

Last modification date :- 29<sup>th</sup> oct 2014 [new version don't prefer old copy]



# Core java

**by**

**by**

# Mr. Ratan

# Mr. Ratan

DurgaSoftwareSolutions

DurgaSoftwareSolutions

*Java Home*

***Java Home***

**Index Page**

<b>1) Introduction</b>	<b>1-54</b>
a. Variables	32
b. Methods	39
c. Constructors	55
<b>2) Oops</b>	<b>72 - 124</b>
a. Inheritance	
b. Polymorphism	95
c. Garbage Collector	109
d. Abstraction	113
e. Main method	120
f. Encapsulation	
<b>3) Packages</b>	<b>125-134</b>
<b>4) Interfaces</b>	<b>135-143</b>
<b>5) String manipulations</b>	<b>144-157</b>
<b>6) Wrapper classes</b>	<b>158-162</b>
<b>7) Java.io</b>	<b>163-170</b>
<b>8) Exception handling</b>	<b>171-194</b>
<b>9) Multi Threading</b>	<b>195 – 207</b>
<b>10) Nested classes</b>	<b>208 – 218</b>
<b>11) Enumeration</b>	<b>219- 221</b>
<b>12) Collections &amp; generics</b>	<b>222 – 244</b>
<b>13) Networking</b>	<b>245 – 250</b>
<b>14) Java.awt</b>	<b>251-283</b>
<b>15) Swings</b>	<b>284-293</b>
<b>16) Arrays</b>	<b>294-296</b>
<b>17) Java interview questions</b>	<b>297-308</b>

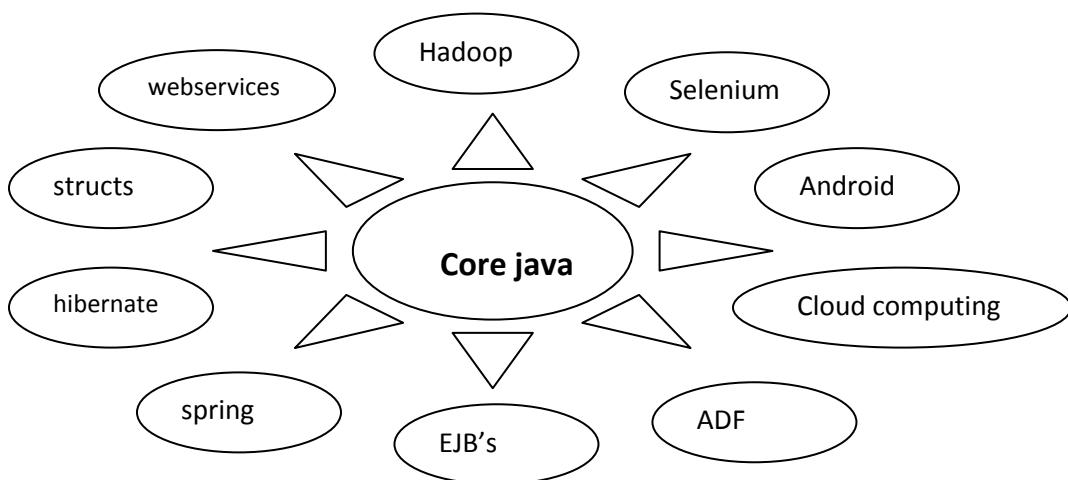
**JAVA introduction:-**

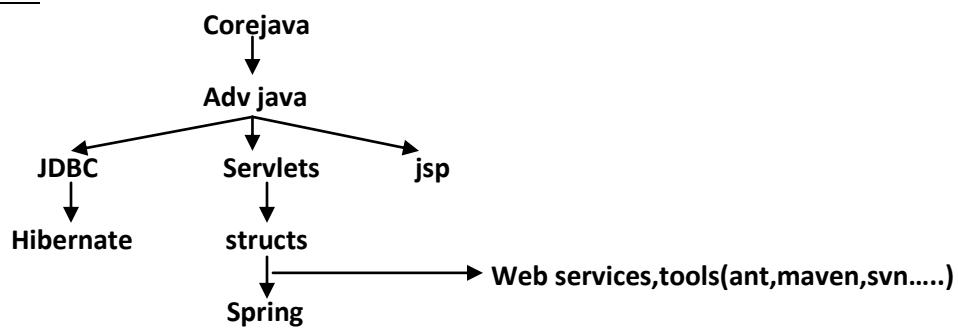
Author	:	<b>James Gosling</b>
Vendor	:	Sun Micro System(which has since merged into Oracle Corporation)
Project name	:	Green Project
Type	:	open source & free software
Initial Name	:	OAK language
Present Name	:	java
Extensions	:	.java & .class & .jar
Initial version	:	jdk 1.0 (java development kit)
Present version	:	J2SE 8 2014
Operating System	:	multi Operating System
Implementation Lang	:	c, cpp.....
Symbol	:	coffee cup with saucer
Objective	:	To develop web applications
SUN	:	<b>Stanford Universally Network</b>
Slogan/Motto	:	WORA(write once run anywhere)

