
    public String toString()
    {
        return "ratansoft";
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        //below two lines are same
        System.out.println(t);           //Test class toString() executed
        System.out.println(t.toString()); //Test class toString() executed
    }
};
```

**Example-3:- very important**

```
class Student
{
    //instance variables
```

```

String sname;
int sid;
Student(String sname,int sid) //local variables
{
    //conversion
    this.sname = sname;
    this.sid = sid;
}
public String toString() //overriding toString() method
{
    return "student name:->" + sname + " student id:->" + sid;
}
};

class TestDemo
{
    public static void main(String[] args)
    {
        Student s1 = new Student("ratan",111);
        //below two lines are same
        System.out.println(s1);           //student class toString() executed
        System.out.println(s1.toString()); //student class toString() executed

        Student s2 = new Student("anu",222);
        //below two lines are same
        System.out.println(s2);           //student class toString() executed
        System.out.println(s2.toString()); //student class toString() executed
    }
};

```

Example :- overriding of toString() method

```

class Test
{
    //overriding method
    public String toString()
    {
        return "ratnsoft";
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        System.out.println(t);
        System.out.println(t.toString()); // [here overriding toString() executed it means
                                         our class toString() method will be executed]
    }
};

```

In above example overriding toString() method will be executed.

Example :- employee class is not overriding toString()

```

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 static void main(String[] args)
{
    Employee e1 = new Employee("ratan",111,60000);
//whenever we are printing reference variables internally it calls toString() method
System.out.println(e1); //e1.toString() [our class toString() executed output printed]
}
};

D:\morn11>java Employee
Employee@530daa

```

In above example Employee class is not overriding `toString()` method so parent class(**Object**) `toString()` method will be executed it returns hash code of the object.

**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);
        Employee e2 = new Employee("aruna",222,70000);
        Employee e3 = new Employee("nandu",222,80000);
//whenever we are printing reference variables internally it calls toString() method
System.out.println(e1);//e1.toString() [our class toString() executed output printed]
System.out.println(e2);//e2.toString() [our class toString() executed output printed]
System.out.println(e3);//e3.toString() [our class toString() executed output printed]
    }
};


```

In above example when you print reference variables it is executing `toString()` hence Employee values will be printed.

#### Final modifier:-

- 1) Final is the modifier applicable for classes, methods and variables (for all instance, Static and local variables).

#### Case 1:-

- 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.
- 2) Every method present inside a final class is always final but every variable present inside the final class not be final variable.

#### Example :-

```
final class Parent //parent is final class child class creation not possible
{
    };
class Child extends Parent //compilation error
{
    };
```

#### Example :-

Note :- 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:-

- 1) If a variable declared as a final we can not reassign that variable if we are trying to reassign compiler generate error message.
- 2) 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 :-**

```
class Parent
{
    void m1(){}
};

class Child extends Parent
{
    int m1(){}
};

D:\morn11>javac Test.java
m1() in Child cannot override m1() in Parent
return type int is not compatible with void
```

**Advantage of final modifier :-**

The main advantage of final modifier is we can achieve security as no one can be allowed to change our implementation.

**Disadvantage of final modifier:-**

But the main disadvantage of final keyword is we are missing key benefits of OOPS like inheritance and polymorphism. Hence is there is no specific requirement never recommended to use final modifier.

***Garbage Collector***

- Garbage collector is destroying the useless object and it is a part of the JVM.
- To make eligible objects to the 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();
        Test t2=new Test();
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1();
        System.gc();
    }
};
class Test
{
    //overriding finalize()
    public void finalize()
    {
        System.out.println("ratan sir object is destroyed");
    }
};

```

```

        System.out.println(10/0);
    }
    public static void main(String[] args)
    {
        Test t1 = new Test();
        Test t2 = new Test();
        ;;;;;//usage of objects
        t1=null;      //this object is eligible to Gc
        t2=null;      //this object is eligible to Gc
        System.gc();   //calling GarbageCollector
    }

}

//import java.lang.System;
//import static java.lang.System.*;
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();
        t1=null;
        t2=null;
        gc(); //static import
    }
};

Ex:- if the garbage collector is calling finalize() method at that situation exception is raised such type of exception are ignored.
class Test
{
    public void finalize()
    {
        System.out.println("ratan sir destroyed");
        int a=10/0;
    }
    public static void main(String[] args)
    {
        Test t1=new Test();
        Test t2=new Test();
        t1=t2;
        System.gc();
    }
};

```

**Abstraction:-**

There are two types of methods in java

- a. Normal methods
- b. Abstract methods

***Ex:- If user is calling finalize() method explicitly at that situation exception is raised.***

```

class Test
{
    public void finalize()
    {
        System.out.println("ratan sir destroyed");
        int a=10/0;
    }
    public static void main(String[] args)
    {
        Test t1=new Test();
        Test t2=new Test();
        t1=t2;
        t2.finalize();
    }
};

```

**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 :- inside the abstract class it is possible to declare**

```

abstract class Test
{
    public int a=10;
    public final int b=20;
    public static final int c=30;
    void disp1()
    {
        System.out.println("a value is="+a);
    }
};

class Test1 extends Test
{
    void disp2()
    {
        System.out.println("b value is="+b);
        System.out.println("c value is="+c);
    }
};

```

```

    }
    public static void main(String[] args)
    {
        Test1 t = new Test1();
        t.disp1();
        t.disp2();
    }
}

```

**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:-**

```

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.**

```

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 contains 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.

We are able to provide more encapsulation by taking the private data(variables) members.

To get and set the values from private members use getters and setters to set the data and to get the data.

**Example:-**

```
class Encapsulation
```

```

{      private int sid;
      private int sname;
      //mutator methods
      public void setSid(int x)
      {      this.sid=sid;      }
      public void setSname(String sname)
      {      this.sname=sname;      }
      //Accessor methods
      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** ---→ 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  
**string[] args**    **String []args**    **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... a)
static public void main(String... anushka)
```

- 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)
```

- 5) the applicable modifiers on main method.

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 strictfp synchronized void main(String... anushka)
```

#### 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... nandu) | --->valid   |

#### Example-1:-

```
class Test
{
    final strictfp synchronized static public void main(String...ratan)
    {
        System.out.println("hello ratan sir");
    }
};
```

#### Example-2:-

```
class Test1
{
    final strictfp synchronized static public void main(String...ratan)
    {
        System.out.println("Test-1");
    }
};

class Test2
{
    final strictfp synchronized static public void main(String...ratan)
    {
        System.out.println("Test-2");
    }
};

class Test3
{
    final strictfp synchronized static public void main(String...ratan)
    {System.out.println("Test-3");}
}
```

*In above two example execute all [.class] files to check the output.*

#### Example-3:-main method VS inheritance

```

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");
    }
};

```

**In above two examples execute both Parent and Child [.class] files to check the output.**

#### Example-4:-main method VS overloading

```

class Test
{
    public static void main(String[] args)
    {
        System.out.println("String[] parameter main method start");
        main(100); //1-argument ( int ) method calling
    }

    public static void main(int a)
    {
        main('r'); //1-argument ( char ) method calling
        System.out.println("int main method->" + a);
    }

    public static void main(char ch)
    {
        System.out.println("char main method->" + ch);
    }
}

```

#### 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 follows IEEE754  
**strictfp void m1(){}**                                  ---> m1() method follows IEEE754

#### Native modifier:-

- a. Native is the modifier applicable only for methods.
- b. Native method is used to represent particular method implementations there in non-java code (other languages like C,CPP) .
- c. Native methods are also known as “foreign methods”.

#### Examples:-

```

public final int getPriority();
public final void setName(java.lang.String);
public static native java.lang.Thread currentThread();
public static native void yield();

```

**Command Line Arguments:-**

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

**Example-1 :-**

```
class Test
{
    public static void main(String[] 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 commandline 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. introduced in 1.5 version.
2. it allows the methods to take any number of parameters.

**Syntax:-(only 3 dots)**

Void m1(int... a)

The above m1() is allows any number of parameters.(0 or 1 or 2 or 3.....)

**Example-1:-**

```
class Test
```

{

```

void m1(int... a){      System.out.println("Ratan");    }//var-arg method
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
}
}

Example-2:-
class Student
{
    void classRoom(int... fee) {System.out.println("class room --> B.tech --> CSE");}
    public static void main(String[] ratan)
    {
        Student s = new Student();
        s.classRoom();           //scholarship students
        s.classRoom(30000); //counselling fee students
        s.classRoom(100000,30000); //NRI student with donation + counselling fee
        s.classRoom(100000,30000,40000); //NRI student donation+mediator fee+counselling
    }
}

```

**Example-3:-printing var-arg values**

```

class Test
{
    void m1(int... a)
    {
        System.out.println("Ratan");
        for (int a1:a)
            {System.out.println(a1);}
    }
    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(10,20,30,40); //int var-arg method executed
    }
}

```

**Example-4:-var-arg VS normal arguments**

```

class Test
{
    void m1(int a,double d,char ch,String... str)
    {
        System.out.println(a+" "+d+" "+ch); //printing normal arguments
        for (String str1:str)//printing var-arg by using for-each loop
            {System.out.println(str1);}
    }
    public static void main(String... args)
    {
        Test t=new Test();
        t.m1(10,20.5,'s');
    }
}

```

```

        t.m1(10,20.5,'s',"aaaa");
        t.m1(10,20.5,'s',"aaaa","bbb");
    }
};

```

**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-5 :- var-arg method vs overloading**

```

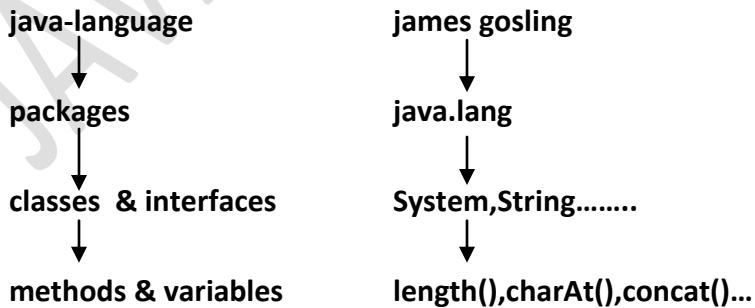
class Test
{
    void m1(int... a)
    {
        for (int a1:a)
        {
            System.out.println(a1);
        }
    }
    void m1(String... str)
    {
        for (String str1:str)
        {
            System.out.println(str1);
        }
    }
    public static void main(String[] args)
    {
        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 is **java.lang**. package.

<b>Java.lang</b>	<b>java.beans</b>	<b>java.text</b>	<b>java.sql</b>
<b>Java.io</b>	<b>java.net</b>	<b>java.nio</b>	<b>java.math</b>
<b>Java.util</b>	<b>java.applet</b>	<b>java.rmi</b>	
<b>Java.awt</b>	<b>java.times</b>	<b>java.security</b>	

**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-3:Invalid***

```
import java.io.*;
import java.lang.*;
package com.sravya;
```

#### ***Example-2:Invalid***

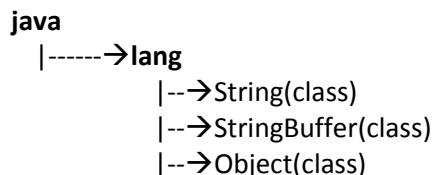
```
import java.io.*;
package com.sravya;
import java.io.*;
```

#### ***Example-4:Invalid***

```
package com.sravya;
package com.tcs;
```

### **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.



```

|-->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.

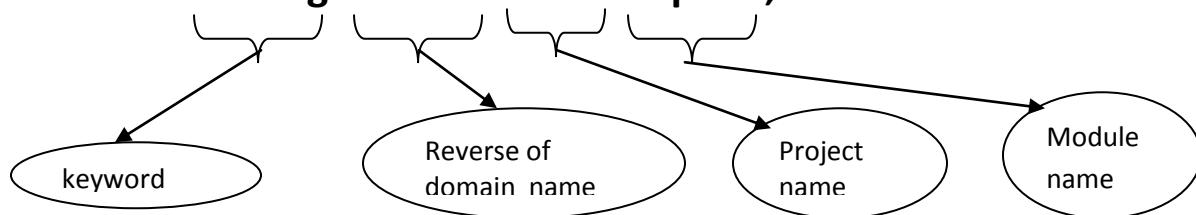
Domain name:-           **www.tcs.com**

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 moneytransfer**

```

com
|-->tcs
    |--bank
        |--moneytransfer
            |--->.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.**

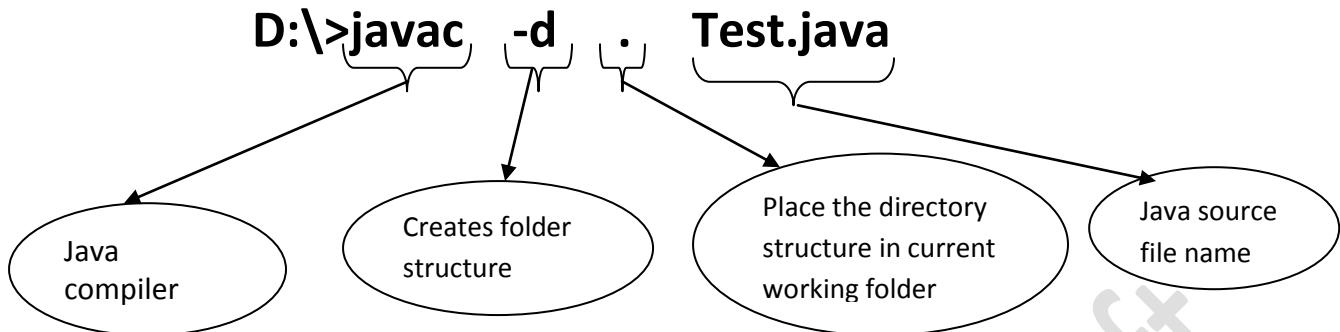
```

package com.sravya.java.corejava;
class Test
{
    public static void main(String[] args)
    {
        System.out.println("package first example");
    }
}
class A
{}
class B
{}
interface I
{}

```

**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
E:\>javac -d . Tcs.java
E:\>java com.tcs.states.requiredinfo.Tcs
jai telengana
jai andhra
jai jai others
```

**compilation of Sravya  
compilation of Tcs  
execution of Tcs**

**Example:-**

**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[] args)
    {
        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.**

**Example :-****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 in 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);
    }
}
```

**Parent-class method****Default****child-class method**

default	--> valid (same level)
protected , public	--> valid (increasing permission )
Private	--> invalid (decreasing permission)

**Public****public**

--> valid (same level)

Default,private,protected -->invalid(decreasing permission)

<b>Protected</b>	<b>protected</b>	---> valid (same level)
	<b>Public</b>	---> valid (increasing permission)
	<b>Default,private</b>	----> invalid (decreasing permission)

**Case 1:- same level [default-default]**

```
abstract class Test
{
    abstract void m1();      // default modifier
};

class Test1 extends Test
{
    void m1(){System.out.println("m1 method");} //default modifier
}
```

**Case 2:- increasing permission [protected-public]**

```
abstract class Test
{
    protected abstract void m1(); // protected modifier access
};

class Test1 extends Test
{
    public void m1(){System.out.println("m1 method");} //public modifier access
};
```

**Case3 :- decreasing permission [public-protected]**

```
abstract class Test
{
    public abstract void m1(); // public modifier
};

class Test1 extends Test
{
    protected void m1(){System.out.println("m1 method");} //protected modifier
};
```

**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 :-****Test.java:-**

```
package app1;
public class Test
{
    public void m1(){System.out.println("app1.Test class m1()");}
}
```

**A.java:-**

```
package app1.corejava;
public class A
{
    public void m1(){System.out.println("app1.corejava.A class m1()");}
}
```

**Ratan.java:-**

```
import app1.Test;
```

```

import app1.corejava.A;
class Ratan
{
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
        A a =new A();
        a.m1();
    }
}

```

**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();
    }
};

```

*Note :- applicable modifiers on constructors*

- 1) Public
- 2) Private
- 3) Default (if we are not providing modifiers)

## 4) Protected

**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
```

**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.

**Ex:-**

```

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)
    {
        A a = new A(); a.m1();
        B b = new B(); b.m1();
        C c = new C(); c.m1();
    }
}
```

**Example :- in java it is not possible to use predefined package names as a user defined packages.**

```
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

**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
    }
};
```

#### **Source file Declaration rules:-**

The source file contains the following elements

- 1) Package declaration---→optional----→at most one package(0 or 1)---→1<sup>st</sup> statement

- 2) Import declaration----→optional----→any number of imports-----→2<sup>nd</sup> statement
  - 3) Class declaration-----→optional----→any number of classes-----→3<sup>rd</sup> statement
  - 4) Interface declaration---→optional----→any number of interfaces-----→3<sup>rd</sup> statement
  - 5) Comments declaration→optional----→any number of comments----→3<sup>rd</sup> statement
- a. The package must be the first statement of the source file and it is possible to declare at most one package within the source file .
  - b. The import session must be in between the package and class statement. And it is possible to declare any number of import statements within the source file.
  - c. The class session is must be after package and import statement and it is possible to declare any number of class within the source file.
    - i. It is possible to declare at most one public class.
    - ii. It is possible to declare any number of non-public classes.
  - d. The package and import statements are applicable for all the classes present in the source file.
  - e. It is possible to declare comments at beginning and ending of any line of declaration it is possible to declare any number of comments within the source file.
- Preparation of user defined API (application programming interface document):-**
1. API document nothing but user guide.
  2. Whenever we are buying any product the manufacturing people provides one document called user guide. By using user guide we are able to use the product properly.
  3. James gosling is developed java product whenever james gosling is delivered the project that person is providing one user document called API(application programming interface) document it contains the information about how to use the product.
  4. To prepare user defined api document for the user defined projects we must provide the description by using documentation comments that information is visible in API document.

## 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 abstract class contains atleast one abstract method but the interface contains **only abstract methods**.
3. For the interfaces the compiler will generates .class files.
4. Interfaces giving the information about the functionalities and these are not giving the information about internal implementation.
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 { }**

and bydefault above three methods are public

the interface contains constants and these constants by default **public static final**

**Note-1 :- if u dont no the anything about implementation just we have the requirement specification them we should go for interface**

**Note-2:- If u know the implementation but not completly then we shold go for abstract class**

**Note-3 :-if you know the implementation completly then we should go for concrete class**

### 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 abstract methods provide the implementation in the implementation classes.
- Implementation class is nothing but the class that 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    // 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]
}

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();
    }
}

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 abstract 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, take child class of implementation class then complete the implementation of remaining abstract methods.
- In java it is possible to take any number of child classes but at final complete the implementation of all abstract methods.

```

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)
    {
        Test3 t = new Test3();
        t.m1();          t.m2();          t.m3();
    }
};

interface It1 //abstract
{
    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 3:-**

*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**.
3. The interface allows declaring only abstract methods.
4. Interface methods are by default **public abstract**.
5. The interface contains abstract methods with 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**.

### 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();**}

### 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.*

**Possibility of extends & implements keywords:-**

<i>class extends class</i>	
<i>class implements interface</i>	
<i>interface extends interface</i>	
<i>class A extends B</i>	=====>valid
<i>class A extends B,C</i>	=====>invalid
<i>class A implements it1</i>	=====>valid
<i>class A implements it1,it2,it3</i>	=====>valid
<i>interface it1 extends it2</i>	---->valid
<i>interface it1 extends it2,it3</i>	---->valid
<i>interface it1 extends A</i>	---->invalid
<i>interface it1 implements A</i>	---->invalid
<i>class A extends B implements it1,i2,it3</i>	=====>valid ( <i>extends must be first</i> )
<i>class A implements it1 extends B</i>	=====>invalid
<i>class extends class</i>	
<i>interface extends interface</i>	
<i>class implements interface</i>	
<i>class A extends B ----&gt;valid</i>	
<i>class A extends B,C ----&gt;Invalid</i>	
<i>class A implements It ----&gt;valid</i>	
<i>class A implements It1,It2 ----&gt;valid</i>	
<i>class A extends It ----&gt;Invalid</i>	
<i>class A extends A ----&gt;Invalid</i>	
<i>interface It1 extends It2 ---&gt;valid</i>	

```

interface It1 extends It2,It3 --->valid
interface It1 extends A --->invalid
interface It1 implements It2 --->invalid
interface It1 extends It1 ---->invalid

class A extends B implements It1,It2 --->valid [extends first]
class A implements It1,It2 extends B--->Invalid

```

#### 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

```

**Example :-****Demo.java**

```

package a;
public interface Demo
{
    public void sayHello(String msg);
}

```

**ImplClass:-**

```

package a;
class Test implements Demo
{
    public void sayHello(String msg)//overriding method of Demo interface
    {
        System.out.println("hi ratan--->"+msg);
    }
}
public class ImplClass
{
    public Test objectcreation()//it returns Test class Object
    {
        Test t = new Test();
        return t;
    }
}
Client.java
import a.ImplClass;
import a.Demo;
class Client
{
    public static void main(String[] args)
    {
        ImplClass i = new ImplClass();
        Demo d = i.objectcreation();
        //it returns Object of class Test but we don't know internally which object is created
        d.sayHello("hello");
    }
}

```

### ***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.

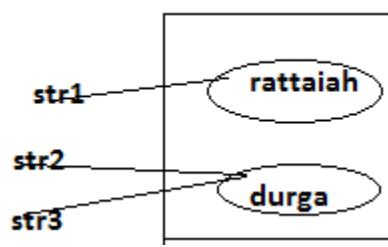
- 1) **Without using new operator**      **String str="ratan";**
- 2) **By using new operator**      **String str = new String("ratan");**

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 in SCP area then just before object creation it is always checking previous objects.

- If the previous object is available with the same content then it won't create new object that reference variable pointing to existing object.
  - If the previous objects are not available then JVM will create new object.
- SCP area does not allow duplicate objects.

#### 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");
```

#### 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);

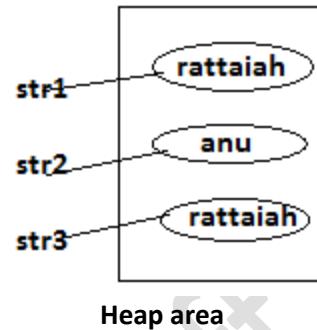
        //one approach to create StringBuffer Object (by using new operator)
        StringBuffer sb = new StringBuffer("ratansoft");
        System.out.println(sb);
    }
}
```

#### == 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
    }
}
```



- When we create object in Heap area instead of checking previous objects it directly creates objects.

- Heap memory allows duplicate objects.

```

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); //false
}

}

```

*Case 3:- String*

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

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

*StringBuffer is a mutability class it means once we are creating StringBuffer objects on that existing object it is possible to perform modification.*

#### **Example :-**

```

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

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

```

#### **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);
    }
}

```

- One java class method is able to return same class object or different class object that method is called factory method.
- In java if the method is returning some class object that method is called factory method.
- There are two types of factory methods in java
  - Instance factory method
  - Static factory method

#### **Instance factory method:-**

- Concat() is factory method because it is present in String class and able to return String class object only.

```
String str="ratan";
String str1 = str.concat("soft");
System.out.println(str1);
```

- toString() is factory method because StringBuffer class toString() method is returning String class object.

```
StringBuffer sb = new StringBuffer("anu");
String sss = sb.toString();
```

#### **Static factory method:-**

- if the factory method is called by using class name that method is called static factory method.

```
Integer i = Integer.valueOf(100);
System.out.println(i);
```

#### **Example :-**

```
class Test
{
    public static void main(String[] args)
    {
        //instance factory method [factory method is called ]
        String str="ratan";
        String str1 = str.concat("soft");
        System.out.println(str1);
        StringBuffer sb = new StringBuffer("anu");
        String sss = sb.toString();

        //static factory method
        Integer i = Integer.valueOf(100);
        System.out.println(i);
    }
}
```

#### **Java.lang.String vs java.lang.StringBuffer:-**

##### **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));
    }
}

```

**Java.lang.String vs java.lang.StringBuffer:-****Internal implementation of toString method:-**

- `toString()` method Returns a string representation of the object and it is present in `java.lang.Object` class.
- `String` is child class of `Object` and `String` is overriding `toString()` used to return content of the `String`.
- `StringBuffer` is child class of `Object` and `StringBuffer` is overriding `toString()` used to return content of the `StringBuffer`.

**Note :- whenever we are printing reference variable internally it is calling `toString()` method**

*In java when we print any type of reference variables internally it calls `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)
    {
        Test t = new Test();
        //the below two lines are same (if we are printing reference variables it's calling toString() method)
        System.out.println(t);           //object class toString() executed
        System.out.println(t.toString()); //object class toString() executed

        String str="ratan";
        System.out.println(str); //String class toString() executed
        System.out.println(str.toString()); //String class toString() executed

        StringBuffer sb = new StringBuffer("anu");
        System.out.println(sb);           //StringBuffer class toString() executed
        System.out.println(sb.toString()); //StringBuffer class toString() executed
    }
};
```

D:\>java Test

Test@530daa Test@530daa Ratan ratan Anu anu

In above example when we call `t.toString()` JVM searching `toString()` in `Test` class since not there then parent class(`Object`) `toString()` method is executed.

**`== 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
}

}

class Test extends Object
{
    Test(String str){}
    public static void main(String[] args)
    {
        Test t1 = new Test("ratan");
        Test t2 = new Test("anu");
        Test t3 = t2;
        Test t4 = new Test("ratan");
        System.out.println(t1==t2);//false
        System.out.println(t1==t3);//false
        System.out.println(t3==t2);//true
        System.out.println(t1==t4);//false
        //object class equals() executed reference comparison
        System.out.println(t1.equals(t2));//false
        System.out.println(t3.equals(t2));//true

        String str1 = "ratan";
        String str2="ratan";
        String str3=str2;
        System.out.println(str1==str2);//true
        System.out.println(str3==str2);//true
        System.out.println(str1==str3);//true
        //String class equals() executed content comparison
        System.out.println(str1.equals(str2));//true

        String s1= new String("ratan");
        String s2= new String("ratan");
        String s3=s2;
        System.out.println(s1==s2);//false
```

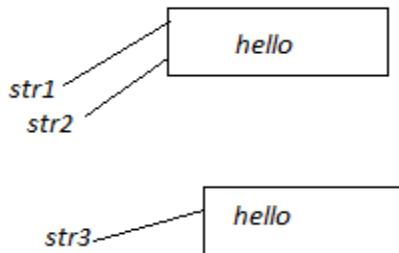
```

System.out.println(s2==s3);//true
//String class equals() executed content comparison
System.out.println(s1.equals(s2));//true

StringBuffer sb1 = new StringBuffer("anu");
StringBuffer sb2 = new StringBuffer("anu");
StringBuffer sb3 = sb1;
System.out.println(sb1==sb2);//false
System.out.println(sb1==sb3);//true
//StringBuffer class equals() executed reference comparison
System.out.println(sb1.equals(sb3));//true
}
}

```

#### 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");
        System.out.println(str1==str2);           //true
        System.out.println(str1==str3);           //false
        System.out.println(str1==str3);           //false
        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() & compareToIgnoreCase():-

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

- compareToIgnoreCase() 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".compareToIgnoreCase("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
```

**charAt(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)
    {
        //charAt() method
        String str="ratan";
        System.out.println(str.charAt(1));
        //System.out.println(str.charAt(10));StringIndexOutOfBoundsException
        char ch="ratan".charAt(2);
        System.out.println(ch);
        //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():-**

```

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

```

**Append():-**

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

Ex:-

```

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

```

**Insert():-**

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

```

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

```

**indexOf() and lastIndexOf():-**

Ex:-

```

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

```

```

        System.out.println(i);
        i=sb.lastIndexOf("hi");
        System.out.println(i);
    }
}

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

```

**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.

**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
    }
}

```

```
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());
        }

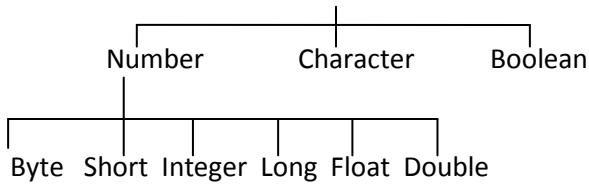
        //used our string to split given String
        String str1 = "hi,rata,mf,sdfsdf,ara"; StringTokenizer st1 = new
        StringTokenizer(str1,",");
        while (st1.hasMoreElements())
        {
            System.out.println(st1.nextElement());
        }
    }
}
```

## ***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:-**

Object

**Wrapper classes constructors:-**

```

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

**datatypes**

byte  
short  
int  
long  
float  
double  
char  
boolean

**wrapper-class constructors**

Byte	byte, String
Short	short, String
Integer	int, String
Long	long, String
Float	double, float, String
Double	double, String
Character	char
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**

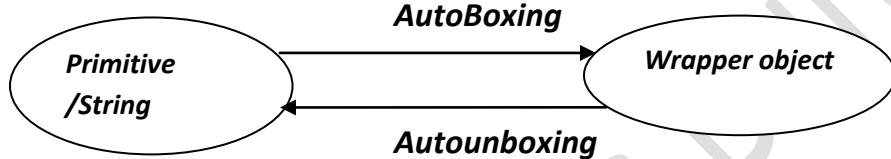
- Automatic conversion of primitive to wrapper object is called **autoboxing**.
- Automatic conversion of wrapper object to primitive is called **autounboxing**.

**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="savyainfotech";
        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

The data stored in computer memory in two ways.

#### 1) Temporary storage.

RAM is temporary storage whenever we're executing java program that memory is created when program completes memory destroyed. This type of memory is called volatile memory.

#### 2) Permanent storage.

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

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

#### 1) Determine whether the file is there or not.

- 2) Open a file.
- 3) Read the data from the file.
- 4) Writing information to file.
- 5) Closing file.**

#### **Stream :-**

Stream is a channel it support continuous flow of data from one place to another place  
 Java.io is a package which contains number of classes by using that classes we are able to send the data from one place to another place.

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

1. Byte format
2. Character format

#### **Stream/channel:-**

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

Streams are two types:-

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

#### **Byte oriented streams:-**

##### **Java.io.FileInputStream**

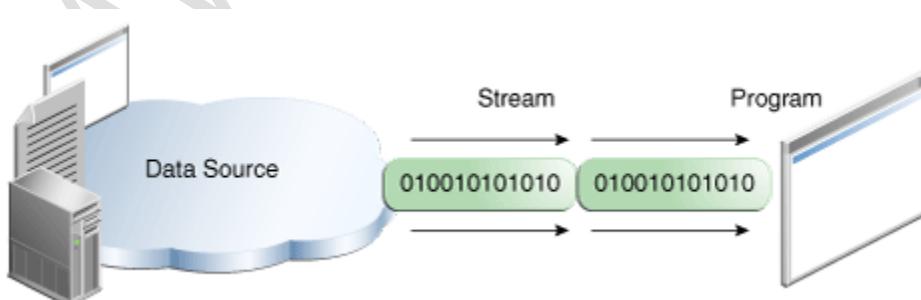
byte channel:-

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

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

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

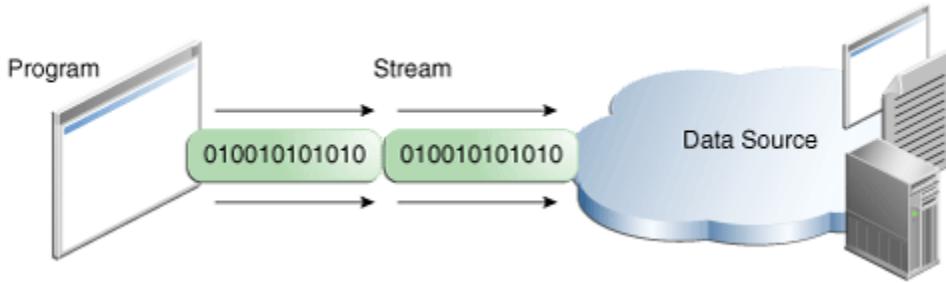
To read the data from the .txt file we have to read() method.



#### **Java.io.FileOutputStream:-**

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

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



Ex:- it will supports one character at a time.

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

#### Example :-

```
import java.io.*;
class Test
{
    public static void main(String[] args) throws Exception
    {
        FileReader fr = new FileReader("abc.txt");//read data from source file
        FileWriter fw = new FileWriter("xyz.txt");//write data to target file
        int c;
        while((c=fr.read())!=-1)           //read and checking operations
        {
            System.out.print((char)c);      //printing data of the file
            fw.write(c);                 //writing data to target file
        }
        System.out.println("read() & write operatoins are completed");
        //stream closing operations
        fr.close();
        fw.close();
    }
}
```

#### **Line oriented I/O:-**

Character oriented streams supports single character and line oriented streams supports single line data.

**BufferedReader**:- to read the data line by line format and we have to use `readLine()` to read the data.

**PrintWriter** :- to write the data line by line format and we have to use `println()` to write the data.

**Example :-**

```
import java.io.*;
class Test
{
    static BufferedReader br;
    static PrintWriter pw;
    public static void main(String[] args)
    {
        try{
            br=new BufferedReader(new FileReader("get.txt"));
            pw=new PrintWriter(new FileWriter("set.txt"));
            String line;
            while ((line=br.readLine())!=null)//reading & checking
            {
                System.out.println(line);      //printing data of file
                pw.println(line);           //writing data to target file
            }
            //close the streams
            br.close();
            pw.close();
        }
        catch(IOException io)
        {
            System.out.println("getting IOException");
        }
    }
}
```

#### **Buffered Streams:-**

Up to we are working with non buffered streams these are providing less performance because these are interact with the hard disk, network.

Now we have to work with Buffered Streams

`BufferedInputStream` read the data from memory area known as Buffer.

We are having four buffered Stream classes

1. `BufferedInputStream`
2. `BufferedOutputStream`
3. `BufferedReader`
4. `BufferedWriter`

Ex:-

```
import java.io.*;
class Test
{
    static BufferedReader br;
    static BufferedWriter bw;
    public static void main(String[] args)
    {
        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");
    }
}
}

```

**Ex:-**

```

import java.io.*;
class Test
{
    static BufferedInputStream bis;
    static BufferedOutputStream bos;
    public static void main(String[] args)
    {
        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");
        }
    }
}

```

**Ex:-**

```

import java.io.*;
class Test
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader br=new BufferedReader(new FileReader("abc.txt"));
        String str;
        while ((str=br.readLine())!=null)
        {
            System.out.println(str);
        }
    }
}

```

**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 from 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

### Information regarding Exception:-

- ❖ Dictionary meaning of the exception is abnormal termination.
- ❖ An expected event that disturbs or terminates normal flow of execution called exception.
- ❖ If the application contains exception then the program terminated abnormally the rest of the application is not executed.
- ❖ To overcome above limitation in order to execute the rest of the application must handle the exception.

In java we are having two approaches to handle the exceptions.

- 1) **By using try-catch block.**
- 2) **By using throws keyword.**

**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 and to get normal termination of the application.
- ✓ Every Exception is a predefined class present in different packages.

*java.lang.ArithmetricException  
java.io.IOException  
java.sql.SQLException  
javax.servlet.ServletException*

The exception are occurred due to two reasons

- a. Developer mistakes
- b. End-user mistakes.
  - i. While providing inputs to the application.
  - ii. Whenever user is entered invalid data then Exception is occur.
  - iii. A file that needs to be opened can't found then Exception is occurred.
  - iv. Exception is occurred when the network has disconnected at the middle of the communication.

**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 is called Checked Exceptions.  
*IOException,SQLException,InterruptedException.....etc*
- If the application contains checked exception the code is not compiled so must handle the checked Exception in two ways
  - By using try-catch block.
  - By using throws keyword.
- 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.

**There are two types of predefined methods**

- ✓ Exceptional methods
 

```
public static native void sleep(long) throws java.lang.InterruptedException
public boolean createNewFile() throws java.io.IOException
public abstract java.sql.Statement createStatement() throws java.sql.SQLException
```
- ✓ Normal methods
 

```
public long length();
public java.lang.String toString();
```

*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.*

***Checked Exception scenario:-*****Unchecked Exception:-**

- ❖ The exceptions which are not checked by the compiler at the time of compilation are called unchecked Exception.  
***ArithmetException,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 :- different types of unchecked exceptions.***

```
class Test
{
    public static void main(String[] args)
    {
        //java.lang.ArithmetException: / by zero
        System.out.println(10/0);

        //java.lang.ArrayIndexOutOfBoundsException
        int[] a={10,20,30};
        System.out.println(a[5]);

        //java.lang.StringIndexOutOfBoundsException
        System.out.println("ratan".charAt(10));
    }
}
```

**Note-1:-**

*If the application contains checked exception compiler generate information about exception so code is not compiled hence must handle that exception by using try-catch block or throws keyword it means for the checked exceptions try-catch blocks or throws keyword mandatory but if the application contains un-checked exception try-catch blocks or throws keyword is optional it means code is compiled but at runtime program is terminated abnormally.*

**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.*

**Note-3:-**

*In java whether it is checked Exception or unchecked exceptions are occurred at runtime but not compile time.*

**Error:-**

- Errors are caused due to lack of system resources like,
  - *Heap memory full.*
  - *Stack memory problem.*
  - *AWT component problems.....etc*

**Ex: - StackOverflowError, OutOfMemoryError, AssertionError.....etc**

- Exceptions are caused due to developers mistakes or end user supplied inputs but errors are caused due to lack of system resources.
- We are handle the exceptions by using try-catch blocks or throws keyword but we are unable to handle the errors.

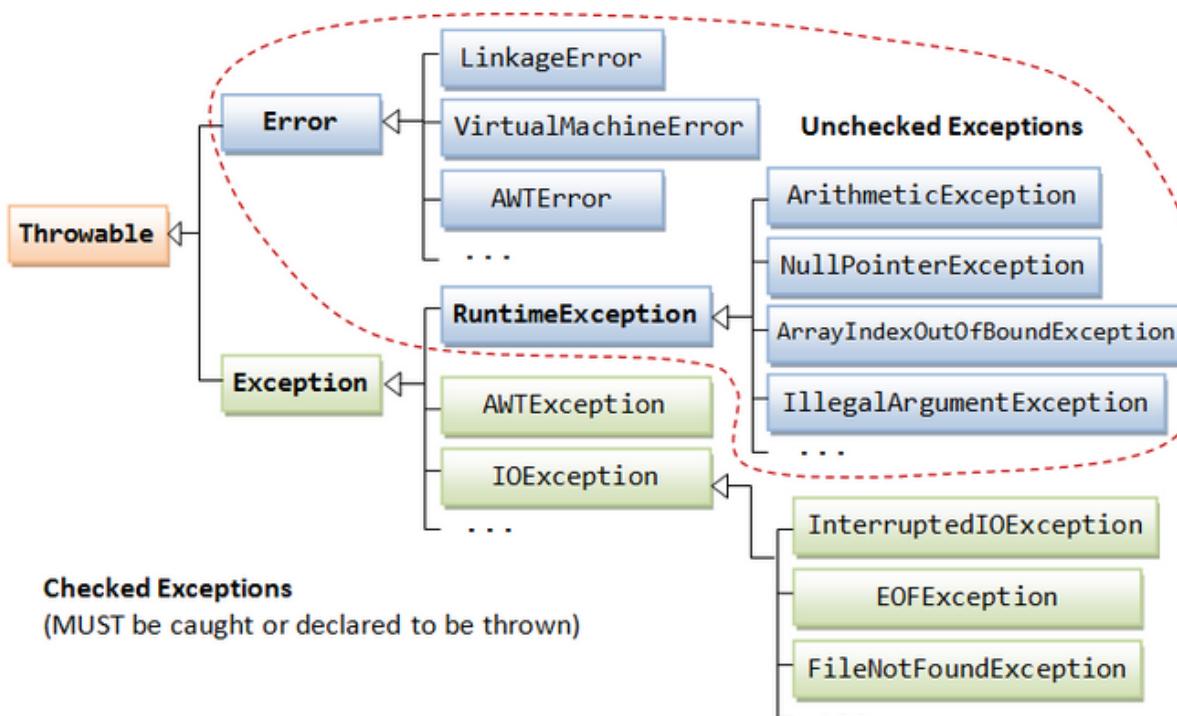
**Example:-**

```
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:-**

Root class of exception handling is Throwable class

**Checked Exceptions**

(MUST be caught or declared to be thrown)

- In above tree Structure RuntimeException its child classes and Error its child classes are Unchecked remaining all exceptions are checked Exceptions.

#### Exception handling key words:-

- 1) try
- 2) catch
- 3) finally
- 4) throw
- 5) throws

#### Exception Handling:-

In java whether it is a checked exception or unchecked Exception must handle the Exception by using try-catch blocks or throws to get normal termination of application.

#### Exception handling by using Try –catch block:-

##### Syntax:-

```
try
{
    exceptional code;
}
catch (ExceptionName reference_variable)
{
    Code to run if an exception is raised;
}
```

##### Example -1:-

##### Application without try-catch

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("rattan 1st class");
        System.out.println("rattan 2st class");
        System.out.println("rattan inter");
        System.out.println("rattan trainer");
        System.out.println("rattan weds anushka"+(10/0));
        System.out.println("rattan kids");
    }
}
```

D:\>java Test

rattan 1st class

rattan 2st class

rattan inter

rattan trainer

Exception in Thread "main" java.lang.ArithmaticException: / by zero

Handled by JVM

type of the Exception

description

##### Application with try-catch blocks:-

- 1) Whenever the exception is raised in the try block JVM won't terminate the program immediately it will search corresponding catch block.
  - a. If the catch block is matched that will be executed then rest of the application executed and program is terminated normally.
  - b. If the catch block is not matched program is terminated abnormally.

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("ratan 1st class");
        System.out.println("ratan 2st class");
        System.out.println("ratan inter");
        System.out.println("ratan trainer");
        try
        {
            //Exceptional code
            System.out.println("ratan weds anushka"+(10/0));
        }
        catch (ArithmaticException ae)
        {
            //alternate code
            System.out.println("ratan weds aruna");
        }
        System.out.println("ratan kids");
    }
}
D:\>java Test
ratan 1st class
ratan 2st class
ratan inter
ratan trainer
ratan weds aruna
ratan kids
```

**Example :-**

- If the exceptions raised in try block JVM will search for corresponding catch block,
  - If the catch block is matched corresponding catch is executed then rest of the application is executed & program terminated normally.
  - **If the catch block is not matched program is terminated abnormally the rest of the application is not executed.**

```
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.ArithmetricException: / by zero

```

**Example-3:-**If there is no exception in try block the 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:-**

in Exception handling independent try blocks 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:-**

In between try-catch blocks it is not possible to declare any statements must declare try with immediate catch block.

```

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

```

```

        {
            System.out.println("sravya");
            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:-**

- ❖ If the exception raised in try block jvm will search corresponding catch block.
- ❖ If the exception raised 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(ArithmeticException e)
        {
            System.out.println(10/0);
        }
        System.out.println("rest of the app");
    }
}

```

E:\sravya>java Test

sravya

**Exception in thread "main" java.lang.ArithmaticException: / by zero**  
at Test.main(Test.java:9)

**Example:-**

- ❖ If the exception raised in try block remaining code of try block won't be executed.
- 1) Once the control is out of the try block the control never entered into try block once again.

```

class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println(10/0);
            System.out.println("sravya");
            System.out.println("ratan");
        }
        catch(ArithmeticException 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);
            String str=null;
            System.out.println("u r name is :" +str);
            System.out.println("u r name length is--->" +str.length());
        }
        catch (ArithmaticException ae)
        {
            System.out.println("good boy zero not allowed getting Exception"+ae);
        }
        catch (NullPointerException ne)
        {
            System.out.println("good girl getting Exception"+ne);
        }
        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 Exceptional catch block we are able to hold any type of exceptions.

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Scanner s=new Scanner(System.in);
        System.out.println("provide the division value");
        int n=s.nextInt();
        try
        {
            System.out.println(10/n);
            String str=null;
            System.out.println("u r name is :" +str);
            System.out.println("u r name length is--->" +str.length());
        }
        catch (Exception e)//this catch block is able to handle all types of Exceptions
        {
System.out.println("I am an inexperienced or lazy programmer here=" +e) ;
        }
    }
}

```

```

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

```

**Example -10:-if we are declaring multiple catch blocks at that situation the catch block order should be child to parent shouldn't be parent to the child.**

**(No compilation error)**

**Child-parent**

```

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 (ArithmaticException ae)
        {
            System.out.println("Exception"+ae);
        }
        catch (Exception ne)
        {
            System.out.println("Exception"+ne);
        }
        System.out.println("rest of the code");
    }
}

```

**Possibilities of try-catch blocks:-**

**Possibility-1**

```

try { }
catch () { }

```

**Possibility-2**

```

try
{
}
catch ()
{
}
try
{
}
catch ()
{
}

```

**Possibility-3**

```

try
{
}

```

**Compilation error**

```

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 (Exception ae)
        {
            System.out.println("Exception"+ae);
        }
        catch (ArithmaticException ne)
        {
            System.out.println("Exception"+ne);
        }
        System.out.println("rest of the code");
    }
}
catch () { }
catch () { }

```

**Possibility-4**

```

try
{
    try
    {
    }
    catch ()
    {
    }
}
catch () { }

```

**Possibility-5**

```

try
{
}

```

```

catch ()
{
    try
    {
        }
        catch ()
        {
            }
    }
Possibility-6
try
{
    try
    {
        }
}

```

```

        catch ()
        {
            }
        }
        catch ()
        {
            try
            {
                }
                catch ()
                {
                    }
            }
}

```

**Example 11:-**

It is possible to combine two exceptions in single catch block the syntax is `catch(ArithmeticException | StringIndexOutOfBoundsException a)` .

```

import java.util.Scanner;
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(ArithmeticException | StringIndexOutOfBoundsException a)
    {
        System.out.println("ratansoft");
    }
    System.out.println("Rest of the application");
}
}
D:\DP>java Test
enter a number
0
ratansoft
Rest of the application

```

```

D:\DP>java Test
enter a number
2
5
ratansoft
Rest of the application

```

**Finally block:-**

- 1) Finally block is always executed irrespective of try and catch.
- 2) It is used to provide clean-up code
  - a. Database 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`  $\rightarrow$  valid
  - b. `try-catch`  $\rightarrow$  valid

- |                            |                |
|----------------------------|----------------|
| c. catch-finally           | -- → invalided |
| d. try-catch-catch-finally | -- → valided   |
| e. try-finally             | -- → valided   |
| f. catch-catch-finally     | -- → invalided |
| g. Try                     | -- → invalided |
| h. Catch                   | -- → invalided |
| i. Finally                 | - → invalided  |

**Syntax:-**

```
try
{   risky code;
}
catch (Exception obj)
{   handling code;
}
finally
{   Clean-up code;(database connection closing,streams closing.....etc)
}
```

**Example:-**

if the exception raised in try block the JVM will search for corresponding catch block ,

- If the corresponding catch block is matched the catch block is executed then finally block is executed.
- If the corresponding catch block is not matched the program is terminated abnormally just before abnormal termination the finally block will be executed then program is terminated abnormally.