**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 durgajobs.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:-Java keywords:-

<u>Data Types</u>	<u>flow-control</u>	<u>Exception-handling</u>	<u>object-level</u>	<u>modifiers</u>
byte	if	try	new	public
short	else	catch	this	private
int	switch	finally	super	protected
long	case	throw	instanceof	final
float	default	throws	<b>(4)</b>	static
double	break	(5)		<b>1.5 version</b>
char	while	<u>source-file</u>		abstract
Boolean	do	class		synchronized
<b>(8)</b>	continue	extends	enum	volatile
<u>Unused</u>	<b>(10)</b>	interface	<b>(2)</b>	native
goto		implements	<u>method-level</u>	<b>(11)</b>
const		package	void	
<b>(2)</b>		import	return	
		(6)	<b>(2)</b>	

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

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)

**Differences:-**

C-lang	Cpp-lang	Java -lang
1) Author: Dennis Ritchie	1)Author : Bjarne Stroustrup	1) Author : James Gosling
2) Implementation languages: BCPL, B...	2) implementation languages are c ,ada,ALGOL68.....	2) implementation languages are C,CPP,ObjectiveC.....
3) In C lang program execution starts from main method called by <b>Operating system.</b>	3) program execution starts from main method called by <b>operating system.</b>	3) program execution starts from main method called by <b>JVM(java virtual machine).</b>
4) In c-lang the predefined support is available in the form of header files <b>Ex:- Stdio.h , Conio.h</b>	4) cpp language the predefined is maintained in the form of header files. <b>Ex:- iostream.h</b>	4) In java predefined support available in the form of packages. <b>Ex: - java.lang, java.io</b>
5) The header files contain predefined functions. <b>Ex:- printf,scanf.....</b>	5) The header files contains predefined functions. <b>Ex:- cout,cin....</b>	5) The packages contains predefined classes and class contains predefined funtions. <b>Ex:- String,System</b>
6) To make available predefined support into our applications use #include statement. <b>Ex:- #include&lt;stdio.h&gt;</b>	6) To make available predefined support into our application use #include statement. <b>Ex:- #include&lt;iostream&gt;</b>	6) To make available predefined support into our application use import statement. <b>Ex:- import java.lang.*;</b> [*] mean all
7) size of the data types are varied from operating system to the operating system. 16-bit int -2bytes char- 1byte 32-bit int -4bytes char – 2 bytes	7) Irrespective of any os Intsize -4 Char size -2	7) Irrespective of any os Int -4 Char-2
8) To print some statements into output console use “printf” function. <b>Printf(“hi ratan ”);</b>	8) To print the statements use “cout” function. <b>Cout&lt;&lt;”hi ratan”;</b>	8)To print the statements we have to use <b>System.out.println(“hi ratan”);</b>
9)extensions used :- <b>.c , .obj</b> <b>, .h</b>	9) extensions used :- <b>.cpp ,.h</b>	9)extensions used : - <b>.java , .class</b>

**C –sample application:-**

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

**CPP –sample application:-**

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

**Java sample application:-**

```
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:-**

java-language  
↓  
packages  
↓  
classes & interfaces  
↓  
methods & variables

james gosling  
↓  
java.lang  
↓  
System,String.....  
↓  
length(),charAt(),concat()...

**public static void main(String[] args)**  
(execution startsfrom 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.

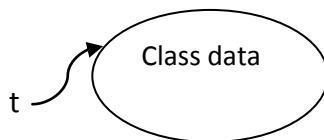
**2. Object Oriented:-**

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
{
    Class data;
}
```