### **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 3:-**

```
try
{   System.out.println(10/0);
}
catch (NullPointerException ae)
{   System.out.println("catch");
}
finally
```

**case 2:-**

```
try
{   System.out.println(10/0);
}
catch (ArithmaticException ae)
{   System.out.println("catch");
}
finally
{   System.out.println("finally");
}
```

**Output:-**

```
catch
finally
```

**Output:**

```
finally
Exception in thread "main"
java.lang.ArithmaticException: / by zero
at Test.main(Test.java:4)
```

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)**

```

{
    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**  
**at Test.main(Test.java:15)**

case 5:-

```

try
{
    System.out.println("try");
}
catch(ArithmaticException ae)
{
    System.out.println("catch");
}
finally

```

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:-in only two cases finally block won't be executed

Case 1:- whenever we are giving chance to 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.ArithmetricException: / by zero
at Test.main(Test.java:5)

```

**Case 2:-**In your program whenever we are using `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

#### Methods to print Exception information:-

```

class Test1
{
    void m1()
    {
        m2();
    }
    void m2()
    {
        m3();
    }
    void m3()
    {
        try{
            System.out.println(10/0);
        catch(ArithmetricException 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 Test1

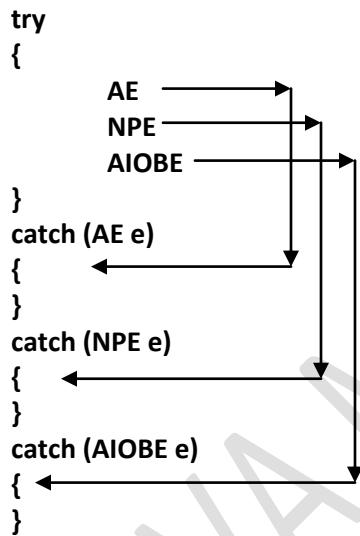
```
java.lang.ArithmaticException: / by zero
/ by zero
java.lang.ArithmaticException: / by zero
at Test1.m3(Test1.java:8)
at Test1.m2(Test1.java:5)
at Test1.m1(Test1.java:3)
at Test1.main(Test1.java:17)
```

//*toString()* method output  
 //*getMessage()* method output  
 //*printStackTrace()* method

### Possibilities of exception handling:-

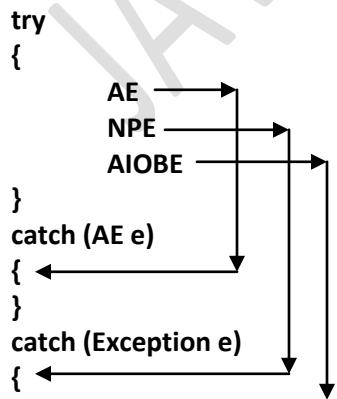
#### Example-1:-

```
try
{
  AE
  NPE
  AIOBE
}
catch (AE e)
{
}
catch (NPE e)
{
}
catch (AIOBE e)
{
}
```



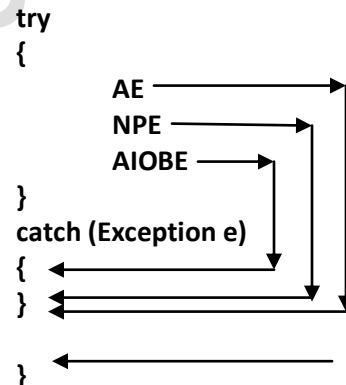
#### Example-3:-

```
try
{
  AE
  NPE
  AIOBE
}
catch (AE e)
{
}
catch (Exception e)
{
}
```



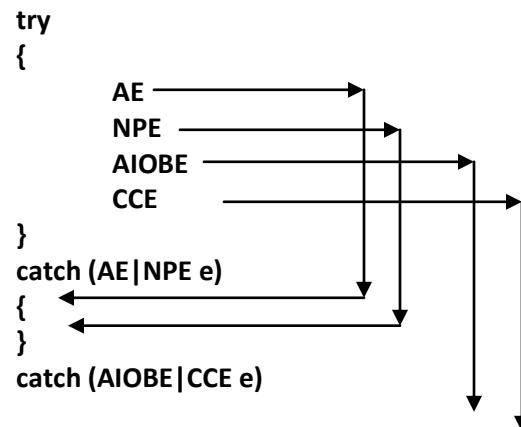
#### Example-2:-

```
try
{
  AE
  NPE
  AIOBE
}
catch (Exception e)
{
}
}
```



#### Ex 4:-introduced in 1.7 version

```
try
{
  AE
  NPE
  AIOBE
  CCE
}
catch (AE|NPE e)
{
}
catch (AIOBE|CCE e)
{
}
```



{ ← } ←

**Example :-**

```

statement 1
statement 2
try
{
    statement 3
    try
    {
        statement 4
        statement 5
    }
    catch ()
    {
        statement 6
        statement 7
    }
}
catch ()
{
    statement 8
    statement 9
    try
    {
        statement 10
        statement 11
    }
    catch ()
    {
        statement 12
        statement 13
    }
}
Finally{
statement 14
statement 15
}
Statement -16
Statement -17

```

**case 1:- if there is 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 , Abnrmal Termination

**Case 3:- if the exception is raised in the statement 3 the corresponding catch block is matched.**

1,2,8,9,10,11,14,15 normal termination

**Case 4:- if the exception is raise in the statement-4 the corresponding catch block is not matched and outer catch block is not matched.**

1,2,3 abnormal termination.

**Case 5:- If the exception is raised in the statement 5 and corresponding catch block is not matched and outer 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 5 and the corresponding catch block is not matched and outer catch block is matched while executing outer catch inside the try**

**block the exception is raised in the statement 10 and the corresponding catch is matched.**

1,2,3,4,8,9,12,13,14,15 normal termination.

**Case 7:- If the exception raised in statement 14.**

1,2,3,4,5 abnormal termination.

**Case 8:- if the Exception raised in statement 17.**

### Throws :-

- 1) In the exception handling must handle the exception in two ways
  - a. By using try-catch blocks.
  - b. By using throws keyword.
- 2) Try-catch block is used to handle the exception but throws keyword is used to **delegate** the responsibilities of the exception handling to the caller method.
- 3) The main purpose of the throws keyword is **bypassing** the generated exception from present method to caller method.
- 4) Use throws keyword at method declaration level.
- 5) It is possible to throws any number of exceptions at a time based on the programmer requirement.
- 6) If main method is throws the exception then JVm is responsible to handle the exception.

### By using try-catch blocks:-

```
import java.io.*;
class Student
{
    void studentDetails()
    {
        try
        {
            BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
            System.out.println(" enter student name");
            String sname=br.readLine();
            System.out.println("u r name is:"+sname);
        }
        catch(IOException e)
        {
            System.out.println(" getting Exception"+e);
        }
    }
    public static void main(String[] args)
    {
        Student s1=new Student();
        s1.studentDetails();
    }
}
```

### Ex ample:-

```
import java.io.*;
class Student
{
    void studentDetails()throws IOException ////(delegating responsibilities to caller method principal())
    {
    }
```

### Handling the exception by using throws keyword:-

```
Ex 1:-
import java.io.*;
class Student
{
    void studentDetails()throws IOException
    {
        BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
        System.out.println(" enter student name");
        String sname=br.readLine();
        System.out.println("u r name is:"+sname);
    }
    public static void main(String[] args)throws
IOException
    {
        Student s1=new Student();
        s1.studentDetails();
    }
}
```

```

{
    BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
    System.out.println("please enter student name");
    String sname=br.readLine();
    System.out.println("please enter student rollno");
    int sroll=Integer.parseInt(br.readLine());
    System.out.println("enter student address");
    String saddr=br.readLine();
    System.out.println("student name is:"+sname);
    System.out.println("student rollno is:"+sroll);
    System.out.println("student address is:"+saddr);
}
void principal() throws IOException ////(delegating responsibilities to caller method officeBoy())
{
    studentDetails();
}
void officeBoy()throws IOException ////(delegating responsibilities to caller method main())
{
    principal();
}
public static void main(String[] args) throws IOException ////(delegating responsibilities to JVM)
{
    Student s1=new Student();
    s1.officeBoy();
}
}

```

**Throw:-**

- 1) The main purpose of the throw keyword is to creation of Exception object explicitly either for predefined or user defined exception.
- 2) Throw keyword works like a try block. The difference is try block is automatically find the situation and creates an Exception object implicitly. Whereas throw keyword creates an Exception object explicitly.
- 3) Throws keyword is used to delegate the responsibilities to the caller method but throw is used to create the exception object.
- 4) If exception object created by JVM it will print predefined information (**/ by zero**) but if exception Object created by user then user defined information is printed.
- 5) We are using throws keyword at method declaration level but throw keyword used at method implementation (body) level.

throw keyword having two objectives

1. Handover the user created exception object JVM for predefined Exception.
2. Handover the user created exception object JVM for user defined Exception.

**Ex:- Objective-1 of the throw keyword**

**throw keyword is used to create the exception object explicitly by the developer for predefined exceptions.**

Step -1 :- create the Exception object explicitly by the developer.

```
new ArithmeticException("ratan not eligible");
```

Step -2:- handover user created Exception object to jvm by using throw keyword.

```
throw new ArithmeticException("ratan not eligible");
```

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

```

class Test
{
    static void validate(int age)
    {
        if (age<18)
        {
            //creating Exception object by user & handover to Jvm
            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 ");
        int n=s.nextInt();
        validate(n);
        System.out.println("rest of the code");
    }
}

```

**Objective-2 :- throw keyword is used to create the exception object explicitly by the developer for the user defined exceptions.**

There are two types of exceptions present in the java language

- 1) Predefined Exceptions.
- 2) User defined Exceptions.

#### **Predefined Exception:-**

These exceptions are introduced by James Gosling comes along with software.  
Ex:- ArithmeticException, IOException, NullPointerException.....etc

#### **User defined Exceptions:-**

Exceptions created by user are called userdefined Exceptions.  
Ex: InvalidAgeException, BombBlotException.....etc

#### **Step-1: create user defined Exception.**

```

class InvalidAgeException extends Exception
{
};

```

#### **Step-2:- create the object of user defined Exception.**

```
new InvalidAgeException();
```

#### **step-3:- handover user defined Exception object to Jvm by using throw keyword.**

```
throw new InvalidAgeException();
```

#### **creation of user defined Exceptions:-(customization of Exceptions)**

there are two types of user defined exceptions

- i) User defined checked Exception (these Exceptions are extends Exception class)
- ii) User defined un-checked Exception (extends RuntimeException class)

The naming conventions are every exception suffix must be the word Exception.

#### ***Creation of Userdefined checked Exception:-***

*There are two approaches to create Userdefined checked Exception*

1. Default constructor approach
2. Parameterized constructor approach

#### **Creation of userdefined checked Exception by using default constructor approach:-**

##### **Step-1:- create the user defined 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
```

;Note: - in this example we are creating user defind checked Exception hence must handle the Exception by using try-catch or throws keyword otherwise compiler generate compilation error "unreportedException"

##### **Step-2:- use created Exception in our project.**

##### **Project.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(); //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>javac -d . InvalidAgeExcepiton.java

D:\morn11>javac -d . Test.java

D:\morn11>java com.tcs.project.Test

enter u r age

19

*Exception in thread "main" com.tcs.userexceptions.InvalidAgeExcepiton*

*at com.tcs.project.Test.status(Test.java:11)*

*at com.tcs.project.Test.main(Test.java:18)*

#### **creation of userdefined checked exception by using parameterized constructor approach:-**

##### **step-1:- create the userdefined exception class.**

##### **InvalidAgeException.java**

```
package com.tcs.userexceptions;
```

```
public class InvalidAgeExcepiton extends Exception
{
    public InvalidAgeExcepiton(String str)
    { //super constructor calling inorder to print your information
        super(str);
    }
};
```

**Step-2:- use user created Exception in our project.**

**Project.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)
```

**Ex:- creation of user defined un-checked exception by using default constructor approach:-**

**Step-1:- create userdefined exception.**

**InvalidAgeException.java**

```
//InvalidAgeException.java
```

```
package com.tcs.userexceptions;
public class InvalidAgeExcepiton extends RuntimeException
{
    //default constructor
};
```

Note: - in this example we are creating user defined unchecked exception so try-catch blocks and throws keywords are optional.

### **Step-2:- use user created Exception in our project.**

#### **Project.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);
    }
}
```

#### **Ex:- creation of user defined un-checked exception by using parameterized constructor**

#### **approach:-**

#### **InvalidAgeException.java**

```
//InvalidAgeException.java
package com.tcs.userexceptions;
public class InvalidAgeExcepiton extends RuntimeException
{
    public InvalidAgeExcepiton(String str)
    {super(str);
    }
};
```

#### **Project.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 InvalidAgeException("not eligible for mrg");
    }
}

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);
}
}

```

**Different types of exceptions:-****ArrayIndexOutOfBoundsException:-**

```

int[] a={10,20,30};
System.out.println(a[0]);//10
System.out.println(a[3]);//ArrayIndexOutOfBoundsException

```

**NumberFormatException:-**

```

String str="123";
int a=Integer.parseInt(str);
System.out.println(a);//conversion(string - int) is good
String str1="abc";
int b=Integer.parseInt(str1);
System.out.println(b);//NumberFormatException

```

**NullPointerException:-**

```

String str="rattaiah";
System.out.println(str.length());//8
String str1=null;
System.out.println(str1.length());//NullPointerException

```

```

Test t = new Test();
t.m1(); //output printed
t=null;
t.m1(); //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(3));//t
System.out.println(str.charAt(13));//StringIndexOutOfBoundsException
```

**NegativeArraySizeException:-**

```
int[] a1=new int[100];
System.out.println(a1.length);//100
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

**Different types of Errors:-****StackOverflowError:-**

```
class Test
{
    void m1()
    {
        m2();
        System.out.println("this is Rattaiah");
    }
    void m2()
    {
        m1();
        System.out.println("from Sravyasoft");
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1();
    }
}
```

**OutOfMemoryError:-**

```
class Test
{
    public static void main(String[] args)
    {
        int[] a=new int[1000000000]; //OutOfMemoryError
    }
}
```

**Different types of Exceptions in java:-**

Checked Exception	Description
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>	<b>Description</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 of co-operative execution

**To overcome above problem a new programming style will be introduced is called multiprogramming.**

- 1) Multiprogramming means executing 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 an 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 execute simultaneously that each and every part is called thread.
- 4) The thread is a 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 at a time is called multithreading.

#### **Information about main Thread:-**

When a Java program starts 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 performs 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);
    }
}

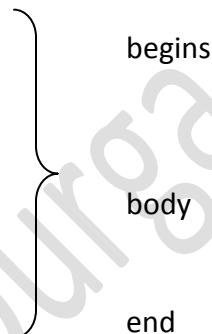
```

### 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 Srawyasoft");
    }
}

```



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 following 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 a 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("Srawyasoft 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");
    }
}

```

```

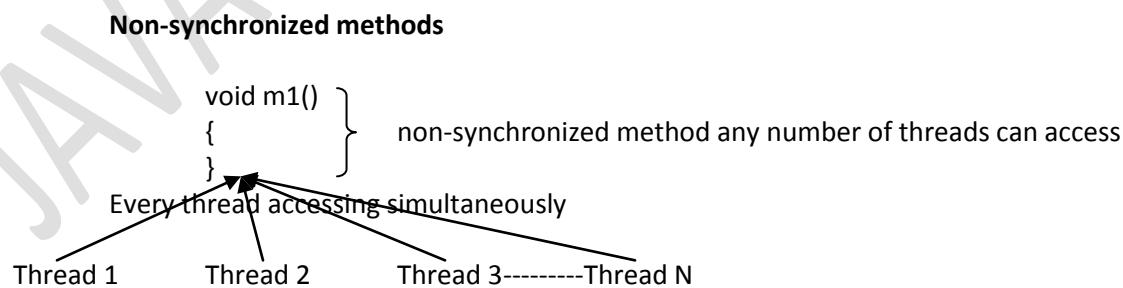
class ThreadDemo
{
    public static void main(String[] args) throws InterruptedException
    {
        MyThread t = new MyThread();
        //creating Runtime class Object by using factory method
        Runtime r = Runtime.getRuntime();
        r.addShutdownHook(t); //adding Thread to JVM hook
        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.

```

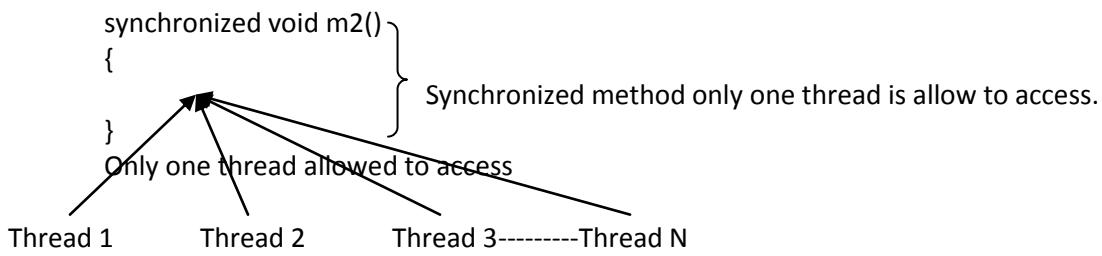
### Synchronized :-

- 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 inconsistency problems.



- 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 allow 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 synchronized only 10 lines of code use synchronized blocks.*

*The synchronized block contains less scope compare to method.*

*If we are writing all the method code inside the synchronized blocks it will work same as the synchronized method.*

**Syntax:-**

```
synchronized(object)
{
    //code
}

class Heroin
{
    public void message(String msg)
    {
        synchronized(this){
            System.out.println("hi "+msg+" "+Thread.currentThread().getName());
            try{Thread.sleep(5000);}
            catch(InterruptedException e){e.printStackTrace();}
        }
        System.out.println("hi 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.

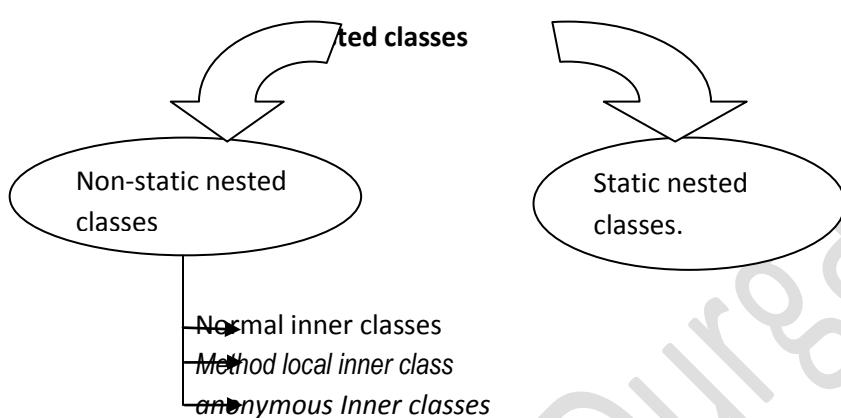
### Nested classes

- Declaring the class inside another class is called nested classes. This concept is introduced in the 1.1 version.
- Declaring the methods inside another method is called inner methods java not supporting inner methods concept.

The nested classes are divided into two categories

1. **Static nested classes**(nested class declared with static modifier)
2. **Non static nested classes**( these are called inner classes)
  - a. **Normal inner classes**
  - b. **Method local inner classes**
  - c. **Anonymous inner classes**

**Static nested classes**:- The nested classes declare as a static modifier is called static nested classes.



#### **syntax of nested classes :-**

```

class Outerclasses
{
    //static nested class
    static class staticnestedclass
    {
        ;
    }
    //non-static nested class
    class Innerclass
    {
        ;
    }
}
  
```

#### **Uses of nested classes:-**

1. **It is the way logically grouping classes that are only used in the one place.**

If a class is useful to other class only one time then it is logically embedded it into that classes make the two classes together.

#### **A is only one time usage in the B class**

**(without using inner classes)**

```

class A
{
    ;
}
class B
{
    A a=new A();
}
  
```

#### **by using inner classes**

```

class B
{
    class A
    {
        ;
    };
}
  
```

2. **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.

3. **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.

For the outer classes the compiler will provide the .class and for the inner classes also the compiler will provide the .class file.

The .class file name for the inner classes is **OuterclassName\$innerclassname.class**

<b>Outer class object creation</b>	<b>::-</b>	<b>Outer o=new Outer();</b>
<b>Inner class object creation</b>	<b>::-</b>	<b>Outer.Inner i=o.new Inner();</b>
<b>Outer class name</b>	<b>::-</b>	<b>Outer.class</b>
<b>Inner class name</b>	<b>::-</b>	<b>Outer\$Inner.class</b>

#### **Member inner classes:-**

1. If we are declaring any data in outer class then it is automatically available to inner classes.
2. If we are declaring any data in inner class then that data is should not have the scope of the outer class.

#### **Syntax:-**

```
class Outer
{
    class Inner
    {
    };
}
```

#### **Object creation syntax:-**

##### **Syntax 1:-**

```
OuterClassName o=new OuterClassName();
OuterClassName.InnerClassName oi=OuterObjectreference.new InnterClassName();
```

##### **Syntax 2:-**

```
OuterClassName.InnerClassName oi=new OuterClass().new InnerClass();
```

**Note:- by using outer class name it is possible to call only outer class properties and methods and by using inner class object we are able to call only inner classes properties and methods.**

**Example :-**

```
class Outer
{
    private int a=100;
    class Inner
    {
        void data()
        {
            System.out.println("the value is :" + a);
        }
    }
}
class Test
{
    public static void main(String[] args)
    {
        Outer o=new Outer();
        Outer.Inner i=o.new Inner();
        i.data();
    }
};
```

#### **Example :-**

```
class Outer
{
    int i=100;
    void m1()
    {
        //j=j+10;// compilation error
        //System.out.println(j);//compilation error
    }
}
```

```

        System.out.println("m1 method");
    }
    class Inner
    {
        int j=200;
        void m2()
        {
            i=i+10;
            System.out.println(i);
        }
    };
}
class Test
{
    public static void main(String[] args)
    {
        A a=new A();
        System.out.println(a.i);
        a.m1();
        A.B b=a.new B();
        System.out.println(b.j);
        b.m2();
        //b.m1(); compilation error
    }
};

Example :-
class Outer
{
    private int a=10;      private int b=20;
    void m1()
    {
        //m2(); not possible
        System.out.println("outer class m1()");
    }
    class Inner
    {
        int i=100; int j=200;
        void m2()
        {
            System.out.println("inner class m1()");
            System.out.println(a+b);
            System.out.println(i+j);
            m1();
        }
    };
}
class Test
{
    public static void main(String... ratan)
    {
        Outer o = new Outer();
        Outer.Inner i = o.new Inner();           o.m1();
                                                i.m2();
    }
};

```

**Application required this & super keywords:-**

```

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)
    {
        Outer.Inner i = new Outer().new Inner();
        i.m1(1000,2000);
    }
};

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();
    }
};
```

#### Method local inner classes:-

1. Declaring the class inside the method is called method local inner classes.
2. In the case of the method local inner classes the class has the scope up to the respective method.
3. Method local inner classes do not have the scope of the outside of the respective method.
4. whenever the method is completed
5. we are able to perform any operations of method local inner class only inside the respective method.

**Syntax:-**

```
class Outer
{
    void m1()
    {
        class inner
        {
        };
    }
};
```

**Example:-**

```
class Outer
{
    private int a=100;
    void m1()
    {
        class Inner
        {
            void innerMethod()
            {
                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();
    }
};

class Outer
{
    void m1()
    {
        class Inner
        {
            void m1(){System.out.println("inner class m1()");}
        };
        Inner i = new Inner();
        i.m1();
    }
};

public static void main(String[] args)
{
    Outer o = new Outer();
    o.m1();
}

class Outer
{
    private int a=100;
    void m1()
    {
        final int b=200;//local variables must be final variables
        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)
{
    Outer o = new Outer();
    o.m1();
}
};

class Outer
{
    int a=10;//instance variable
    static int b=20; //static variable
    class Inner //inner class able to access both instance and static variables
    {
        void m1()
        {
            System.out.println(a);
            System.out.println(b);
        }
    };
}

class Outer
{
    static int a=10;//static variable
    int b=20;           //instance variable
    static class Inner //this inner class able to access only static members of outer class
    {
        void m1(){
            System.out.println(a);
            System.out.println(b); //compilation error
        }
    };
}

```

Ex 2:-in method local inner classes it is not possible to call the non-final variables inside the inner classes hence we must declare that local variables must be final then only it is possible to access that members.

```

class Outer
{
    private int a=100;
    void m1()
    {final int b=1000;
    class Inner
    {
        void innerMethod()
        {
            System.out.println("inner class method");
            System.out.println(a);
            System.out.println(b);
        }
    };
    Inner i=new Inner();
    i.innerMethod();
}

```

```

        }
    };
class Test
{
    public static void main(String[] args)
    {
        Outer o=new Outer();
        o.m1();
    }
};

```

**Static inner classes:-**

In general in java classes it is not possible to declare any class as a abstract class but is possible to declare inner class as a static modifier.

Declaring the static class inside the another class is called static inner class.

Static inner classes can access only static variables and static methods it does not access the instance variables and instance methods.

**Syntax:-**

```

class Outer
{
    static class Inner
    {
    };
}

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();
    }
};

class Outer
{
    static int a=10;//static variable
    static int b=20;//static variable
    static class Inner //this inner class able to access only static members of outer class
    {
        void m1(){
            System.out.println(a);
            System.out.println(b);
        }
    };
    public static void main(String[] args)
    {
        Outer.Inner i = new Outer.Inner();//it creates object of static inner class
        i.m1();
    }
};

```

```

    }
};

class Outer
{
    static int a=10;//static variable
    static int b=20;//static variable
    static class Inner //this inner class able to access only static members of outer class
    {
        void m1(){
            System.out.println(a);
            System.out.println(b);
        }
    };
    public static void main(String[] args)
    {
        Outer.Inner i = new Outer.Inner(); //it creates object of static inner class
        i.m1();
    }
}

```

**Anonymous inner class:-**

1. The name less inner class is called anonymous inner class.
2. it can be used to provide the implementation of normal class or abstract class or interface

**Anonymous inner classes for abstract classes:-**

**it is possible to provide abstract method implementations by taking inner classes.**

**Ex:- we are able to declare anonymousinner class inside the class.**

```

abstract class Animal
{
    abstract void eat();
};

class Test
{
    //anonymous inner class
    Animal a=new Animal()
    {
        void eat() { System.out.println("animals eating gross"); }
    };

    public static void main(String[] args)
    {
        Test t=new Test();
        t.a.eat();
    }
}

```

**Ex:- we are able to declare anonymous inner class inside the main method.**

```

abstract class Animal
{
    abstract void eat();
};

class Test
{
    public static void main(String[] args)
    {
        Animal a=new Animal()
        {
            void eat()
            {
                System.out.println("animals eating gross");
            }
        };
        a.eat();
    }
}

```

```
}

Note :- In above example we are taking animal class having eat() method and we are overriding method but this thing can done by creating subclasses of existing class by using extends keyword then what is the need of anonymous inner classes.

The answer is creating anonymous inner class simple. And whenever we are inherit few properties (only method) of superclass instead of extending class use anonymous inner class.

//interface (contains abstract methods)
interface It1
{
    void m1();
    void m2();
    ::::::::::::::::::::
    void m100();
}

//adaptor class(contains empty implementation of interface methods)
class X implements It1
{
    void m1(){}
    void m2(){}
    ::::::::::::
    void m100(){}
};

//userdefined class extending adaptor class
class Test extends X
{
    //all methods are visible here
};

//useing anonymous inner class (override required method)
class Test
{
    //anonymous inner class
    X x = new X()
    {
        //override required methods(required methods are loaded)
        void m1(){System.out.println("anonymous inner class");}
        //semicolon mandatory
    };
    interface It1 //interface
    {
        void m1(); //by default interface methods are puliv abstract
        void m2();
        ::::::::::::::::::::
        void m100();
    }
    class X implements It1//adaptor class
    { //it is adaptor class contains empty implementation of all interface methods
        public void m1(){}
        //implementation method must be public
        public void m2(){}
    }
}
```

```
:::::::::::  
public void m100(){}
};  
abstract class Test implements It1
{
    //must provide the implementation of 100 methods
};  
//approach-1 it is possible to extends the class and override required method  
class Test1 extends X
{
    //override the required methods
    public void m1(){System.out.println("m1 method");}
};  
//approach-2 without extending class it is possible to create the object directly and override required  
method  
class Ratan
{
    X x= new X()//this is anonymous inner class
    {
        public void m1(){System.out.println("anonymous inner class");} }; //semicolon  
mandatory
    public static void main(String[] args)
    {
        Ratan r = new Ratan();
        r.x.m1();
    }
};  
//predefined class contains 2-methods  
class A
{
    void m1(){System.out.println("A m1 method ");}
    void m2(){System.out.println("A m2 method");}
};  
//approach-1 extends the then override required methods  
class Test extends A
{
    void m1(){System.out.println("Test extends A --> m1 method");}
    public static void main(String[] args)
    {
        Test t = new Test(); t.m1();
    }
};  
//approach-2 don't extends the class declare anonymous inner class then override the required methods  
class Ratan
{
    A a = new A()
    {
        void m1(){System.out.println("Anonumous inner class m1 method");}
    }; //semicolon mandatory
```

```
public static void main(String[] args)
{
    Ratan r = new Ratan();
    r.a.m1();
}
};

//predefined class contains 2-methods
class A
{
    void m1(){System.out.println("A m1 method");}
    void m2(){System.out.println("A m2 method");}
};

//approach-1 extends the then override required methods
class Test extends A
{
    void m1(){System.out.println("Test extends A --> m1 method");}
    public static void main(String[] args)
    {
        Test t = new Test(); t.m1();
    }
};
//approach-2 don't extends the class declare anonymous inner class then override the required methods
class Ratan
{
    A a = new A()
    {
        void m1(){System.out.println("Anonomous inner class m1 method");}
        //semicolon mandatory
        public static void main(String[] args)
        {
            Ratan r = new Ratan();
            r.a.m1();
        }
    };
}
```

### ENUMARATION

1. This concept is introduced in 1.5 version
2. enumeration is used to declare group of named constant s.
3. we are declaring the enum by using enum keyword. For the enums the compiler will generate .classes

4.enum is a keyword and **Enum** is a class and every enum is directl child class of **java.lang.Enum** so it is not possible to inherit the some other class. Hence for the enum inheritance concept is not applicable  
 5. by default enum constants are **public static final**

```
enum Heroin
{
    Samantha,tara,ubanu ; public static final smantha;
}
enum Week
{
    public static final tara;
    Public static final ubanu;
}
```

#### **EX:-calling of enum constants individually**

```
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);
    }
};
```

#### **EX:-**

1. printing the enumeration constants by using for-each loop.
2. values() methods are used to print all the enum constants.
3. ordinal() is used to print the index values of the enum constants.

```
enum Heroin
{
    samantha,tara,anu;
}
class Test
{
    public static void main(String... ratan)
    {
        Heroin[] s=Heroin.values();
        for (Heroin s1:s)
        {
            System.out.println(s1+"----"+s1.ordinal());
        }
    }
};
```

1. inside the enum it is possible to declare constructors. That constructors will be eecuted for each and every constant. If we are declaring 5 constants then 5 times constructor will be executed.
2. Inside the enum if we are declaring only constants the semicolon is optional.
3. Inside the enum if we are declaring group of constants and constructors at that situation the group of constants must be first line of the enum must ends with semicolon.

#### **Ex :-Semicolan optional**

```
enum Heroin
{
    samantha,tara,anu,ubanu
}
```

```
class Test
{
    public static void main(String... ratan)
    {
        Heroin s=Heroin.samantha;
    }
};
```

```

        }
    }
}

class Test
{
    public static void main(String... ratan)
    {
        Heroin s=Heroin.samantha;
    }
}

```

**Ex:- semicolon mandatory**

```

enum Heroin
{
    samantha,tara,anu,ubanu;
    Heroin()
    {
        System.out.println("ratan sir");
    }
}

```

**Ex:- constructors with arguments**

```

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());
        }
    }
};

```

**Ex:-inside the enum it is possible to provide main method.**

```

enum Heroin
{
    samantha,tara,anu;
    public static void main(String[] args)
    {
        System.out.println("enum main method");
    }
}
class Test
{
    public static void main(String... ratan)
    {
        Heroin[] s=Heroin.values();
        for (Heroin s1:s)
        {
            System.out.println(s1+"-----"+s1.ordinal());
        }
    }
};

```

**Ex:- inside the enums it is possible to declare group of constants and constructors and main method**

```

enum Heroin
{
    //group of constants
    ANUSHKA,UBANU,DEEPIKA;
}

```

```
//constructor
Heroin()
{
    System.out.println("ratan");
}
//enum main method
public static void main(String[] args)
{
    System.out.println("enum main method");
}//end main
}//end enum
class Test
{
    public static void main(String[] args)
    {
        //accessing enum constants
        Heroin[] h = Heroin.values();
        for (Heroin h1 : h)
        {
            System.out.println(h1+"----"+h1.ordinal());
        }
    }
}//end main
}//end class
```

## Collections framework (java.util)

### Pre-requisite topics for Collections framework:-

- 1) AutoBoxing.
- 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.

- Collection frame contains group of classes and interfaces by using these classes & interfaces we are representing group of objects as a single entity.
- Collection is sometimes called a **container**. And it is object that groups multiple elements into a **single unit**.
- Collections are used to store, retrieve ,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`

Note :- The root interface of Collection framework is **Collection** it contains 15 methods so all implementation classes are able to use that methods.

```
public abstract int size();
public abstract boolean isEmpty();
public abstract boolean contains(java.lang.Object);
public abstract java.util.Iterator<E> iterator();
public abstract java.lang.Object[] toArray();
public abstract <T extends java.lang.Object> T[] toArray(T[]);
public abstract boolean add(E);
public abstract boolean remove(java.lang.Object);
```

```

public abstract boolean containsAll(java.util.Collection<?>);
public abstract boolean addAll(java.util.Collection<? extends E>);
public abstract boolean removeAll(java.util.Collection<?>);
public abstract boolean retainAll(java.util.Collection<?>);
public abstract void clear();
public abstract boolean equals(java.lang.Object);
public abstract int hashCode();

```

*The interface contains abstract method and for that interfaces object creation is not possible hence think about implementation classes of that interfaces.*

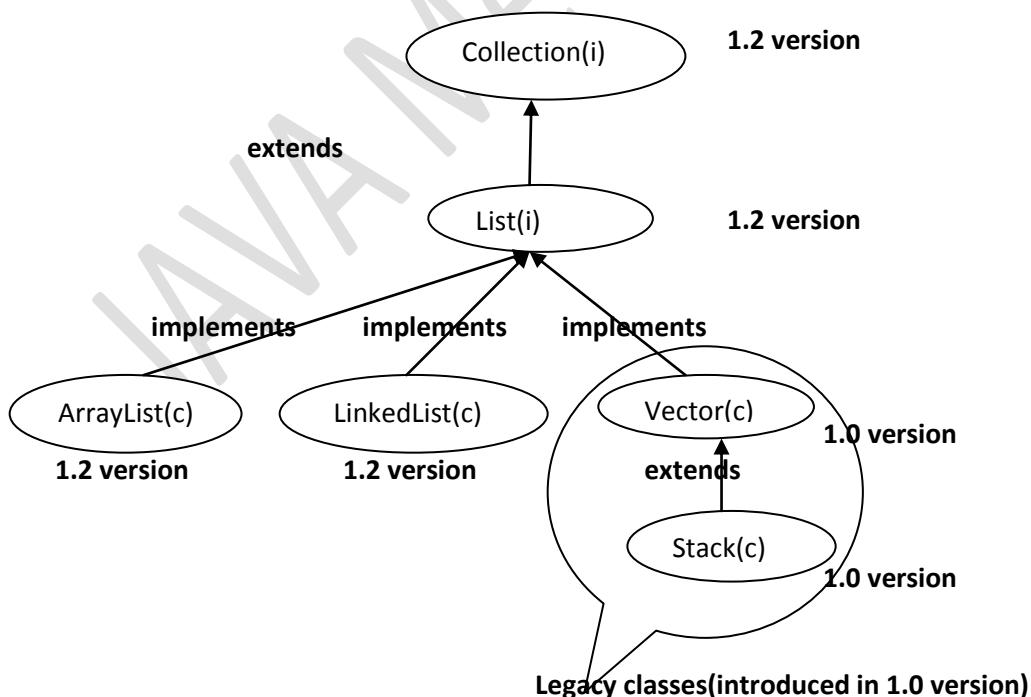
#### Collection vs Collections:-

**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.**

#### Characteristics of Collection frame work classes:-

- 1) The collection framework classes are introduced in different Versions.
- 2) Heterogeneous data allowed or not allowed.  
*All classes allowed heterogeneous data except two classes*  
i. TreeSet      ii. TreeMap
- 3) Null insertion is possible or not possible.
- 4) Insertion order is preserved or not preserved.  
*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*
- 5) Collection classes' methods are synchronized or non-synchronized.
- 6) Duplicate objects are allowed or not allowed.  
*add(e1)*  
*add(e1)*
- 7) Collections classes underlying data structures.
- 8) Collections classes supported cursors.



I -----→Interface

c-----→class

**Legacy class:-** The java classes which are introduced in 1.0 version are called legacy classes.

Ex :- *Vector, Stack, HashTable.....etc*

#### **Implementation classes of List interface :-**

- 1) ArrayList      2) LinkedList    3) Vector 4) Stack

#### **List interface common properties:-**

- 1) All List interface implementation classes allows null insertion.
- 2) All classes allows duplicate objects.
- 3) All classes preserved insertion order.

#### **Java.util.ArrayList:-**

*ArrayList* is implementing List interface it widely used class in projects because it is providing functionality and flexibility

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) Inside ArrayList we can insert Null objects.
- 4) ArrayList preserved Insertion order it means whatever the order we inserted the data in the same way output is printed.
  - a. Input -→e1 e2 e3   output -→e1 e2 e3      ***insertion order is preserved***
  - b. Input --→e1 e2 e3   output --→e1 e3 e2      ***insertion order is not- preserved***
- 5) ArrayList methods are non-synchronized methods.
- 6) Duplicate objects are allowed.
- 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:-**

***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***

It is possible to create ArrayList with initial capacity

```
ArrayList al = new ArrayList ( int initial-capacity);
```

Adding one collection data into another collection(Vector data intoArrayList) use following constructor.

```
ArrayList al = new ArrayList(Collection c);
```

#### **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
```

#### **Example :-Collections vs Autoboxing**

upto 1.4 version by using wrapper classes, create objects then add that objects in 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);
    }
}
```

```
}
```

But from 1.5 version onwards autoboxing concept is introduced so add the primitive value directly that is automatically converted into wrapper objects format.

```
import java.util.ArrayList;
class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(10);      //primitive int value --->Integer Object conversion //AutoBoxing
        al.add('a');    //primitive char value --->Integer Object conversion //AutoBoxing
        al.add(10.5);   //primitive double value --->Integer Object conversion //AutoBoxing
        System.out.println(al);
    }
}
```

#### **Example:-Basic operations of ArrayList**

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList al =new ArrayList();
        al.add("A");
        al.add("B");
        al.add('a');
        al.add(190);
        al.add(null);
        System.out.println(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(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(al);
    }
}
```

- ✓ In above example when we are adding primitive char value **al.add('c')** in ArrayList that value is automatically converted **Character** object format because ArrayList is able to store objects that is called AutoBoxing.
- ✓ When we print the ArrayList data for every object internally it is calling **toString()** method.

**Example :-** in Collection framework when we remove the data by using 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);
```

In above example if u want remove **10** object by using object name then we are using below code.

```
al.remove(10);
```

But whenever we are writing above code then JVM treats that 10 is index value hence it is generating exception **java.lang.IndexOutOfBoundsException: Index: 10, Size: 3**

To overcome above limitation if we want 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);
```

#### Example-2:- ArrayList vs toString()

##### Emp.java:-

```
class Emp
{
    //instance variables
    int eid;
    String ename;
    Emp(int eid,String ename)
    {//conversion of local - instance
    this.eid=eid;
    this.ename=ename;
    }
}
```

```
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() --->it execute Integer class toString()
        al.add('a');          //toString() --->it execute Character class toString()
        al.add(e1);           //toString() --->it executes Object class toString()
        al.add(s1);           //toString() --->it executes Object class toString()
        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);
            }
        }
    }
}
```

##### Student.java

```
class Student
{
    //instance variables
    int sid;
    String sname;
    Student(int sid,String sname)
    {//conversion of local to instance
    this.sid=sid;
    this.sname = sname;
    }
```

} }

### **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");
            }
        };
        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);
    }
}

```

**All collection classes are having 2-versions:-**

- 1) Normal version(no type safety ).
- 2) Generic version.(type safety )

**Note :-**

in java it is recommended to use generic version of collections class to store specified type of data.

**Syntax:-**

```
ArrayList<type-name> al = new ArrayList<type-name>();
```

**Examples:-**

ArrayList<Integer> al = new ArrayList<Integer>();	//store only Integer objects
ArrayList<String> al = new ArrayList<String>();	//store only String objects
ArrayList<Student> al = new ArrayList<Student>();	//store only Student objects
ArrayList<product> al = new ArrayList<product>();	//store only produce objects

**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');
al.add(10.5);
System.out.println(al);
```

- 2) 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 holding different types of Objects.**

```
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);
        al.add(true);
        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 holding only Integer data.**

```

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);
        }
    }
}

```

**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]
```

**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 fallowing 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)***

*Example:-*

```
ArrayList al = new ArrayList(); //non- synchronized version of ArrayList  
List l = Collections.synchronizedList(al); // synchronized version of ArrayList
```

```
HashSet 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
```

### Conversion of Arrays to ArrayList & ArrayList to Arrays:

#### Conversion of ArrayList to String array by using toArray( T ) method of ArrayList class:-

```
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);
        //normal approach to print the data
        System.out.println(a[0]);
        System.out.println(a[1]);
        System.out.println(a[2]);
        System.out.println(a[3]);
        //for-each loop to print the data
        for (String s:a)
        {
            System.out.println(s);
        }
    }
}
```

#### Example :- 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);
        }
    }
}
```

**Example :-**

```

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());
    }
}
}
}
}

```

**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);
}
}
}

```

we are able to retrieve objects from collection classes in three ways:-

- 1) By using for-each loop.
- 2) By using cursors.
- 3) By using get() method.

#### Cursors:-

Cursors are used to retrieve the Objects from collection classes.

There are three types of cursors present in the java.

1. Enumeration
2. Iterator
3. ListIterator

#### Applying Enumeration cursor on ArrayList:-

```
import java.util.*;
class ArrayListDemo
{
    public static void main(String[] args)
    {
        ArrayList<String> al = new ArrayList<String>();
        al.add("anu");
        al.add("Sravya");
        al.add("ratan");
        al.add("natraj");
        Enumeration<String> e = Collections.enumeration(al);
        while (e.hasMoreElements())
        {
            String str = e.nextElement();
            System.out.println(str);
        }
    }
}
```

#### 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 ascenning 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 decening order");
for (String str2 :al)
{
    System.out.println(str2);
}

}
}

```

### Comparable vs Comparator :-

Note :- it is possible to sort String and all wrapper objects because these objects are implementing Comparable interface.

- ❖ If we want to sort user defined class Emp based on eid or ename then your class must implements Comparable interface.
- ❖ Comparable present in java.lang package it contains only one method **compareTo(obj)**  
**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.
- ❖ By using comparable it is possible to sort the objects by using only one instance variable either eid or ename.

### Emp.java:-

```

class Emp implements Comparable
{
    int eid;
    String ename;

```

```

    Emp(int eid,String ename)
    {
        this.eid=eid;
        this.ename=ename;
    }
    /* it is sorting the data by using ename instance variable
    public int compareTo(Object o)
    {
        Emp e = (Emp)o;
        return ename.compareTo(e.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;
        }
    }
}

```

**Another format:-**

```

class Emp implements Comparable<Emp>
{
    *****
    public int compareTo(Emp e)
    {
        *****
    }
}

```

**Test.java:-**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        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);
        }
    }
}

```

}

#### Java.utilComparator :-

- ✓ 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);
}
```

✓

#### 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;
        }
    }
}
```

#### EnnameComp.java:-

```
import java.util.Comparator;
class EnnameComp implements Comparator
{
    public int compare(Object o1, Object o2)
    {
        Emp e1 = (Emp)o1;
        Emp e2 = (Emp)o2;
        return (e1.ename).compareTo(e2.ename);
        //return -(e1.ename).compareTo(e2.ename); //print data descending order
    }
}
```

```

    }
}

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 project level 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)
    {
        ****
        ****
    }
}
```

### Property

#### 1. Sorting logics

#### Comparable

- 1) *Sorting logics must be in the class whose class objects are sorting.*
- 2) **Int compareTo(Object o1)**  
*This method compares this object with o1 object and returns a integer. Its value has following meaning*  
**positive** – this object is greater than o1  
**zero** – this object equals to o1  
**negative** – this object is less than o1
- 3) **Collections.sort(List)**  
*Here objects will be sorted on the basis of CompareTo method*
- 4) Java.lang

#### 2. Sorting method

#### Comparator

- I. *Sorting logics in separate class hence we are able to sort the data by using different attributes.*
- II. **int compare(Object o1, Object o2)**  
*This method compares o1 and o2 objects. and returns a integer. Its value has following meaning.*  
**positive** – o1 is greater than o2  
**zero** – o1 equals to o2  
**negative** – o1 is less than o1
- III. **Collections.sort(List, Comparator)**  
*Here objects will be sorted on the basis of Compare method in Comparator*
- IV. **Java.util**
- V. **Collections.sort(List, Comparator)**  
*Here objects will be sorted on the basis of Compare method in Comparator*

### 3. Method calling to perform sorting

### 4. package

### 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]
    }
}

```

#### Enumeration:-0

1. Enumeration cursor introduced in 1.0 version hence it is called legacy cursor.

2. Enumeration is a legacy cursor it is used to retrieve the objects from only legacy classes (vector, Stack, HashTable...) hence it is not a universal cursor.

3. to retrieve Object from collection classes Enumeration Object uses two methods.

**public abstract boolean hasMoreElements();**

This method is used to check whether the collection class contains Objects or not, if collection class contains objects return true otherwise false.

**public abstract E nextElement();**

This method used to retrieve the objects from collection classes.

4. **elements()** method used to get Enumeration Object.

```
Vector v =new Vector();
v.addElement(10);
v.addElement(20);
v.addElement(30);
Enumeration e = v.elements();
```

5. **Normal version of Enumeration**

```
Vector v =new Vector();
v.addElement(10);
v.addElement(20);
v.addElement(30);
Enumeration e = v.elements();
while (e.hasMoreElements())
{//typecasting required
Integer i = (Integer)e.nextElement();
System.out.println(i);
}
```

**Generic version of Enumeration**

```
Vector v =new Vector();
v.addElement(10);
v.addElement(20);
v.addElement(30);
Enumeration<Integer> e = v.elements();
while (e.hasMoreElements())
{
Integer i = e.nextElement();
//type casting is not required
System.out.println(i);
}
```

6. By using this cursor it is possible to read the data only, it is not possible to update the data and not possible to remove the data.

7. By using this cursor we are able to retrieve the data only in forward direction.

**Iterator:-**

- 1) Iterator cursor introduced in 1.2 versions.

- 2) Iterator is a universal cursor applicable for all collection classes.

- 3) **iterator()** method is used to get Iterator object.

```
ex:- ArrayList al =new ArrayList();
al.add(10);
al.add(20);
al.add(30);
Iterator itr = al.iterator();
```

- 4) The Iterator object uses three methods to retrieve the objects from collections classes.

```
public abstract boolean hasNext();
```

This is used to check whether the Objects are available in collection class or not , if available returns true otherwise false.

```
public abstract E next();
```

This method used to retrieve the objects.

```
public abstract void remove();
```

This method is used to remove the objects from collections classes.

5) **Normal version of Iterator**

```
ArrayList al = new ArrayList();
al.add(10);
al.add(20);
al.add(30);
Iterator itr = al.iterator();
while (itr.hasNext())
{
    Integer i = (Integer)itr.next();
}
//normal version typecasting is required
System.out.println(i);
}
```

**Generic version of ArrayList**

```
ArrayList<Integer> al = new
ArrayList<Integer>();
al.add(10);
al.add(20);
al.add(30);
Iterator<Integer> itr = al.iterator();
while (itr.hasNext())
{
    Integer i = itr.next();
}
//generic version type casting is not required
System.out.println(i);
}
```

- 6) By using Iterator cursor we are able to perform read and remove operations but it is not possible to perform update operation.
- 7) By using Iterator we are able to read the data only in forward direction.

**Property**

1. Purpose

**2. Legacy or not**

- 1) Used to retrieve the data from collection classes.

**3. Applicable for which type of classes**

- 2) Introduced in 1.0 version it is legacy

**4. How to get the object**

- 3) It is used to retrieve the data from only legacy classes like vector, Stack...etc

- 4) Get the Enumeration Object by using elements() method.

```
Vector v = new Vector();
v.add(10);
v.add(20);
Enumeration e = v.elements();
```

**5. How many methods**

- 5) It contains two methods

- a. hasMoreElements()
- b. nextElement()

**6. Operations**

- 6) only read operations.

**7. Cursor moment**

- 7) Only forward direction.

**8. Universal cursor or not**

- 8) Not a universal cursor because it is applicable for only legacy classes.

**9. Class or interface**

- 9) Interface

**10. Versions supports**

- 10) It supports both normal and generic version.