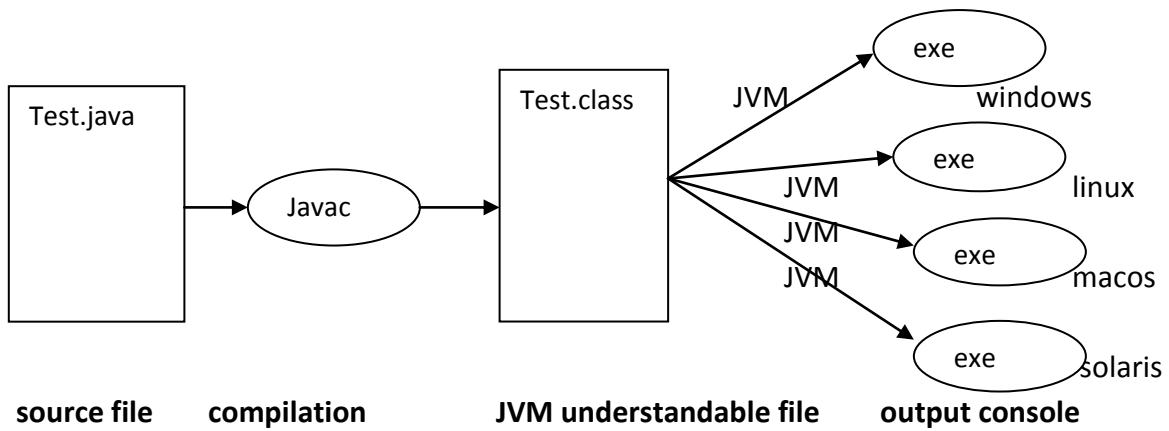
Test t=new Test();



The total java language is dependent on object only hence we can say java is a object oriented technology.

**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 allows 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

- a. JAVA is having very good predefined Exception Handling mechanism whenever we are getting exception we are having meaning full information.
- b. 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`.

Web security for web applications we are having JAAS(Java Authentication and Authorization Services) for distributed applications.

**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 Interpretator we are converting source code into byte code and the interpretator 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 :-**

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 fallowing 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 fallowing 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 a Program.
- 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.**

**IDE:- ( 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 .
- 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 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

1. 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.
2. if the source file does not contain public class then save the source file with any name (**anyName.java**) like A.java , Rtan.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
{
}
```

**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.**

##### **Syntax:-**

```
Javac filename
Javac Test.java
```

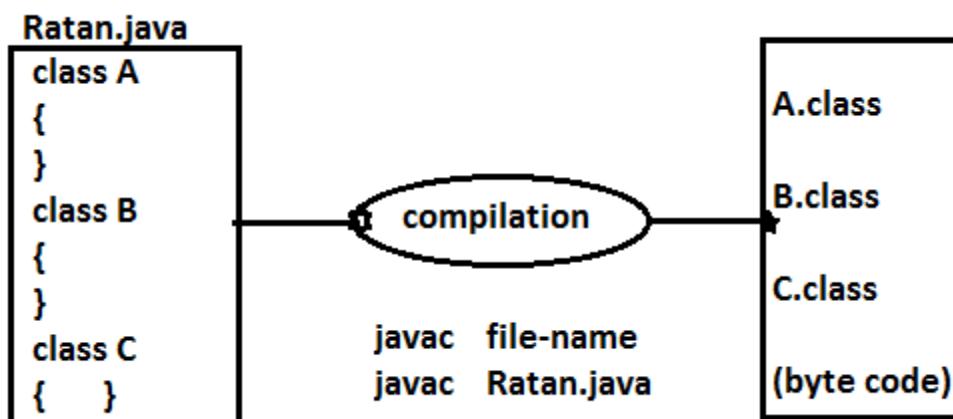
Whenever we are doing compilation compiler perform following actions.

- Compiler checks the syntax error,
- a. if syntax error are there compiler generate error message.
  - b. If syntax errors are not there compiler generate **.class files**

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

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

<b><i>To compile single source file</i></b>	<b><i>----&gt; javac file-name</i></b>
<b><i>To compile multiple source files at a time</i></b>	<b><i>-----&gt; Javac *.java (* represent all files)</i></b>



#### Step-5:- Execution process.

*Java class-name*

*Java Test*

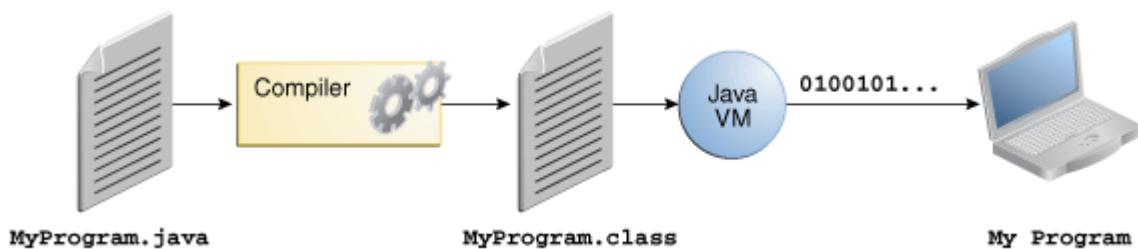
Whenever you are executing particular class file then JVM perform fallowing actions.

- 1) 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**".
- 2) 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**".

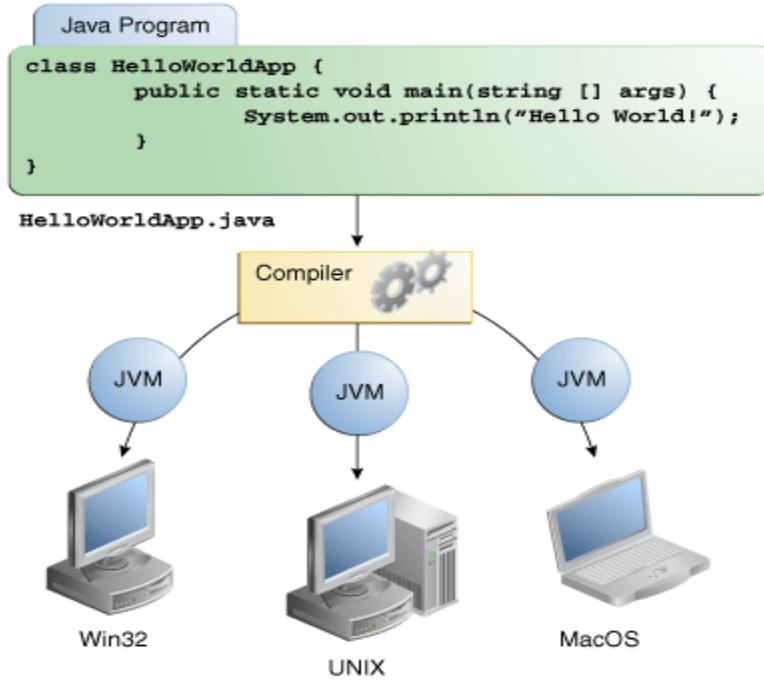
*Note 1:- compiler is translator it is translating .java file to .class where as JVM is also a translator it is translating .class file to machine code.*

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

Environment of the java programming development:-



First program development :-

**Example :-**

```

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 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**

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

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

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

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

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

**Java Identifiers:-**

any name in the java program like variable name, class name, method name, 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:-**

- 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

- The identifier will not contains symbols like

+ , - , . , @ , # , \* .....

- The identifier should not duplicated.

```
class Test
```

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

```
}
```

the identifier should not be duplicated.

- 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;           //predefine String class
          int Serializable=20;    //predifined Serializable Interface
          float Exception=10.2f;   //predefined Exception class
          System.out.println(String);
          System.out.println(Serializable);
          System.out.println(Exception);
    }
}
```

**JAVA NAMING CONVENTIONS:-**

Java is a case sensitive language so the way of writing code is important.

- All Java classes, Abstract classes and Interface names should start with uppercase letter ,if any class contain more than one word every inner word also start with capital letters.

Ex: **String, StringBuffer, FileInputStream**

- All java methods should start with lower case letters and if the method contains more than one word every innerword should start with capital letters.

Ex :- **post(), toString(), toUpperCase()**

- All java variables should start with lowercase letter and inner words start with uppercase letter.

Ex:- **pageContent bodyContent**

- All java constant variables should be in uppercase letter.

Ex: **MIN\_PRIORITY MAX\_PRIORITY NORM\_PRIORITY**

5. All java packages should start with lower case letters only.

Ex:      java.awt, Java.io, java.awt, java.util.....etc

**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 Comments :-**

- Comments are used to provide detailed description about application and these 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
```

**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 Durga
{
    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");
        }
    }
};
```

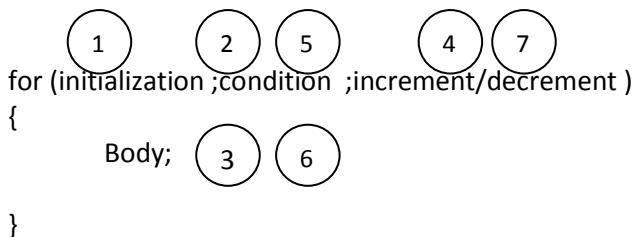
**Iteration Statements:-**

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:-****Ex 1: Inside the for loop initialization part is optional.**

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

**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("durgasoft");//unreachable statement
    }
}
```

**Example :-**

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

**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");
        System.out.println("durga");
        break;
        System.out.println("aruna");
        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. Representing Type of the variables and expressions it means the variable is able to which type of value.
2. Representing how much memory is allocated for variable.
3. Specifies range value of the variable.