**Enumeration****ListIterator:-**

1. ListIterator cursor Introduced in 1.2 version
2. This cursor is applicable only for List type of classes(ArrayList,LinkedList,Vector,Stack...etc) hence it is not a universal cursor.
3. listIterator() method used to get ListIterator object  
**ex:-** `LinkedList ll = new LinkedList();`  
`ll.add(10);`  
`ll.add(20);`  
`ll.add(30);`  
`ListIterator lstr = ll.listIterator();`
4. ListIterator contains following methods
  - `public abstract boolean hasNext();`---->to check the Objects
  - `public abstract E next();` ---->to retrieve the objects top to bottom
  - `public abstract boolean hasPrevious();` --->check the objects in previous direction
  - `public abstract E previous();` ---->to retrieve the Objects from previous direction
  - `public abstract int nextIndex();`---->to get index
  - `public abstract int previousIndex();`--->to get the index from previous direction.
  - `public abstract void remove();` --->to remove the Objects
  - `public abstract void set(E);` ----->to replace the particular Object
  - `public abstract void add(E);`---->to add new Objects
5. By using this cursor we are able to read & remove & update the data.
6. By using this cursor we are able to read the data both in forward and backward direction.

#### Differences between Enumeration & Iterator & ListIterator:-

Characterstics

1. Version

2.Legacy or not

	legacy	1.2 version	
<b>3. How to get object</b>	By using elements() method	Not a legacy	<b><u>ListIterator</u></b> 1.2 version
<b>4. How many versions</b>	Normal version & generic version	By using iterator() method	Not a legacy
<b>5. Methods</b>	2 methods	Normal version & generic version	By using listIterator() method
<b>6. Operations</b>	Read operations	3 methods	Normal version & generic version
<b>7. Class or interface</b>	interface	Read & remove	9 methods
<b>8. Applicable to which type of classes</b>	legacy classes	interface	Read & remove & update
<b>9. Cursor moment</b>	only forward direction	all collection classes	interface
<b>10. Return which object</b>	implementation class of Enumeration Interface	only forward direction. Implementation class of Iterator interface.	only for List type of classes both forward and backward directions. Implementation class of ListIterator interface.

**Enumeration**

1..0 version

**Iterator****Synchronized Collection Methods of Collections class:-**

*Collections.synchronizedSortedSet(SortedSet<T> s)*  
*Collections.synchronizedSortedMap(SortedMap<K,V> m)*  
*Collections.synchronizedSet(Set<T> s)*  
*Collections.synchronizedCollection(Collection<T> c)*  
*Collections.synchronizedList(List<T> list)*  
*Collections.synchronizedMap(Map<K,V> m)*

**Application shows implementation class object of cursor interfaces(Enumeration ,Iterator,ListIterator)**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Vector v = new Vector();
        v.addElement(10);
        v.addElement(20);
        v.addElement(30);
        //it returns implementation class object of Enumeration interface
        Enumeration e = v.elements();
        System.out.println(e.getClass().getName());

        //it returns implementation class object of Iterator interface
        Iterator itr = v.iterator();
        System.out.println(itr.getClass().getName());

        //it returns implementation class object of ListIterator interface
        ListIterator lstr = v.listIterator();
        System.out.println(lstr.getClass().getName());
    }
};

D:\>java Test
java.util.Vector$1
java.util.Vector$Iter
java.util.Vector$ListIter
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 cursors.
- 3) By using get() method.

#### **Example application:-**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        //ArrayList able to store only String Objects
        ArrayList<String> al =new ArrayList<String>();
        al.add("A");
        al.add("B");
        al.add("C");
        al.add("D");
        al.add(null);

        //1st approach to print Collection class elements (by using for-each loop)
        for (String a : al)
        {
            System.out.println(a);
        }

        //2nd approach to print Collection class elements (by using cursors)
        Iterator itr1 = al.iterator();//normal version of Iterator
    }
};

```

```

        while (itr1.hasNext())
        {
            String str =(String)itr1.next();//type casting required because normal version
            System.out.println(str);
        }
        Iterator<String> itr2 = al.iterator();//generic version of Iterator
        while (itr2.hasNext())
        {
            String str =itr2.next();//type casting not required because generic version
            System.out.println(str);
        }
//3rd approach to print objects by using get() method
int size = al.size();
for (int i=0;i<size;i++)
{
    System.out.println(al.get(i));
}
}

```

**Example :-[ListIterator VS ArrayList]**

<i>add(E);</i>	<i>-----&gt;to add the Object</i>
<i>remove(java.lang.Object);</i>	<i>-----&gt;to remove the object</i>
<i>size();</i>	<i>-----&gt;to find size</i>
<i>isEmpty();</i>	<i>-----&gt;returns true if empty otherwise false</i>
<i>clear();</i>	<i>-----&gt;to remove all objects</i>
<i>addAll();</i>	<i>-----&gt;to add one collection object into another collection</i>
<i>removeAll();</i>	<i>-----&gt;to remove all the elements of particular collection</i>
<i>retainAll();</i>	<i>-----&gt;to remove all elements except particular collections</i>

**Example:-**

```

import java.util.*;
class Emp
{
    //instance variables
    int eid;
    String ename;
    Emp(int eid ,String ename) //local variables
    {
        //conversion of local variables to instance variables
        this.eid = eid;
        this.ename = ename;
    }
    public static void main(String[] args)
    {
        Emp main1 = new Emp(111,"ratan");
        Emp main2 = new Emp(222,"Sravya");
        Emp main3 = new Emp(333,"aruna");
        Emp sub1 = new Emp(444,"anu");
        Emp sub2 = new Emp(555,"banu");

        ArrayList<Emp> al1 = new ArrayList<Emp>(); //generic version of ArrayList
        al1.add(main1);
        al1.add(main2);
    }
}

```

```

al1.add(main3);

ArrayList<Emp> al2 = new ArrayList<Emp>(); //generic version of ArrayList
al2.add(sub1);
al2.add(sub2);
al1.addAll(al2); //add all objects of al2 into al1
al1.remove(main2); //it removes main1 object from al1
al1.removeAll(al2); //it removes all objects of al2
//it checks whether main2 available or not
System.out.println(al1.contains(main2));
System.out.println(al1.size()); //print size

//printing elements by using for-each loop
for (Emp o1 : al1)
{
    System.out.println(o1.eid+" "+o1.ename);
}

System.out.println("printing objects in forward direction");
ListIterator<Emp> lstr = al1.listIterator(); //generic version of ListIterator cursor
while (lstr.hasNext())
{
    Emp e = lstr.next(); //type casting not required because it is generic version
    System.out.println(e.eid+" "+e.ename);

}
System.out.println("printing objects in backward direction");
while (lstr.hasPrevious())
{
    Emp e1 = lstr.previous();
    System.out.println(e1.eid+" "+e1.ename);
}
}
}

```

**ArrayList vs Iterator vs ListIterator:-**

```

import java.util.*;
class Student
{
    //instance variables
    int sno;      String sname;  int smarks;
    Student(int sno, String sname, int smarks) //local variables
    { //conversion of local variables to instance variables
        this.sno = sno;
        this.sname = sname;
        this.smarks = smarks;
    }
    public static void main(String[] args)
    {
        Student s1 = new Student(111, "ratan", 100);
        Student s2 = new Student(222, "anu", 99);
        Student s3 = new Student(333, "aruna", 98);
    }
}

```

```

Student s4 = new Student(444,"pavan",97);
ArrayList<Student> ar1 = new ArrayList<Student>();//generic version of ArrayList
ar1.add(s1);
ar1.add(s2); //ar1 contains 2 objects
ArrayList<Student> ar2 = new ArrayList<Student>();//generic version of ArrayList
ar2.add(s3);
ar2.add(s4); //ar2 contains 2 object
ar1.addAll(ar2); //it's adding all objects ar2 into ar1
ar1.removeAll(ar2); //it removes all objects of ar1 except ar2
Iterator<Student> itr = ar1.iterator(); //generic version of Iterator

System.out.println("using Iterator retrieving objects only forward direction");
while (itr.hasNext())
{Student st =itr.next(); //type casting is not required because it is generic
 System.out.println(st.sno+" "+st.sname+" "+st.smarks);
}
ListIterator<Student> ltr = ar1.listIterator(); //generic version of ListIterator

System.out.println("print objects forward direction by using ListIterator");
while (ltr.hasNext())
{ Student stt = ltr.next(); //type casting is not required because it is generic
 System.out.println(stt.sno+" "+stt.sname+" "+stt.smarks);
}
System.out.println("print objects backward direction by using ListIterator");
while (ltr.hasPrevious())
{ Student sttt = ltr.previous();
 System.out.println(sttt.sno+" "+sttt.sname+" "+sttt.smarks);
}
}

```

**LinkedList:-**

Class LinkedList extends AbstractSequentialList implements List, Deque, Queue

- 1) Introduced in 1.2 v
- 2) Heterogeneous objects are allowed.
- 3) Null insertion is possible.
- 4) Insertion order is preserved
- 5) LinkedList methods are non-synchronized method
- 6) Duplicate objects are allowed.
- 7) The underlying data structure is double linkedlist.
- 8) cursors :- Iterator, ListIterator **Ex:-LinkedList with generics.**

if we are using AL to store the data at that situation whenever we are adding one new object middle of ArrayList then number of shift operations are required it will degrade the performance

Ex:-

```

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]

```

#### **Vector:- (legacy class introduced in 1.0 version)**

- 1) Introduced in 1.0 v legacy classes.
- 2) Duplicate objects are allowed.
- 3) Null insertion is possible.
- 4) Heterogeneous objects are allowed.
- 5) The underlying data structure is growable array.
- 6) Insertion order is preserved.
- 7) Every method present in the Vector is synchronized and hence vector object is Thread safe.
- 8) Cursors: - Enumeration.

#### **Vector :-**

##### **Case:-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

```

##### **Case 2:-**

*It is possible to create vector with specified capacity by using following 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
```

### Case 3:-

*It is possible to create vector with initial capacity and providing increment capacity by using following 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
```

*Adds the specified component to the end of this vector*

**public void addElement(Object obj) --→Vector class method**

*Appends the specified element to the end of this Vector*

**public boolean add(Object o) --→List interface method**

```
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>();
        v.add("ratan");
        v.add("aruna");
        v.add("Sravya");
        v.addElement("ccc");
        System.out.println(v);
        System.out.println(v.size());
        v.add(2,"xxx");
        v.remove("Sravya"); //removed based on object
        v.remove(1); //removed based on index
```

```

        System.out.println(v);
        v.addAll(al);           //adding ArrayList data into Vector
        System.out.println(v);
        v.removeAll(al);        //it removes all objects of al
        System.out.println(v);
        System.out.println(v.firstElement());      //to retrieve first element
        System.out.println(v.lastElement());        //to retrieve last element
    }
}

D:\vikram>java Test
[ratan, aruna, Sravya, ccc]
4
[ratan, xxx, ccc]
[ratan, xxx, ccc, no1, no2]
[ratan, xxx, ccc]
ratan
ccc

```

**Example :-**

```

//product.java
class Product
{
    //instance variables
    int pid;
    String pname;
    double pcost;
    Product(int pid,String pname,double pcost)      //local variables
    {
        //conversion [passing local variable values to instance variable]
        this.pid = pid;
        this.pname = pname;
        this.pcost = pcost;
    }
};

//ArrayListDemo.java
import java.util.*;
class ArrayListDemo
{
    public static void main(String[] args)
    {
        Product p1 = new Product(111,"pen",1300);
        Product p2 = new Product(222,"laptop",13000);
        Product p3 = new Product(333,"bag",1000);
        Product p4 = new Product(444,"java",5000);
        Product p5 = new Product(555,".net",4000);
        Vector<Product> v = new Vector<Product>();
        v.addElement(p1);
        v.addElement(p2);
        v.addElement(p3);
        System.out.println("****Enumeration cursor only read operations****");
    }
};

```

```

Enumeration<Product> e = v.elements();
while (e.hasMoreElements())
{
    Product p = e.nextElement();
    System.out.println(p.pid+"---"+p.pname+"---"+p.pcost);
}
System.out.println("****Iterator cursor both read & remove operations****");
Iterator<Product> itr = v.iterator();
while (itr.hasNext())
{
    Product pp = itr.next();
    if ((pp.pname).equals("pen"))
        itr.remove();           //pen object removed
}

System.out.println("****ListIterator cursor read & remove & update operations****");
ListIterator<Product> lstr = v.listIterator();
lstr.add(p4);   //p4 object is added by ListIterator
while (lstr.hasNext())
{
    Product p = lstr.next();
    if (p.pid==333)
        lstr.remove(); //bag object removed
    if ((p.pname).equals("laptop"))
        lstr.set(p5); //laptop is replaced by .net
}

System.out.println("****printing remaining objects ****");
Iterator<Product> it = v.iterator();
while (it.hasNext())
{
    Product p = it.next();
    System.out.println(p.pid+"---"+p.pname+"---"+p.pcost);
}
}

Example:-
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);
    }
}

```

**Stack:- (legacy class introduced in 1.0 version)**

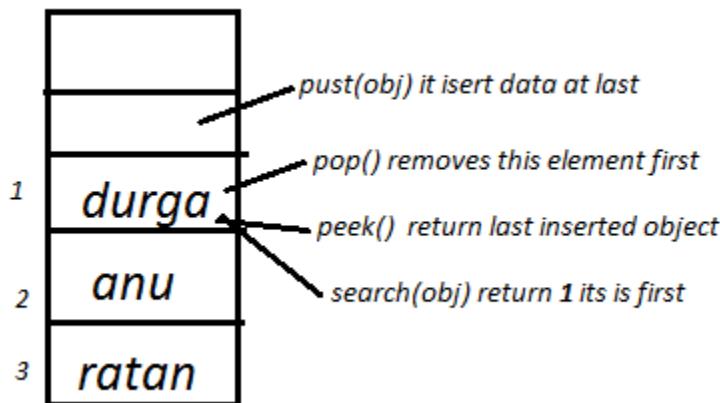
- 1) It is a child class of vector
- 2) Introduce in 1.0 v legacy class
- 3) It is designed for LIFO(last in fist order )

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());
    }
}

```



Example :-

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        String reverse="";
        Scanner s = new Scanner(System.in);
        System.out.println("enter input string to check palindrome or not");
        String str = s.nextLine();
        Stack stack = new Stack();

```

```

        for (int i=0;i<str.length();i++)
        {
            stack.push(str.charAt(i));
        }
        while (!stack.isEmpty())
        {
            reverse=reverse+stack.pop();
        }
        if (str.equals(reverse))
        {
            System.out.println("the input String palindrome");
        }
        else
        {
            System.out.println("the input String not- palindrome");
        }
    }
}

```

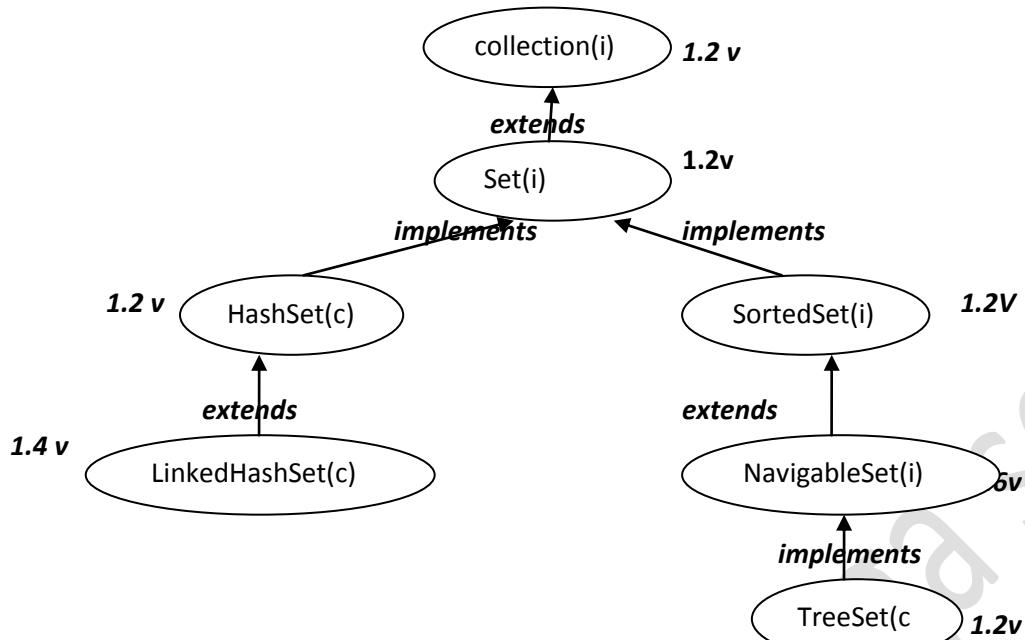
**5) by using ListIterator we are able to read & remove & update the data.**

1. It is applicable for only list type of objects.
2. By using this it is possible to read the data upate the data and delete data also.
3. By using listIterator() method we are getting ListIterator object

**EmpBean.java:-**

```

public class EmpBean implements Comparable<EmpBean>
{
    private int eid;
    private String ename;
    public void setEid(int eid)
    {
        this.eid=eid;
    }
    public void setEname(String ename)
    {
        this.ename=ename;
    }
    public int getEid()
    {return eid;
    }
    public String getEname()
    {return ename;
    }
    public int compareTo(EmpBean o)
    {
        if (eid==o.eid)
        {return 0;
        }
        if (eid>o.eid)
        {return 1;
        }
        else{return -1;
    }
}
};
```

**HashSet:-**

- 1) Introduced in 1.2 v.
- 2) Duplicate objects are not allowed if we are trying to insert duplicate values then we won't get any compilation errors and won't get any Execution errors simply add method return old value.
- 3) Null insertion is possible but if we are inserting more than one null it return only one null value.
- 4) Heterogeneous objects are allowed.
- 5) The underlying data structure is `HashTable`.
- 6) It does not maintain any order the elements are returned in any random order .[Insertion order is not preserved].
- 7) Methods are non-synchronized.
- 8) cursor : Iterator

**Example:-**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        //HashSet object creation
        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);
        }
    }
}
  
```

```

        }
    }
}

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        HashSet<String> h = new HashSet<String>();
        h.add("ratan");
        h.add("anu");
        h.add("Sravya");
        HashSet<String> hsub = new HashSet<String>();
        hsub.add("no1");
        hsub.add("no2");
        hsub.addAll(h);
        System.out.println(hsub.contains("anu"));
        hsub.remove("anu");
        System.out.println(hsub.containsAll(h));
        System.out.println(hsub);
        hsub.removeAll(h);
        System.out.println(hsub);
        hsub.retainAll(h);
        System.out.println(hsub);
    }
}

```

**LinkedHashSet:-**

1. Introduced in 1.4 version and It is a child class of HashSet.
2. Duplicate objects are not allowed if we are trying to insert duplicate values then we won't get any compilation errors and won't get any Execution errors simply add method return false.
3. Null insertion is possible.
4. Heterogeneous objects are allowed
5. The underlying data structure is LinkedList & hashTable.
6. Insertion order is preserved.
7. Methods are non-synchronized.
8. Cursors :- Iterator.

**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");
        //retrieving objects by using Iterator cursor
        Iterator<String> itr = h.iterator();
    }
}

```

```

        while (itr.hasNext())
        {String str = itr.next();
        System.out.println(str);
        }
//retrieving objects by using Enumeration cursor
Enumeration<String> e = Collections.enumeration(h);
while (e.hasMoreElements())
{    System.out.println(e.nextElement());
}
}

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        HashSet<String> h = new HashSet<String>();
        h.add("ratan");
        h.add("anu");
        h.add("Sravya");
//passing data from HashSet to LinkedHashSet
LinkedHashSet<String> lh = new LinkedHashSet<String>(h);
//lh.addAll(h);
lh.add("xxx");
lh.add("yyy");
System.out.println(lh);
//passing data from LinkedHashSet to HashSet
HashSet<String> hh = new HashSet<String>(lh);
//hh.addAll(lh);
hh.add("zzz");
System.out.println(hh);
    }
}

```

**TreeSet:-**

1. TreeSet is same as HashSet but TreeSet sorts the elements in ascending order but HashSet does not maintain any order.
2. The underlying data structure is Balanced Tree.
3. Insertion order is not preserved it is based some sorting order.
4. Heterogeneous data is not allowed.
5. Duplicate objects are not allowed
6. Null insertion is possible only once.

```

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);
}

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);
        System.out.println(t.size());
        t.remove("ratan");
        System.out.println(t.contains("ratan"));

        TreeSet<String> t1 = new TreeSet<String>(t);
        //t1.addAll(t);
        t1.add("xxx");
        t1.removeAll(t);
        System.out.println(t1);
    }
}

import java.util.*;
class Fruit
{
    public static void main(String[] args)
    {
        TreeSet<String> t = new TreeSet<String>(new MyComp());
        t.add("orange");
        t.add("bananna");
        t.add("apple");
        System.out.println(t);
    }
}

class MyComp implements Comparator<String>
{
    public int compare(String s1, String s2)
    {
        return s1.compareTo(s2); // [apple, bananna, orange]
        //return -s1.compareTo(s2); // [orange, bananna, apple]
    }
};

```

*Example :- Elimination duplicate objects*

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

```

class Test
{
    public static void main(String[] args)
    {
        String[] str={"ratan","anu","Sravya","anu"};
        //conversion of String[] to List
        List<String> l = Arrays.asList(str);
        //conversion of List to Set [it eliminates duplicates]
        TreeSet<String> t = new TreeSet<String>(l);
        System.out.println(t);
    }
}

```

**Example :-**

```

import java.util.Iterator;
import java.util.TreeSet;
public class Test {
    public static void main(String[] args) {
        // creating a TreeSet Object
        TreeSet <Integer>treeadd = new TreeSet<Integer>();
        // adding object in the tree set
        treeadd.add(10);
        treeadd.add(30);
        treeadd.add(70);
        treeadd.add(20);
        // create iterator Object
        Iterator iterator;
        iterator = treeadd.iterator();

        // displaying the Tree set data
        System.out.println("Tree set data in ascending order: ");
        while (iterator.hasNext()){
            System.out.println(iterator.next() + " ");
        }
    }
}

```

**Example :-**

```

public E first();
public E last();
public E lower(E);
public E higher(E);
public java/util/SortedSet<E> subSet(E, E);
public java/util/SortedSet<E> headSet(E);
public java/util/SortedSet<E> tailSet(E);
public E pollFirst();
public E pollLast();

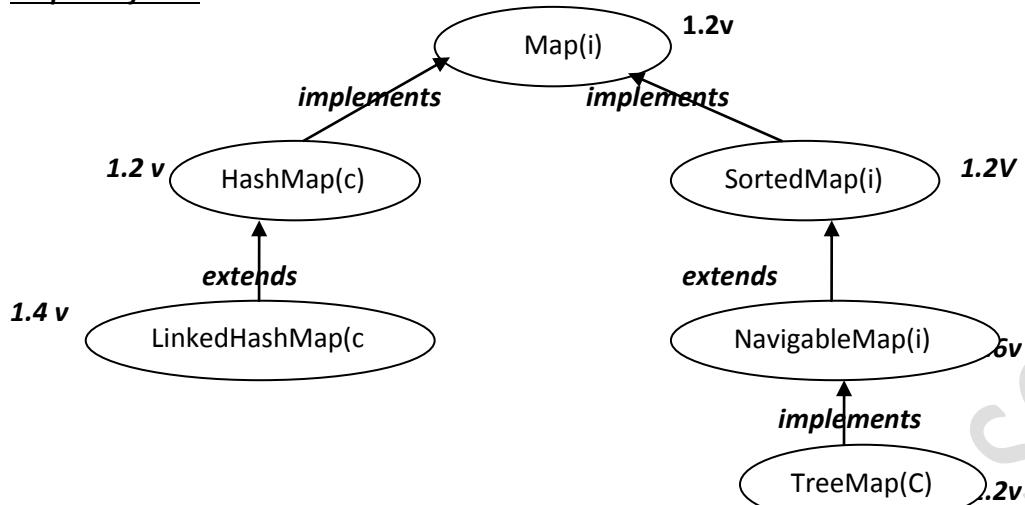
```

*it print first element*  
*it print last element*  
*it print lower object of specified object*  
*it print higher object of specified object*  
*it print subset*  
*it print specified object above objects*  
*it print specified objects below values*  
*it print and remove first*  
*it print and remove last.*

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

```
class Test
{
    public static void main(String[] args)
    {
        //creating TreeSet object
        TreeSet t=new TreeSet();
        //adding object in TreeSet
        t.add(50);           t.add(20);           t.add(40);
        t.add(10);           t.add(30);
        System.out.println(t);
        SortedSet s1=t.headSet(50);
        System.out.println(s1);      // [10,20,30,40]
        SortedSet s2=t.tailSet(30);
        System.out.println(s2);      // [30,40,50]
        SortedSet s3=t.subSet(20,50);
        System.out.println(s3);      // [20,30,40]
        System.out.println("last element="+t.last());
        System.out.println("first element="+t.first());
        System.out.println("lower element="+t.lower(50));
        System.out.println("higher element="+t.higher(20));
        System.out.println("print & remove first element="+t.pollFirst());
        System.out.println("print & remove last element="+t.pollLast());
        System.out.println("final elements="+t);
        System.out.println("TreeSet size="+t.size());
        System.out.println("TreeSet size="+t.remove(10));
        System.out.println("TreeSet size="+t.remove(30));
    }
}

D:\morn11>java Test
[10, 20, 30, 40, 50]
[10, 20, 30, 40]
[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=false
TreeSet size=true
```

**Map interface:-****Map:-**

1. Map is a child interface of collection.
2. Up to know we are working with single object and single value where as in the map collections we are working with two objects and two elements.
3. The main purpose of the collection is to compare the key value pairs and to perform necessary operation.
4. The key and value pairs we can call it as map Entry.
5. Both keys and values are objects only.
6. In entire collection keys can't be duplicated but values can be duplicate.

**HashMap:-**

- 1) interdicted 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.
- 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.
- 8) cursor :- Iterator.

```

//public java/util/Set<K> keySet(); method syntax
// public java/util/Collection<V> values(); method syntax
// public java/util/Set<java/util/Map$Entry<K, V>> entrySet(); method syntax
  
```

**Example :-**

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        //creation of HashMap Object
        HashMap h = newHashMap();
        h.put("ratan",111);           //h.put(key,value);
        h.put("anu",111);
    }
  
```

```

        h.put("banu",111);
        Set s1=h.keySet();           //used to get all keys
        System.out.println("all keys--->"+s1);
        Collection c = h.values();   //used to get all values
        System.out.println("all values--->"+c);
        Set ss = h.entrySet();       //it returns all entries nothing but [key,value]
        System.out.println("all entries--->"+ss);
        //get the Iterator Object
        Iterator itr = ss.iterator();
        while (itr.hasNext())
        {
            //next() method retrun first entry to represent that entry do typeCasting
            Map.Entry m= (Map.Entry)itr.next();
            System.out.println(m.getKey()+"---"+m.getValue()); //printing key and value
        }
    }
};

```

**LinkedHashMap:-**

- 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.
- 8) cursor :- Iterator

**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)
    {
        //creates LinkedList object with generic version
        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"));
//get Set interface before getting Iterator object
Set s = h.entrySet();
//creates iterator Object
Iterator itr = s.iterator();
while (itr.hasNext())
{//holding Entry by using Entry interface
Map.Entry m = (Map.Entry)itr.next();
Emp e = (Emp)m.getKey();           //getting Emp object
System.out.println(e.ename+"--"+e.eid);
Student ss = (Student)m.getValue(); //getting Student object
System.out.println(ss.sname+"--"+ss.sid);
}
}

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        HashMap h = new LinkedHashMap();
        h.put(111,"ratan");
        h.put(222,"anu");
        h.put(333,"sravya");
        System.out.println(h);

        Set s1 = h.keySet();
        System.out.println(s1);

        Collection c = h.values();
        System.out.println(c);

        Set s2 = h.entrySet();
        Iterator itr = s2.iterator();
        while (itr.hasNext())
        {
            Map.Entry m = (Map.Entry)itr.next();
            System.out.println(m.getKey()+"---"+m.getValue());
        }
    }
}

```

**HashTable:-**

1. Introduced in the 1.0 version it's a legacy class.
2. Every method is synchronized hence only one thread is allowed to access it is a Thread safe but performance is decreased.

3. Null insertion is not possible if we are trying to insert null values we are getting NullPointerException.

```
h.put(null,"ratan");
h.put("4",null);
```

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);
        }
    }
}
```

Example :-

We are able to add one class data into another class in two ways

- 1) Passing one class reference variable to another class

```
Hashtable h = new Hashtable();
```

```
HashMap<String, String> h1 = new HashMap<String, String>(h);
```

- 2) By using **putAll()** method

```
h1.putAll(h);
```

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

```

class Test
{
    public static void main(String[] args)
    {
        Hashtable<String, String> h = new Hashtable<String, String>();
        h.put("1", "one");
        h.put("2", "two");
        h.put("3", "three");
        //passing Hashtable data into HashMap
        HashMap<String, String> h1 = new HashMap<String, String>(h);
        //h1.putAll(h);
        h1.put("hm", "ratan");
        System.out.println(h1);
        //passing HashMap data into LinkedHashMap
        LinkedHashMap<String, String> lhm = new LinkedHashMap<String, String>(h1);
        //lhm.putAll(h1);
        lhm.put("lhm", "anu");
        System.out.println(lhm);
    }
}

```

**TreeMap:-****Example-1:-**

```

import java.util.TreeMap;
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("4", "no4");
        System.out.println(tsub);
    }
}

```

**Example -2:-**

```

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);
}
}

```

**Java.util.Properties:-****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());
        }
    }
}

```

**Comparable interface :-**

- Comparable interface used to perform sorting of user defined class objects.
- Comparable present in `java.lang` package and it contains only method  
**`public abstract int compareTo(Object obj-name);`**
- By using comparable We are able to sort the object by using single data member like **`sid,sname`**.
- String & all Wrapper classes are implement Comparable interface hence if we are storing these Objects these are comparable.

If first object sid value is greater than existing object then it returns positive//no change in data  
If the object sid values is less than existing object then it returns negative.//change location  
If any negative or both are equals then it returns zero. //no change in data

**Student.java**

class Student implements Comparable

```
{      int sid;
String sname;
Student(int sid,String sname)//local var
{ this.sname=sname; this.sid=sid;
}
public int compareTo(Object obj)
{
    Student s = (Student)obj;
    if (sid>s.sid)
    {return 1;
    }
    if (sid<s.sid)
    {return -1;
    }
    If(sid==0){
    return 0;
    }
}
Test.java:-
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<Student> al = new ArrayList<Student>();
        al.add(new Student(11,"ratan"));
        al.add(new Student(2,"Sravya"));
        al.add(new Student(333,"anu"));
        Collections.sort(al);
        Iterator<Student> itr =al.iterator();
        while (itr.hasNext())
        {
            Student s = itr.next();
            System.out.println(s.sid+"----"+s.sname);
        }
    }
import java.util.*;
class Comp implements Comparator
{
    public int compare(Object o1,Object o2)
    {
        EmpBean e1 = (EmpBean)o1;
        EmpBean e2 = (EmpBean)o2;
        if (e1.eid==e2.eid)
        {return 0;
    }
}
```

```
        }
        if(e1.eid>e1.eid)
        {return 1;
        }
        else{return -1;}
    }
}

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        TreeSet<EmpBean> s = new TreeSet<EmpBean>(new Comp());
        EmpBean e1 = new EmpBean();
        e1.setEid(111);
        e1.setEname("ratan");
        EmpBean e2 = new EmpBean();
        e2.setEid(22);
        e2.setEname("anu");

        s.add(e1);
        s.add(e2);

        for (EmpBean e:s)
        {System.out.println(e.eid+"---"+e.ename);
        }
    }
}

public class EmpBean implements Comparable<EmpBean>
{
    int eid;
    String ename;
    public void setEid(int eid)
    {
        this.eid=eid;
    }
    public void setEname(String ename)
    {
        this.ename=ename;
    }
    public int getEid()
    {return eid;
    }
    public String getEname()
    {return ename;
    }
    public int compareTo(EmpBean o)
```

```
{  
    if (eid==o.eid)  
    {return 0;  
    }  
    if (eid>o.eid)  
    {return 1;  
    }  
    else{return -1;}  
}  
};
```

## ***networking***

### **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.net package provide support for the TCP (Transmission Control Protocol), UDP(user data gram protocol) protocols.
- 4) In the network we are having to components
  - a. Sender
  - b. Receiver

**Sender/source:** -the person who is sending the data is called sender.

**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 everyone says client and server.
  - I. Client
  - II. Server

**Client:-**the person who is sending the request and taking the response is called client.

**Server:-** the person who is taking the request and sending the response is called server.

### **Categories of network:-**

We are having two types of networks

- 1) Per-to-peer network.
- 2) Client-server network.

#### **Client-server:-**

In the client server architecture always client system behaves as a client and server system behaves 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.

##### **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.

##### **Extranet:-**

This is extension to the private network means other than the organization , authorized persons able to access.

#### **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.

#### **Protocol:-**

Protocol is a set of rules followed by the every computer present in the network this is useful to send the data physically from one place to another place in the network.

- TCP(Transmission Control Protocol)(connection oriented protocol)
- UDP (User Data Gram Protocol)(connection less protocol)
- Telnet
- SMTP(Simple Mail Transfer Protocol)
- IP (Internet Protocol)

#### **IP Address:-**

- 1) IP Address is a unique identification number given to the computer to identify it uniquely in the network.
- 2) The IP Address is uniquely assigned to the computer it is not duplicated.
- 3) The IP Address range is 0-255 if we are giving the other than this range that is not allowed.
- 4) We can identify the particular computer in the network with the help of IP Address.
- 5) The IP Address contains four digit number
  - a. 125.0.4.255----good
  - b. 124.654.5.6----bad
  - c. 1.2.3.4.5.6-----bad

- 6) 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

Ex:-

```
import java.net.*;
import java.io.*;
class Test
{
    public static void main(String[] args) throws Exception
    {
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("please enter site name");
        String sitename=br.readLine();

        InetAddress in=InetAddress.getByName(sitename);
        System.out.println("the ip address is:"+in);

    }
}
```

Compilation :- javac Test.java  
 Execution :- java Test  
[www.google.com](http://www.google.com)  
 The IP Address is:www.google.com/74.125.236.176

java Test  
[www.yahoo.com](http://www.yahoo.com)  
 The IP Address is: [www.yahoo.com/106.10.139.246](http://www.yahoo.com/106.10.139.246)

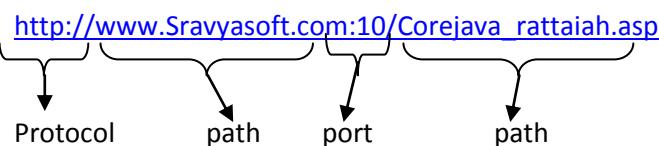
Java Test  
 Please press enter key then we will get IP Address of the system.  
 The IP Address is : local host/we are getting IP Address of the system

#### Note:-

If the internet is not available we are getting java.net.UnKnownHostException.

#### 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.
- 3) Example of URL is:-



The URL contains information like

- a. Protocol to use **http://**
  - b. Server name/IP address **[www.Sravyasoft.com](http://www.Sravyasoft.com)**
  - c. Port number of the particular application and it is optional(:10)
  - d. File name or directory name **Corejava\_rattaiah.asp**
- 4) To create the object for URL we have to use the following syntax
- a. URL obj=new URL(String protocol, String host, int port, String path);
  - b. URL obj=new URL(String protocol, String host, String path);

**Ex:-**

```
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)

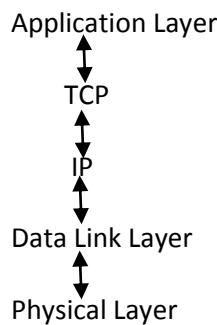
#### Connection Oriented:-

- a) In this type of communication we are using combination of two protocols TCP,IP.
- b) In this type of communication the main purpose of the TCP is transferred in the form of packets between the source and destination. And the main purpose of the IP is finding address of a particular system.

To achieve the following communication the java peoples are provided the following classes.

- a. Socket
- b. ServerSocket

#### Layers of the TCP/IP connection.



**Application Layer:-**

Takes the data from the application and sends it to the TCP layer.

**TCP Protocol:-**

It will take the data which is coming from Application Layer and divides it into small units called Packets. Then transfer those packets to the next layer called IP. The packet contains group of bytes of data.

**IP:-**

It will take the packets which are coming from TCP and prepare envelope called 'frames' hence the frame contains the group of packets. Then it will identify the particular target machine on the basis of the IP address and send those frames to the physical layer.

**Physical Layer:-**

Based on the physical medium it will transfer the data to the target machine.

**Connection Less :- (UDP)**

- 1) UDP is a protocol by using this protocol we are able to send data without using Physical Connection.
- 2) This is a light weight protocol because no need of the connection between the client and server.
- 3) This is very fast communication compared to the TCP/IP communication.
- 4) This protocol does not send the data in proper order there may be chance of missing the data.
- 5) This communication used to send the Audio and Video data if some bits are lost but we are able to see the video and images we are getting any problems.

To achieve the UDP communication the Java people 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.
- 4) Socket is a class present in the java.net package.
- 5) It is acting as a communicator between the client and server.
- 6) Whenever we want to send the data first we have to create a socket that acts as a medium.

**Create the socket**

- 1) `Socket s=new Socket(int IPAddress, int portNumber);`
  - a. `Socket s=new Socket("125.125.0.5",123);`

Server IP Address.      Server port number.
- 2) `Socket s=new Socket(String HostName, int PortNumber);`

a. Socket s=new Socket(Sravyasoft,123);

**Client.java:-**

```
import java.net.*;
import java.io.*;
class Client
{
    public static void main(String[] args) throws Exception
    {
        Socket s=new Socket("localhost",5555);
        String str="ratan from client";
        OutputStream os=s.getOutputStream();
        PrintStream ps=new PrintStream(os);
        ps.println(str);

        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 ");
        InputStream is=s.getInputStream();

        BufferedReader br=new BufferedReader(new InputStreamReader(is));
        String data=br.readLine();
        System.out.println(data);

        //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**

- ❖ Abstract Window Tool kit is an **API** it supports graphical user interface programming.
- ❖ By using **java.awt** package we are able to develop the components like
  - **TextFiled , Label, Button ,Checkbox ,RadioButton.....etc**
- ❖ 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:-**

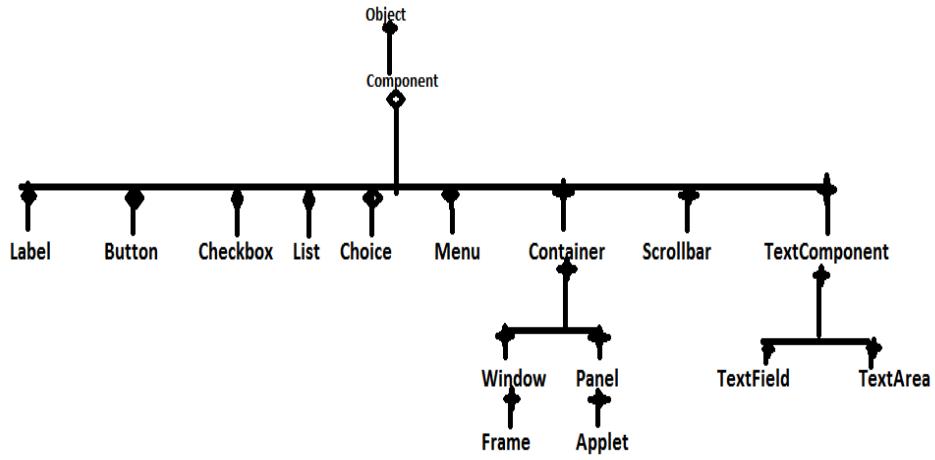
The event nothing but a action generated on the component or the change is made on the state of the object.

Ex:-

Button clicked, Checkboxchecked, Itemselected in the list, Scrollbar scrolled horizontal/vertically.

**Classes of AWT:-**

The classes present in the AWT package.



#### Java.awt.Frame:-

Frame is a Basic component in AWT, it contains other components like Button, TextField...etc.  
There are two approaches to create a frame

- 1) **By extending Frame class.**
- 2) **By creating Object of Frame class.**

**Constructors:-**

```
Frame f=new Frame();
Frame f=new Frame("MyFrame");
```

**Characteristics of the Frame:-**

- ✓ When we create a Frame class object the Frame will be created automatically with invisible mode so to provide 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 initial size of the Frame is 0 pixel heights & 0 pixel width so it is not visible to use. To provide particular size to the Frame we have to 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 :- creation of Frame By creating Object of Frame class.**

```
import java.awt.*;
class Demo
{
    public static void main(String[] args)
    {
        Frame f=new Frame();           //frame creation
        f.setVisible(true);            //now frame is visible by default not visible
        f.setSize(400,400);           //set the size of the frame
        f.setBackground(Color.red);    //set the background
        f.setTitle("myframe");        //set the title of the frame
    }
};
```

**\*\*\*CREATION OF FRAME BY TAKING USER DEFINED CLASS\*\*\***

```
import java.awt.*;
class MyFrame extends Frame
{
    MyFrame()
    {
        setVisible(true);
        setSize(500,500);
        setTitle("myframe");
        setBackground(Color.red);
    }
}
class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};
```

**To display text on the screen:-**

1. If you want to display some textual message or some graphical shapes on the Frame then we have to override paint(), which is present in the Frame class.  
**public void paint(Graphics g)**

2. To set a particular font to the text, we have to use Font class present in java.awt package

```
Font f=new Font(String type,int style,int size);
```

```
Ex: Font f= new Font("arial",Font.Bold,30);
```

Ex :-

```
import java.awt.*;
class Test extends Frame
{
    public static void main(String[] args)
    {
        Test t=new Test();
        t.setVisible(true);
        t.setSize(500,500);
        t.setTitle("myframe");
        t.setBackground(Color.red);
    }
    public void paint(Graphics g)
    {
        Font f=new Font("arial",Font.ITALIC,25);
        g.setFont(f);
        g.drawString("hi ratan how r u",100,100);
    }
}
```

#### Note:-

1. When we create a MyFrame class constructor, jvm executes MyFrame class constructor just before this JVM has to execute Frame class zero argument constructor.
2. In Frame class zero argument constructor repaint() method will be executed, it will access predefined Frame class paint() method. But as per the requirement overriding paint() method will be executed.
3. Therefore the paint() will be executed automatically at the time of Frame creation.

#### Preparation of the components:-

##### Label: -

- 1) Label is a constant text which is displayed along with a TextField or TextArea.
- 2) Label is a class which is present in java.awt package.
- 3) To display the label we have to add that label into the frame for that purpose we have to use add() method present in the Frame class.

##### Constructor:-

```
Label l=new Label();
Label l=new Label("user name");
```

Ex :-

```
import java.awt.*;
class Test
```

```
{
```

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

```
{
```

```
Frame f=new Frame();
```

```
f.setVisible(true);
```

```
f.setTitle("ratan");
```

```
f.setBackground(Color.red);
```

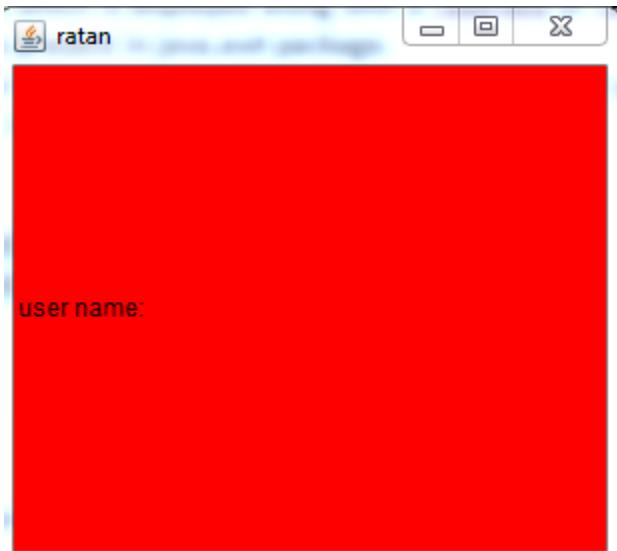
```
f.setSize(400,500);
```

```
Label l=new Label("user name:");
```

```
f.add(l);
```

```
}
```

```
}
```

**TextField:-**

- TextField is an editable area and it is possible to provide single line of text.
- Enter Button doesn't work on TextField.
  - To set Text to the textarea we have to use **t.setText("Sravya");**
  - To get the text from the TextArea we have to use **String s=t.getText();**
  - To append text into the TextArea **t.appendText("ratan");**

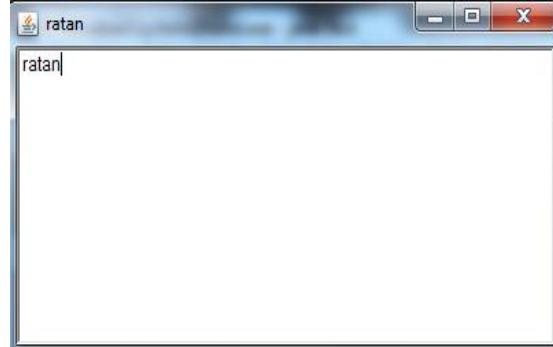
**Constructor:-**

```
TextFiled tx=new TextFiled();
TextField tx=new TextField("ratan");
```

**Ex :-**

```
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);

        //TextField tx=new TextField(); empty TextField
        TextField tx=new TextField("ratan");
        //TextField with data
        f.add(tx);
    }
}
```



**TextArea:-** *TextArea is a Editable Area&enter button will work on TextArea.*

```
TextArea t=new TextArea();
TextArea t=new TextArea(int rows,int columns);
```

- ✓ To set Text to the textarea we have to use **ta.setText("Sravya");**

- ✓ To get the text from the TextArea we have to use **String s=ta.getText();**
- ✓ To append the text into the TextArea use **ta.appendText("ratan");**

```
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);
        f.setLayout(new FlowLayout());
        Label l=new Label("user name:");
        TextArea tx=new TextArea(4,10); //4 character height 10 character width
        tx.appendText("ratan");
        tx.setText("aruna");
        System.out.println(tx.getText());
        f.add(l);
        f.add(tx);
    }
}
```



**Choice:** -List is allows to select multiple items but choice is allow to select single Item.

**Choice ch=new Choice();**

**Methods :-**

1. To add items to the choice use **add()** method.
2. To remove item from the choice based on String use **remove()** method. **choice.remove("HYD");**
3. To remove the item based on the index position use **choice.remove(2);**
4. To remove the all elements **ch.removeAll();**
5. To inset the data into the choice based on the particular position. **choice.insert(2,"ratan");**
6. To get selected item from the choice use **String s=ch.getSelectedItem();**
7. To get the selected item index number use **int a=ch.getSelectedIndex();**

ex:-

```
import java.awt.*;
```

```

class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);
        Choice ch=new Choice();
        ch.add("c");
        ch.add("cpp");
        ch.add("java");
        ch.add(".net");
        ch.remove(".net");
        ch.remove(0);
        ch.insert("ratan",0);
        f.add(ch);
        System.out.println(ch.getItem(0));
        System.out.println(ch.getSelectedItem());
        System.out.println(ch.getSelectedIndex());
        //ch.removeAll();
    }
}

```

D:\bank>javac Test.java

D:\bank>java Test

ratan  
ratan  
0



**List:** List is providing list of options to select. Based on your requirement we can select any number of elements. To add the List to the frame we have to use add() method.

#### CONSTRUCTOR:-

**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 single item.

**List l=new List(5,true);** It will display the five items and it is allow selecting the multiple items.

#### Methods:-

- ✓ To add the elements to the List use **list.add("c");**
- ✓ To add the elements to the List at specified index **list.add("ratan",0);**
- ✓ To remove element from the List use **list.remove("c");**
- ✓ To get selected item from the List use **String x=l.getSelectedItem();**
- ✓ To get selected items from the List we have to use **String[] x=s.getSelectedItems()**

```

import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);
        f.setLayout(new FlowLayout());
        List l=new List(4,true);
        l.add("c");           l.add("cpp");           l.add("java");           l.add(".net");
        l.add("ratan");       l.add("arun",0);       l.remove(0);          f.add(l);
    }
}

```

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



**Checkbox:** - The user can select more than one checkbox at a time.

- 1) Checkbox cb1=new CheckBox();
- 2) Checkbox cb2=new CheckBox("MCA");
- 3) Checkbox cb3=new CheckBox("BSC",true);

**Methods:-**

1. To set a label to the CheckBox explicitly use `cb.setLabel("BSC")`;
2. To get the label of the checkbox use `String str=cb.getLabel()`;
3. To get state of the CheckBox use `Boolean b=ch.getState()`;

Ex:-

```
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);
        Checkbox cb1=new Checkbox("BTECH",true);
        f.add(cb1);
        System.out.println(cb1.getLabel());
        System.out.println(cb1.getState());
    }
}
```

```
D:\bank>javac Test.java
D:\bank>java Test
BTECH
true
```


**RADIO BUTTON:**

- ✓ AWT does not provide any predefined support to create RadioButtons.
- ✓ It is possible to select Only item from group of items and we are able 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,false);
CheckBox cb2=new CheckBox("female",cg,false);
```

**Methods:-**

- 1) To get the status of the RadioButton use ***String str=cb.getState();***
- 2) To get Label of the RadioButton use ***String str=getLabel().***

Ex:-

```
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);
        CheckboxGroup cg=new CheckboxGroup();
        Checkbox cb1=new Checkbox("male",cg,true);
        f.add(cb1);
        System.out.println(cb1.getLabel());
        System.out.println(cb1.getState());
    }
}
```

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

D:\bank>java Test
male
true

```

**Layout Managers:-**

```
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);
        Label l1=new Label("user name:");
        TextField tx1=new TextField();
        Label l2=new Label("password:");
        TextField tx2=new TextField();
        Button b=new Button("login");
        f.add(l1);           f.add(tx1);           f.add(l2);
        f.add(tx1);          f.add(b);
    }
}
```

**Event delegation model:-**

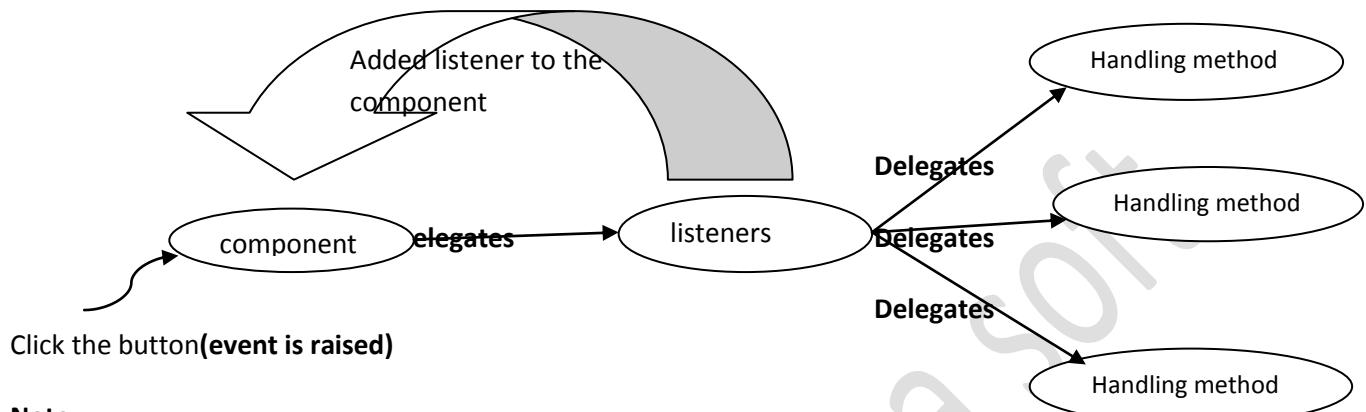
1. When we create a component the components visible on the screen but it is not possible to perform any action for example button.
2. Whenever we create a Frame it can be minimized and maximized and resized but it is not possible to close the Frame even if we click on Frame close Button.
3. The Frame is a static component so it is not possible to perform actions on the Frame.
4. To make static component into dynamic component we have to add some actions to the Frame.
5. To attach actions to the Frame component we need event delegation model.

**Whenever we click on button no action will be performed clicking like this is called event.**

**Event:** - Event is nothing but a particular action generated on the particular component.

1. When an event generates on the component the component is unable to respond because component can't listen the event.
2. To make the component listen the event we have to add listeners to the component.
3. Wherever we are adding listeners to the component the component is able to respond based on the generated event.
4. A listener is a interface which contain abstract methods and it is present in java.awt.event package
5. The listeners are different from component to component.

A component delegate event to the listener and listener is designates the event to appropriate method by executing that method only the event is handled. This is called Event Delegation Model.



**Note:** -

To attach a particular listener to the Frame we have to use following method

**Public void AddxxxListener(xxxListener e)**

Where xxx may be ActionListener,windowListener

The Appropriate Listener for the Frame is “windowListener”

**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:-**

GUI Component	Event Name	Listner Name	Lisener Methods
---------------	------------	--------------	-----------------

1.Frame	Window Event	Window Listener	1. Public Void WindowOpened(WindowEvent e) 2. Public Void WindowActivated(WindowEvent e) 3. Public Void WindowDeactivated(WindowEvent e) 4. Public Void WindowClosing(WindowEvent e) 5. Public Void WindowClosed(WindowEvent e) 6. Public Void WindowIconified(WindowEvent e) 7. Public Void WindowDeiconified(WindowEvent e)
2.Textfield	ActionEvent	ActionListener	1. Public Void Actionperformed(ActionEvent ae)
3.TextArea	ActionEvent	ActionListener	1. Public Void Actionperformed(ActionEvent ae)

4.Menu	ActionEvent	ActionListener	1.Public Void Actionperformed(ActionEvent ae)
5.Button	ActionEvent	ActionListener	1.Public Void Actionperformed(ActionEvent ae)
6.Checkbox	ItemEvent	ItemListener	1.Public Void ItemStatechanged(ItemEvent e)
7.Radio	ItemEvent	ItemListener	1.Public Void ItemStatechanged(ItemEvent e)
8.List	ItemEvent	ItemListener	1.Public Void ItemStatechanged(ItemEvent e)
9.Choice	ItemEvent	ItemListener	1.Public Void ItemStatechanged(ItemEvent e)
10.Scrollbar	AdjustmentEvent	AdjustmentListener	1.Public Void AdjustementValueChanged(AdjustementEvent e)
11.Mouse	MouseEvent	MouseListener	1.Public Void MouseEntered(MouseEvent e) 2.Public Void MouseExited(MouseEvent e) 3.Public Void MousePressed(MouseEvent e) 4.Public Void MouseReleased(MouseEvent e) 5.Public Void MouseClicked(MouseEvent e)
12.Keyboard	KeyEvent	KeyListener	1.Public Void KeyTyped(KeyEvent e) 2.Public Void KeyPressed(KeyEvent e) 3.Public Void KeyReleased(KeyEvent e)

\*\*\*PROVIDING CLOSING OPTION TO THE FRAME\*\*\*

```

import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame
{
    MyFrame()
    {
        this.setSize(400,500);
        this.setVisible(true);
        this.setTitle("myframe");
        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.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\*\*\*

```
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("rattaiah");
        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();
}
};
```

Note :- by using WindowAdaptor class we can close the frame. Internally WindowAdaptor class implements WindowListener interface. Hence WindowAdaptor class contains empty implementation of abstract methods.

\*\*\*PROVIDING CLOSEING OPTION THE FRAME\*\*\*\*\*

```
import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame
{   MyFrame()
    {
        this.setVisible(true);
        this.setSize(500,500);
        this.setBackground(Color.red);
        this.setTitle("rattaiah");
        this.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {
                System.exit(0);
            }
        });
    }
}
class FrameEx
{
    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();
}
};

*****LAYOUT MACHANISMS  FLOWLAYOUT*****
import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame
{
    Label l1,l2;
    TextField tx1,tx2;
    Button b;
    MyFrame()
    {
        this.setVisible(true);
        this.setSize(340,500);
        this.setBackground(Color.green);
        this.setTitle("rattaiah");
        l1=new Label("user name:");
        l2=new Label("password:");
        tx1=new TextField(25);
        tx2=new TextField(25);
        b=new Button("login");
        tx2.setEchoChar('*');
        this.setLayout(new FlowLayout());
        this.add(l1); this.add(tx1); this.add(l2); this.add(tx2);
        this.add(b);
    }
}

class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};

*****BORDERLAYOUT*****
import java.awt.*;
class MyFrame extends Frame
{
    Button b1,b2,b3,b4,b5;
    MyFrame()
    {
        this.setBackground(Color.green);
        this.setSize(400,400);
        this.setVisible(true);
        this.setLayout(new BorderLayout());
        b1=new Button("Boys");
        b2=new Button("Girls");
        b3=new Button("management");
        b4=new Button("Teaching Staff");
        b5=new Button("non-teaching staff");
        this.add("North",b1); this.add("Center",b2);
        this.add("South",b3); this.add("East",b4);
        this.add("West",b5);
    }
}

class Demo
```

```
{     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();
    }
};
```

```
*****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);
        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**

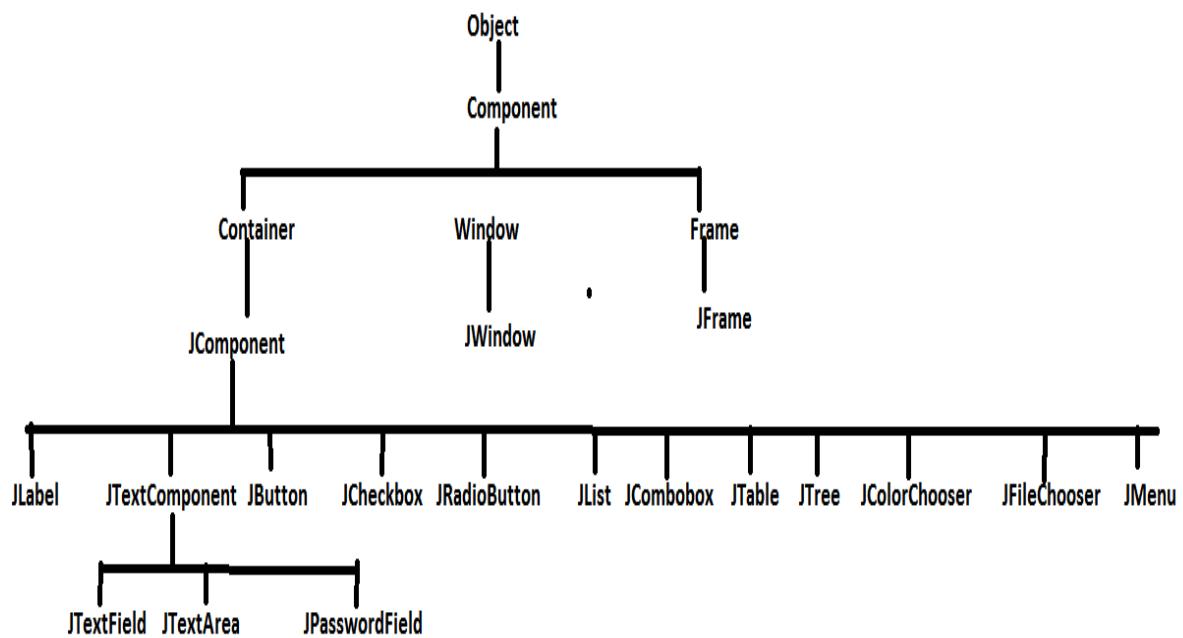
1. Sun Micro Systems introduced AWT to prepare GUI applications.
2. awt components not satisfy the client requirement.
3. An alternative to AWT NetscapeCommunication has provided set of GUI components in the form of IFC(Internet Foundation Class).
4. IFC also provide less performance and it is not satisfy the client requirement.
5. In the above contest[sun+Netscape] combine and introduced common product to design GUI applications.

### **Differences between awt and Swings:**

1. AWT components are heavyweight component but swing components are light weight component.
2. AWT components consume more number of system resources Swings consume less number of system resources.
3. AWT components are platform dependent but Swings are platform independent.

4. AWT is provided less number of components where as swings provides more number of components.
5. AWT doesn't provide Tooltip Test support but swing components have provided Tooltip test support.
6. in awt for only window closing : windowListenerwindowAdaptor  
In case of swing use small piece of code.
  - i. f.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);
7. AWT will not follow MVC but swing follows MVC Model View Controller It is a design pattern to provide clear separation b/w controller part,model part,view part.
  - a. Controller is a normal java class it will provide controlling.
  - b. View part provides presentation
  - c. Model part provides required logic.
8. In case of AWT we will add the GUI components in the Frame directly but Swing we will add all GUI components to panes to accommodate GUI components.

**Classes of swing:-**



Example :-

```
import java.awt.*;
import javax.swing.*;
class MyFrame extends JFrame
{
    JLabel l1,l2,l3,l4,l5,l6,l7;
    JTextField tf;
    JPasswordField pf;
    JCheckBox cb1,cb2,cb3;
    JRadioButton rb1,rb2;
    JList l;
    JComboBox cb;
    JTextArea ta;
    JButton b;
    Container c;
    MyFrame() //constructor
    {
        this.setVisible(true);
        this.setSize(150,500);
        this.setTitle("SWING GUI COMPONENTS EXAMPLE");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        c=this.getContentPane();
        c.setLayout(new FlowLayout());
        c.setBackground(Color.green);
        l1=new JLabel("User Name");
        l2= new JLabel("password");
        l3= new JLabel("Qualification");
        l4= new JLabel("User Gender");
        l5= new JLabel("Technologies");
        l6= new JLabel("UserAddress");
        l7= new JLabel("comments");
        tf=new JTextField(15);
        tf.setToolTipText("TextField");
        pf=new JPasswordField(15);
        pf.setToolTipText("PasswordField");
        cb1=new JCheckBox("BSC",false);
        cb2=new JCheckBox("MCA",false);
        cb3=new JCheckBox("PHD",false);
        rb1=new JRadioButton("Male",false);
        rb2=new JRadioButton("Female",false);
        ButtonGroup bg=new ButtonGroup();
        bg.add(rb1);           bg.add(rb2);
        String[] listitems={"cpp","c","java"};
        l=new JList(listitems);
        String[] cbitems={"hyd","pune","bangalore"};
        cb=new JComboBox(cbitems);
        ta=new JTextArea(5,20);
        b=new JButton("submit");
        c.add(l1);           c.add(tf);           c.add(l2);           c.add(pf);
```

```
c.add(l3);           c.add(cb1);           c.add(cb2);           c.add(cb3);
c.add(l4);           c.add(rb1);           c.add(rb2);           c.add(l5);
c.add(l);            c.add(l6);            c.add(cb);            c.add(l7);
c.add(ta);          c.add(b);             c.add(l);             c.add(l7);
}
}

class SwingDemo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};

*****JCOLORCHOOSER*****
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=ContentPane();
        cc=new JColorChooser();
        cc.getSelectionModel().addChangeListener(this);
        c.add(cc);
    }
    public void stateChanged(ChangeEvent c)
    {
        Color color=cc.getColor();
        JFrame f=new JFrame();
        f.setSize(400,400);
        f.setVisible(true);
        f.getContentPane().setBackground(color);
    }
}
class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};

*****JFILECHOOSER*****
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
```

```
class MyFrame extends JFrame implements ActionListener
{
    JFileChooser fc;
    Container c;
    JLabel l;
    JTextField tf;
    JButton b;
    MyFrame()
    {
        this.setVisible(true);
        this.setSize(500,500);
        this.setTitle("SWING GUI COMPONENTS EXAMPLE");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        c=ContentPane();
        l=new JLabel("Select File:");
        tf=new JTextField(25);
        b=new JButton("BROWSE");
        this.setLayout(new FlowLayout());
        b.addActionListener(this);
        c.add(l);           c.add(tf);           c.add(b);
    }
    public void actionPerformed(ActionEvent ae)
    {
        class FileChooserDemo extends JFrame implements ActionListener
        {
            FileChooserDemo()
            {
                Container c=ContentPane();
                this.setVisible(true);
                this.setSize(500,500);
                fc=new JFileChooser();
                fc.addActionListener(this);
                fc.setLayout(new FlowLayout());
                c.add(fc);
            }
            public void actionPerformed(ActionEvent ae)
            {
                File f=fc.getSelectedFile();
                String path=f.getAbsolutePath();
                tf.setText(path);
                this.setVisible(false);
            }
        }
        new FileChooserDemo();
    }
}
class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};
```

```
*****JTABLE*****
import javax.swing.*;
import java.awt.*;
import javax.swing.table.*;
class Demo1
{
    public static void main(String[] args)
    {JFrame f=new JFrame();
    f.setVisible(true);
    f.setSize(300,300);
    Container c=f.getContentPane();
    String[] header={"ENO","ENAME","ESAL"};
    Object[][] body={{"111","aaa",5000},{"222","bbb",6000}, {"333","ccc",7000}, {"444","ddd",8000}};
    JTable t=new JTable(body,header);
    JTableHeader th=t.getTableHeader();
    c.setLayout(new BorderLayout());
    c.add("North",th);
    c.add("Center",t);
    }
}
```

```
*****APPLET*****
import java.awt.*;
import java.applet.*;
public class Demo2 extends Applet
{
    public void paint(Graphics g)
    {
        Font f=new Font("arial",Font.BOLD,20);
        g.setFont(f);

        g.drawString("Sravya Software Solutions",100,200);
    }
};
```

#### Configuration of Applet:-

```
<html>
<applet code="Demo2.class" width="500" height="500">
</applet>
</html>
```

```
*****INIT() START() STOP() DESTROY()*****
import java.awt.*;
import java.applet.*;
```

```
public class Demo3 extends Applet
{
    String msg="";
    public void paint(Graphics g)
    {
        Font f=new Font("arial",Font.BOLD,20);
        g.setFont(f);
        g.drawString("Sravya Software Solutions "+msg,100,200);
    }
    public void init()
    {
        msg=msg+"initialization"+" ";
    }
    public void start()
    {
        msg=msg+"starting"+" ";
    }
    public void stop()
    {
        msg=msg+"stoping";
    }
    public void destroyed()
    {
        msg=msg+"destroyed";
    }
}
<html>
<applet code="Demo3.class" width="500" height="500">
</applet>
</html>
```

## INTERNATIONALIZATION (i18N)

**i18N enables the application to support in different languages.**

- Internationalization is also called as i18n because in between I & n 18 words 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 ResourceBundle we are able to get data from 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.**

<i>Application.properties</i>	default properties file(base properties file)
<i>Application_fr_FR.properties</i>	French properties file
<i>Allication_ratan_RATAN.properties</i>	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*

***Allication\_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 fr  
 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:-**

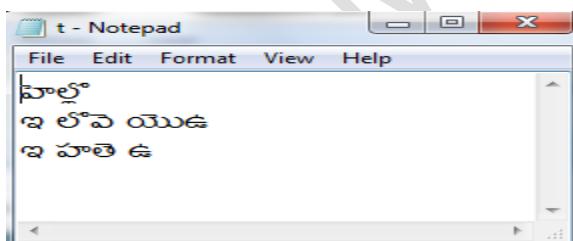
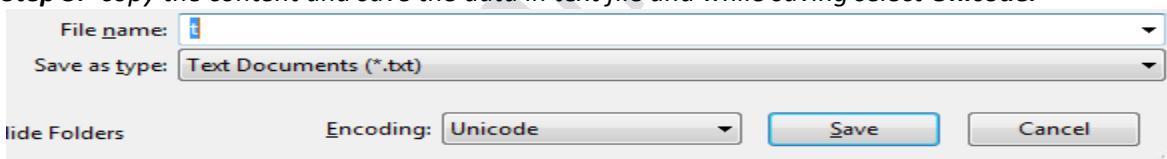
Step 1:- download Unicode editor from internet [www.higopi.com](http://www.higopi.com)

**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.

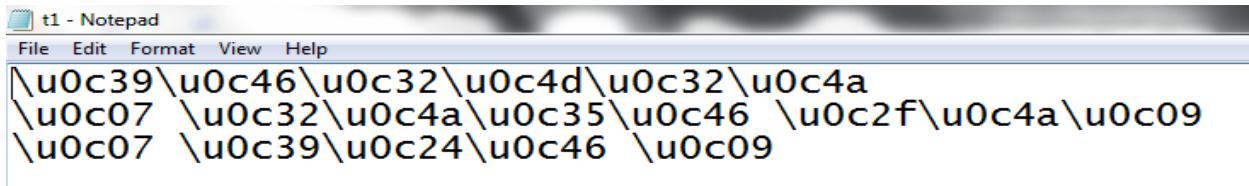
Step 3:- copy the content and save the data in text file and while saving select **Unicode**.



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**



t1 - Notepad  
File Edit Format View Help

```
\u0c39\u0c46\u0c32\u0c4d\u0c32\u0c4a
\u0c07 \u0c32\u0c4a\u0c35\u0c46 \u0c2f\u0c4a\u0c09
\u0c07 \u0c39\u0c24\u0c46 \u0c09
```

**Application :-****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\_tl\_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);
    }
}
```



**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));
}
};

D:\5batch>java Test
Nov 21, 2014
21 nov. 2014
21/11/14

```

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));
    }
};

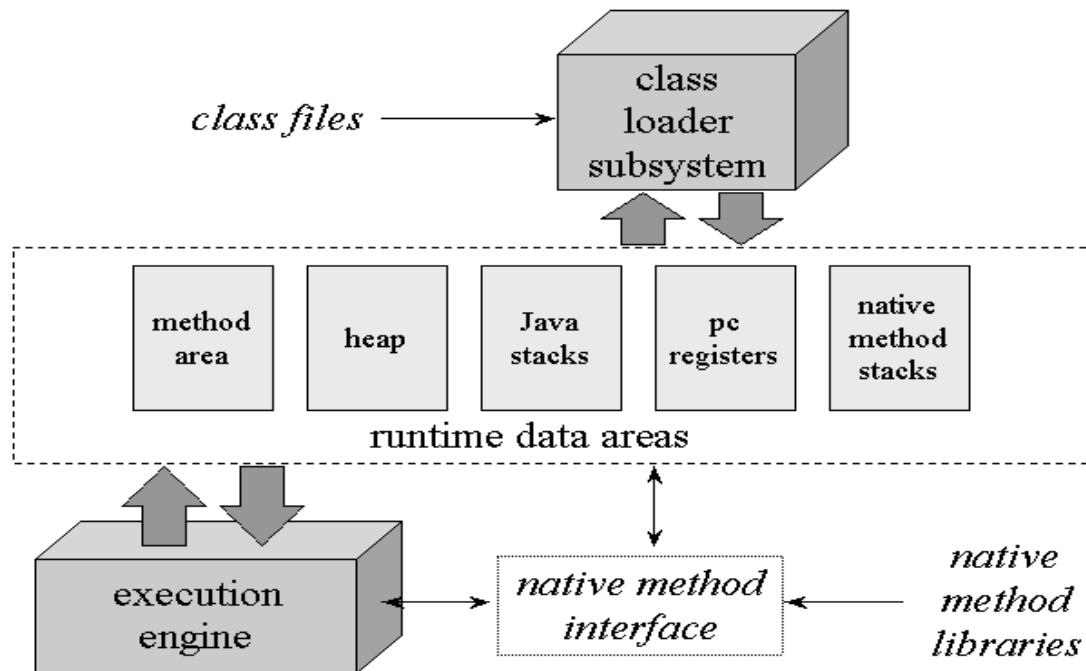
```

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.getDateInstance(DateFormat.FULL,DateFormat.FULL,Locale.getDefault());
        System.out.println(df1.format(d));
        DateFormat df2 = DateFormat.getDateInstance(DateFormat.FULL,DateFormat.FULL,Locale.FRENCH);
        System.out.println(df2.format(d));
    }
};
```

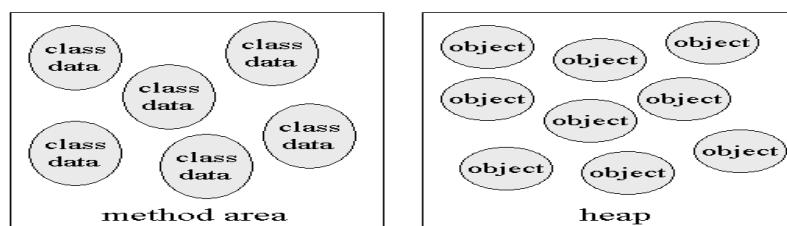
JVM Architecture:-Class loader subsystem:-

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.

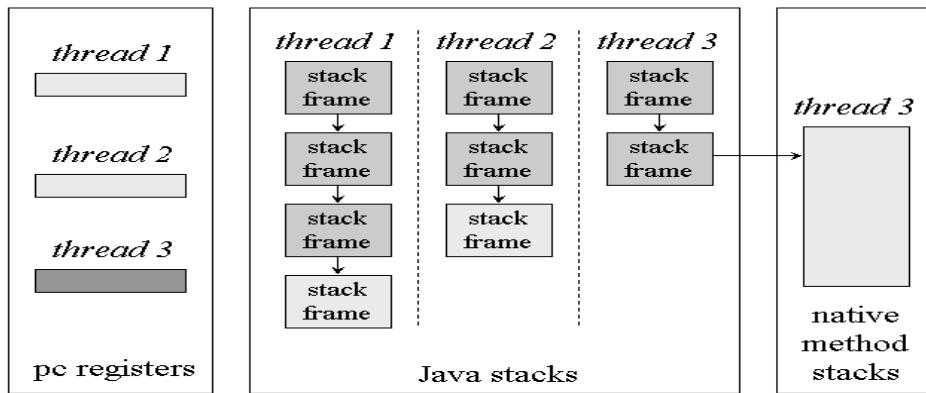
**Heap area:-**It is used to store the Objects.



**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.

**Modifiers summary:-**

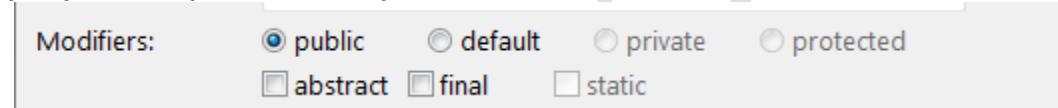
- In java no concept like "access specifiers and access modifiers" and only one concept is there modifiers concept.
- How many Modifiers in java means don't say 3 or 4 or 5 , in java 11 modifiers are there.
- The default modifier in java is "default".
- The most restricted modifier in java is private (only with in the class).
- The most accessible modifier in java is public (all package can access)
- The only one modifier applicable to local variables is "final".

**Proof 1:-**

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

**Compilation Error:-**

```
D:\morn11>javac Test.java
Test.java:1: modifier private not allowed here
private class Test
```

**proof 2:- in eclips IDE shows information like this.**

<b><u>modifier</u></b>	<b><u>classes</u></b>	<b><u>methods</u></b>	<b><u>variables</u></b>
public	yes	yes	yes
private	no	yes	yes
default	yes	yes	yes
protected	no	yes	yes
final	yes	yes	yes
abstract	yes	yes	no
strictfp	yes	yes	no
transient	no	no	yes
native	no	yes	no
static	no	yes	yes
synchronized	no	yes	no
volatile	no	no	yes

**Java is not a pure object oriented programming language:-**

**1)java supporting primitive datatypes there are not objects. To represent these primitives in the form of objects java having concept like Wrapper classes.**

```
Int a=10;
```

```
Boolean b=true;
```

**2)without creation of object we are able to access static members.**

```
class Test
```

```
{    static void m1()
    {    System.out.println("hi ratan");
    }
    public static void main(String[] args)
    {    Test.m1();
    }
}
```

**3) java is not supporting oops concepts like multiple inheritance& hybrid inheritance.**

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

**Different approaches to create objects in java:-**

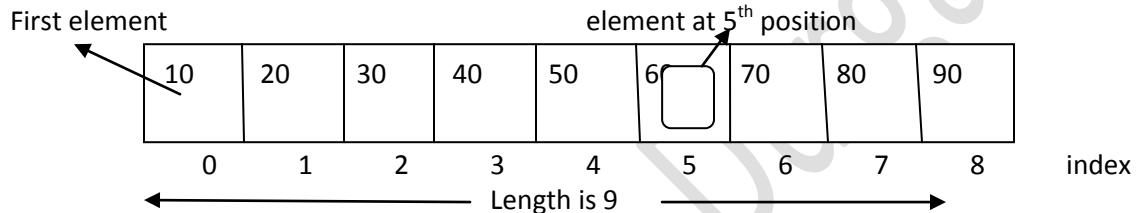
1. By using new operator.
2. by using clone() method.
3. without using new operator by using **String str="ratan"; [by using String content]**.
4. at the time of deserialization we are getting the data from file we are stored in object form.
5. By using factory method
  - a. Instance factory method
  - b. Static factory method
6. By using newInstance method. (used by servers).....etc

## 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;
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);
            }
        }
    }
}
```

**Root structure:-**

```
java.lang.Object
|
```

```
|--java.lang.reflect.Array
```

Array is a final class can't be extended.

**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**

**Test.java:-**

```
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:-**

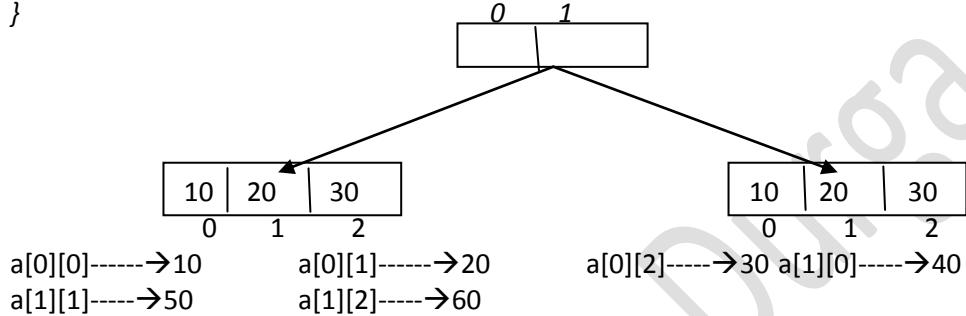
```
class Student
{
    int sid;
    String sname;
    Student(int sid,String sname)
    {
        //conversion of local to instance
        this.sid=sid;
        this.sname=sname;
    }
}
```

**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]);
    }
}
```

**Example :- fibonacci series**

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

```

        }
    }

Example :- febonacci series
import java.util.Scanner;
class Test
{
    public static void main(String[] args)
    {
        System.out.println("enter the no required for febonacci");
        int a = new Scanner(System.in).nextInt();

        System.out.println("enter first no of febonacci");
        int x = new Scanner(System.in).nextInt();
        System.out.println("enter second no of febonacci");
        int y = new Scanner(System.in).nextInt();

        int[] feb = new int[a];
        feb[0]=x;
        feb[1]=y;
        for (int i=2;i<a;i++)
        {
            feb[i]=feb[i-1]+feb[i-2];
        }
        //print the data
        for (int feb1 : feb)
        {
            System.out.print(" "+feb1);
        }
    }
}

```

**Pre-increment & post increment :-**

**Pre-increment**      :- it increases the value by 1 then it will execute statement.

**Post-increment**    :-it executes the statement then it will increase value by 1.

```

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

```

**Pre-decrement & postdecrement :-**

**Pre-decrement**      :- it decreases the value by 1 then it will execute statement.

**Post-decrement**    :-it executes the statement then it will increase value by 1.

class Test

```

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

```

7        5

### Java 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) Can we have multiple public classes in single source file?
- 5) Can we create multiple objects for single class?
- 6) What do you mean by token and literal?
- 7) What do you mean by identifier?
- 8) Is it possible to declare multiple public classes in single source file?
- 9) What is the difference between editor and IDE(integrated development environment)
- 10) Write the examples of editor and IDE?
- 11) Define a class?
- 12) In java program starts from which method and who is calling that method?
- 13) What are the commands required for compilation and execution?

- 14) Can we compile multiple source files at a time and is it possible to execute multiple .classes at a time?
- 15) The compiler understandable file format and JVM understandable file format?
- 16) What is the difference between JRE and JDK?
- 17) What is the difference between path and class path?
- 18) What is the purpose of environmental variables setup?
- 19) What do you mean by open source software?
- 20) What are operations done at compilation time and execution time?
- 21) What is the purpose of JVM?
- 22) JVM platform dependent or independent?
- 23) In java program execution starts from?
- 24) How many types of commands in java and what is the purpose of commands?
- 25) Is it possible to provide multiple spaces in between two tokens?
- 26) Class contains how many elements based on Ratan sir class notes?
- 27) Source file contains how many elements?
- 28) What are dependent languages and technologies in market on java?
- 29) Who is generating .class file and .class files generation is based on what?
- 30) What is .class file contains?
- 31) What is the purpose of data types and how many data types are present in java?
- 32) Who is assigning default values to variables?
- 33) What is the default value of int, char, Boolean, double?
- 34) Is null a keyword or not?
- 35) What do you mean by main class?
- 36) Is it possible to declare multiple classes with main method?
- 37) Can I have multiple main methods in single class?
- 38) What is the default package in java?
- 39) Can I import same class twice yes → what happened no → why?
- 40) Do I need to import java.lang package? yes → why no → why?
- 41) Is empty java source file is valid or not?
- 42) Is it java file contains more than one class?
- 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 and 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 do you mean by method signature?
- 56) What do you mean by method implementation?
- 57) How many types of methods in java and how many types of areas in java?
- 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 coding conventions of classes and interfaces?
- 67) What are the coding conventions of methods and variables?
- 68) What is the default package in java programming?
- 69) Platform dependent vs platform independent?
- 70) Is java a object oriented programming language?
- 71) By using which keyword we are creating object in java?
- 72) Object creation syntax contains how many parts?
- 73) How many types of constructors in java?
- 74) How one constructor is calling another constructor? One constructor is able to call how many constructors at time?
- 75) What do you mean by instantiation?
- 76) What is the difference between object instantiation and object initialization?
- 77) How many ways to create a object in java?
- 78) What is the purpose of this keyword?
- 79) Is it possible to use this keyword inside static area?
- 80) What is the need of converting local variables to instance variables?
- 81) Is it possible to convert instance variables to local variables yes → how no → why?
- 82) When we will get compilation error like “call to this must be first statement in constructor”?
- 83) When we will get compilation error line “cannot find symbol”?
- 84) What do u mean by operator overloading, is it java supporting operator overloading concept?
- 85) What is the purpose of scanner class and it is present in which package and introduced in which version?
- 86) What do you mean by constructor?
- 87) Who is generating default constructor and at what time?
- 88) What is the difference between named object and nameless object and write the syntax ?
- 89) What is object and what is relationship between class and Object?
- 90) Is it possible to execute default constructor and user defined constructor time?
- 91) If we are creating object by using new operator at that situation for every object creation how many constructors are executed?
- 92) What do you mean by object delegation?
- 93) What is the purpose of instance blocks when it will execute?
- 94) Inside class it is possible to declare how many instance blocks and what is syntax?
- 95) What is execution flow of method VS constructor Vs instance blocks Vs static blocks?
- 96) When instance blocks and static blocks are executed?
- 97) What are the new features of java1.5 version VS java1.6 VS java 1.7 VS java 8?

### **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) Switch is allowed String argument or not?
- 8) Inside the switch how many cases are possible and how many default declarations are possible?
- 9) What is difference between if & if-else & switch?
- 10) What is the default condition of for loop?
- 11) Inside for initialization & condition & increment/decrement parts optional or mandatory?
- 12) When we will get compilation error like “incompatible types”?
- 13) We are able to use break statements how many places and what are the places?
- 14) What is the difference between break& continue?
- 15) What do you mean by transfer statements and what are transfer statements present in java?
- 16) for (; ;) representing?
- 17) When we will get compilation error like “unreachable statement ”?
- 18) Is it possible to declare while without condition yes - →what is default condition no →what is error?
- 19) What is the difference between while and do-while?
- 20) 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) What is the purpose of extends keyword?
- 7) What do you mean by cyclic inheritance java supporting or not?
- 8) What is the difference between child class and parent class?
- 9) What is the root class for all java classes?

- 10) Inside the constructor if we are not providing this() and super() keyword the compiler generated which type of super keyword?
- 11) How to call super class constructors?
- 12) Is it possible to use both super and this keyword inside the method?
- 13) Is it possible to use both super and this keyword inside the constructor?
- 14) If the child class and parent class contains same variable name that situation how to call parent class variable in child class?
- 15) One class able to extends how many classes at a time?
- 16) If we are extending the your class will become parent class if we are not extending what is the parent class?
- 17) What do you mean by aggregation and what is the difference between aggregation and inheritance?
- 18) What do you mean by aggregation and composition and Association?
- 19) Aggregation is also known as?
- 20) How many objects are created ?
  - a. MyClassHero c1,c2;  
C1 = New MyClassHero();
- 21) What is the root class for all java classes?
- 22) Which approach is recommended to create object either parent class object or child class object?
- 23) Except one class all class contains parent class in java what is that except class?
- 24) What is the purpose of instance of keyword in java?
- 25) What do you mean by polymorphism?
- 26) What do you mean by method overloading and method overriding?
- 27) How many types of overloading in java?
- 28) Is it possible to override variable in java?
- 29) What do you mean by constructor overloading?
- 30) What are rules must fallow while performing method overriding?
- 31) When we will get compilation error like “overridden method is final”?
- 32) What is the purpose of final modifier java?
- 33) Is it possible to override static methods yes→how no→why?
- 34) Parent class reference variable is able to hold child class object?
- 35) How many types of polymorphism in java?
- 36) What do you mean by dynamic method dispatch?
- 37) The applicable modifiers for local variables?
- 38) Is it all methods present in final class is always final and is it all variables present final class is always final?
- 39) 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?
- 40) Object class contains how many methods?
- 41) When we will get compilation error like “con not inherit from final parent”?
- 42) How many types of type casting in java?
- 43) What do you mean by co-varient return types?
- 44) What do u mean by method hiding?
- 45) What do you mean by abstraction?
- 46) How many types of classes in java?
- 47) Normal class is also known as ?
- 48) What is the difference between normal method and abstract method?

- 49) What is the difference between normal class and abstract class?
- 50) Is it possible to create a object for abstract class?
- 51) What do you mean by abstract variable?
- 52) Is it possible to override non-abstract method as a abstract method?
- 53) Is it possible to declare main method inside the abstract class or not?
- 54) What is the purpose of abstract modifier in java?
- 55) How to prevent object creation in java?
- 56) What is the definition of abstract class?
- 57) In java is it abstract class reference variable is able to hold child class object?
- 58) What do you mean by encapsulation?
- 59) What do you mean by tightly encapsulated class?
- 60) What do you mean accessor method and mutator method ?
- 61) How many ways area there to set some values to class properties?
- 62) Can we overload method?
- 63) Can we inherit main method in child class?
- 64) In java main method is called by ?
- 65) The applicable modifiers on main method?
- 66) While declaring main method public static modifiers order mandatory or optional?
- 67) What is the argument of main method?
- 68) What is the return type of main method?
- 69) What are the mandatory modifiers for main method and optional modifiers of main method?
- 70) Why main method is static?
- 71) What do you by command line arguments?
- 72) Is it possible to pass command line arguments with space symbol no → good yes → how ?
- 73) What is the purpose of strictfp classes?
- 74) What is the purpose of strictfp modifier?
- 75) What is the purpose of native modifier?
- 76) What do you mean by native method and it also known as?
- 77) What do you mean by javaBean class?
- 78) The javabean class is also known as?
- 79) Applicable modifiers on local variables?
- 80) What is the execution process of constructors if two classes are there in inheritance relationship?
- 81) What is the execution process of instance blocks if two classes are there in inheritance relationship?
- 82) What is the execution process of static blocks if two classes are there in inheritance relationship?
- 83) What is the purpose of instanceof operator in java & what is the return-type?
- 84) If we are using instanceof both reference-variable & class-name must have some relationship otherwise compiler generated error message is what?

## **Packages**

1. What do you mean by package and what it contains?
2. What is the difference between user defined package and predefined package?
3. What are coding conventions must fallow while declaring user defined package names?
4. Is it possible to declare motile packages in single source file?

5. What do you mean by import?
6. What is the location of predefined packages in our system?
7. How many types of imports present in java explain it?
8. How to import individual class and all classes of packages and which one is recommended?
9. What do you mean by static import?
10. What is the difference between normal and static import?
11. Is it possible to import multiple packages in single source file?
12. Is it possible to declare multiple packages in single source file?
13. I am importing two packages, both packages contains one class with same name at that situation how to create object of two package classes?
14. If we are importing root package at that situation is it possible to use sub package classes in our applications?
15. What is difference between main package and sub package?
16. If source file contains package statement then by using which command we are compiling that source file?
17. What do you mean by fully qualified name of class?
18. What is the public modifier?
19. What is the default modifier in java?
20. What is the public access and default access?
21. What is private access and protected access?
22. What is the difference between public methods and default method?
23. What is the difference between private method and protected method?
24. What is most restricted modifier in java?
25. What is most accessible modifier in java?

### **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. Checked exceptions are caused by?
6. Unchecked exceptions are caused by?
7. Errors are caused by?
8. Is it possible to handle Errors in java?
9. What the difference is between partially checked and fully checked Exception?
10. What do you mean by exception handling?
11. How many ways are there to handle the exception?

12. What is the root class of Exception handling?
13. Can you please write some of checked and un-checked exceptions in java?
14. What are the keywords present in Exception handling?
15. What is the purpose of try block?
16. In java is it possible to write try with out catch or not?
17. What is the purpose catch block?
18. What is the difference between try-catch?
19. Is it possible to write normal code in between try-catch blocks?
20. What are the methods used to print exception messages?
21. What is the purpose of printStackTrace( ) method?
22. What is the difference between printStackTrace( ) & getMessage()?
23. What is the purpose of finally block?
24. If the exception raised in catch block what happened?
25. Independent try blocks are allowed or not allowed?
26. Once the control is out of try , is it remaining statements of try block is executed?
27. Try-catch , try-catch-catch , catch-catch , catch-try how many combinations are valid?
28. Try-catch-finally , try-finally ,catch-finally , catch-catch-finally how many combinations are valid?
29. Is possible to write code in between try-catch-finally blocks?
30. Is it possible to write independent catch blocks?
31. Is it possible to write independent finally block?
32. What is the difference between try-catch –finally?
33. What is the execution flow of try-catch?
34. If the exception raised in finally block what happened?
35. What are the situations finally block is executed?
36. What are the situations finally block is not executed?
37. What is the purpose of throws keyword?
38. What is the difference between try-catch blocks and throws keyword?
39. What do you mean by default exception handler and what is the purpose of default exception handler?
40. How to delegate responsibility of exception handling calling method to caller method?
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, databaseclosing 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. When we will get compilation error like “unreportedException must be catch”?
56. When we will get compilation error like “Exception XXXException has already been caught”?
57. When we will get compilation error like “try without catch or finally”?
58. How many approaches are there to create user defined unchecked exceptions and un-checked exceptions?
59. What do you mean by exception re-throwing?
60. How to create object of user defined exceptions?
61. How to handover user created exception objects to JVM?
62. What is the difference user defined checked and unchecked Exceptions?
63. Is it possible to handle different exceptions by using single catch block yes-->how no→why?

### **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?

1. 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 following 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. Tell me some of the marker interfaces?
- bb. What are the advantages of marker interfaces?
- cc. Is it possible to create user defined marker interfaces /
- dd. What do you mean nested interface?

#### 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) Factory method
- 14) Template method
- 15) Default method
- 16) Public method
- 17) Private method
- 18) Protected method
- 19) Final method
- 20) Strictfp method
- 21) Native method

Different types of classes in java (must know information about all classes)

- 1) Normal class /concrete class /component class
- 2) Abstract 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 ?

### Collections

- 1) What is the main objective of collections?
- 2) What are the advantages of collections over arrays?
- 3) Collection frame work classes are present in which package?
- 4) What is the root interface of collections?
- 5) List out implementation classes of List interface?
- 6) List out implementation classes of set interface?
- 7) List out implementation classes of map interface?
- 8) What is the difference between heterogeneous and homogeneous data?
- 9) What do you mean by legacy class can you please tell me some of the legacy classes present in collection framework?
- 10) What are the characteristics of collection classe?
- 11) What is the purpose of generic version of collection classes?
- 12) What is the difference between general version of ArrayList and generic version of ArrayList?
- 13) What is purpose of generic version of ArrayList & arrays?
- 14) How to get Array by using ArrayList?
- 15) What is the difference betweenArrayList and LinkedList?
- 16) How to decide when to use ArrayList and when to use LinkedList?
- 17) What is the difference between ArrayList & vector?
- 18) How can ArrayList be synchronized without using vector?
- 19) Arrays are already used to hold homogeneous data but what is the purpose of generic version of Collection classes?
- 20) What is the purpose of RandomAccess interface and it is marker interface or not?
- 21) What do you mean by cursor and how many cursors present in java?
- 22) How many ways are there to retrieve objects from collections classes what are those?
- 23) What is the purpose of Enumeration cursor and how to get that cursor object?
- 24) By using how many cursors we are able to retrieve the objects both forward backward direction and what are the cursors?
- 25) What is the purpose of Iterator and how to get Iterator Object?
- 26) What is the purpose of ListIterator and how to get that object?
- 27) What is the difference between Enumeration vs Iterator Vs ListIterator?

- 28) We are able to retrieve objects from collection classes by using cursors and for-each loop what is the difference?
- 29) All collection classes are commonly implemented some interfaces what are those interfaces?
- 30) What is the difference between HashSet & linkedHashSet?
- 31) all most all collection classes are allowed heterogeneous data but some collection classes are not allowed can you please list out the classes?
- 32) What is the purpose of TreeSet class?
- 33) What is the difference between Set & List interface?
- 34) What is the purpose of Map interface?
- 35) What do you mean by entry.
- 36) What is the difference between HashMap & LinkedHashMap?
- 37) What is the difference between comparable vs Comparator interface?
- 38) What is the difference between TreeSet and TreeMap?
- 39) What is the difference between HashTable and Properties file key=value pairs?
- 40) What do you mean by properties file and what are the advantages of properties file?
- 41) Properties class present in which package?
- 42) What is the difference between collection & collections?

### 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?

### Enumeration

- 1) What is the purpose Enumeration?
- 2) How to declare enum?
- 3) enum constants are by default?
- 4) One enum is able extends other enum or not?

- 5) For the enum compiler generate .class files or not?
- 6) What is the difference enum & Enum?
- 7) Is it possible to declare main method & constructor inside the enum or not?
- 8) Is it possible to provide parameterized constructor inside the enum?
- 9) What is the difference between enum and class?
- 10) What is the purpose of values() methods?
- 11) What is the purpose of ordinal() method?
- 12) Is it possible to create object for enum?
- 13) For enum inheritance concept is applicable or not?
- 14) Is it possible to create object of enum?
- 15) When enum constants are loaded?
- 16) Enums are able to implement interfaces or not?
- 17) Enum introduced in which version?
- 18) What is the difference between **enum& Enumeration & Enum**?
- 19) Is it possible to override toString() method inside enum?
- 20) Can you use enum constants switch case in java?

#### 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) The inner class is able to access outer class private properties or not?
- 6) The outer class is able to access inner classes properties& methods or not?
- 7) How to create object inner class and outer class?
  - a. Class Outer
 

```
{     class Inner{  } }
```
- 8) For the inner classes compiler generates .class files or not? If generates write the name of above inner class .class file name ?
- 9) The outer class object is able to call inner class properties & methods or not?
- 10) The inner class object is able to call outer class properties and methods or not?
- 11) What is the difference between normal inner classes and static inner classes?
- 12) What do you mean by anonymous inner classes?
- 13) What do you mean by method local inner classes?
- 14) Is it possible to create inner class object without outer class object?
- 15) Java supports inner method concept or not ?
- 16) Is it possible to declare main method inside inner classes?
- 17) Is it possible to declare constructors inside inner classes?
- 18) If outer class variables and inner class variables are having same name then hoe to represent outer class variables and how to represent inner class variables?
- 19) Is it possible to declare same method in both inner class and outer class?
- 20) Is it possible to declare main method inside outer 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.
27. IOException is checked Exception or unchecked Exception?

### Multhreading

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 fallows which algorithm?
16. What is purpose of thread priority?
17. What is purpose of sleep() & isAlive() & isDemon() & 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() motify() 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?

# Thank you

by Mr. Ratan.