There are 8 primitive data types in java

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

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

Ex:-    *int a=10;*

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

**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

```

**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
```

**Default values(JVM assigned values)**

```

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

```

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

```

class Test
{
    public static void main(String[] args)
    {
        byte a=10;           System.out.println(a);
        short s=20;          System.out.println(s);
        int i=30;            System.out.println(i);
        long l=40 ;          System.out.println(l);
        float f =10.5f;       System.out.println(f);
        double d=20.5;        System.out.println(d);
        char ch='a';          System.out.println(ch);
        boolean b= true;      System.out.println(b);
    }
}

```

**Representing floating values:-****case 1:- invalid**

```
float f=10.5 ;
System.out.println(f);
compilation error:- because by default point values are (10.5) values are double.
```

D:\morn11>javac Test.java

**Test.java:10: possible loss of precision**

```
    float f=10.5; System.out.println(f);
    required: float
    found: double
```

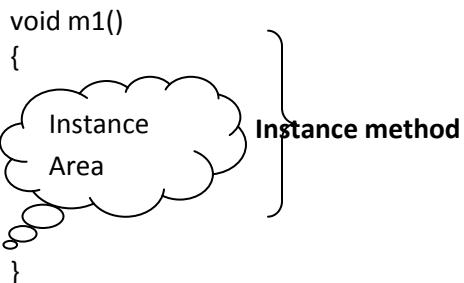
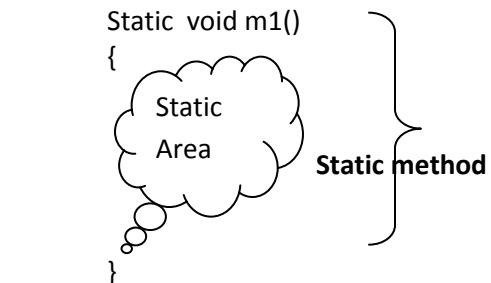
**case 2:- to overcome above problem use "f" constant to represent float value.**

```
float f=10.5f; //valid
System.out.println(f);
byte+byte=int      byte+short=int      short+int=int      byte+int=int      byte+long=long
short+float=float   long+float=float   float+double=double   String+int=String
```

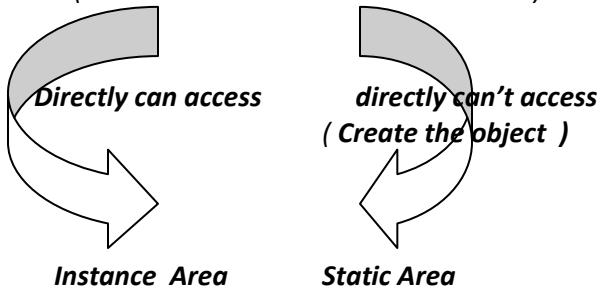
**Areas of java language:-**

There are two types areas in java.

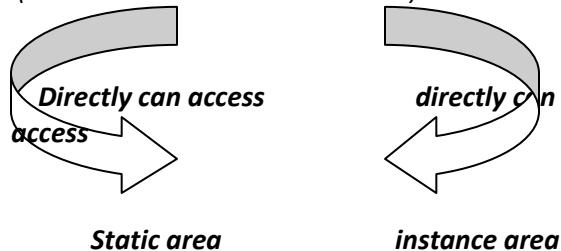
- 1) Instance Area.
- 2) Static Area.

**Instance Area:-****Static Area:-****Instance variables & methods declaration:-**

(Instance variables& instance methods)

**Static variables & methods accessing:-**

(Static variables& static methods)



**Two figures are heart of the java language**

**Java Variables:-**

- Variables are used to hold the constant values by using these values we are achieving project requirements/functionality.
- While declaring variable we must specify the type of the variable by using data types.

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.
- ❖ We are able to use local variable only inside the method or constructor or blocks only it is not possible to access outside of method or constructor or block.
- ❖ Access (calling) the local variables directly.
- ❖ For the local variables memory allocated when method starts and memory released when method completes.
- ❖ Local variables are stored in stack memory.

**Example :-**

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

**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.
- ❖ Access(calling) the instance variables by using object name.
- ❖ For instance variables memory allocated during object creation time and memory released when object destroyed.
- ❖ Instance variables are stored in heap memory.

```
class Test
{
    //instance variables
    int a=10;      int b=20;
    //static method
    public static void main(String[] args)      //program execution starting point called by JVM
    {
        //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.
- ❖ Call the static members by using class Name.
- ❖ 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.

**Example :-**

```

class Test
{
    //static variables
    static int a=1000;
    static int b=2000;
    //static method
    public static void main(String[] args) // program execution starts from main called by JVM
    {
        System.out.println(a);
        System.out.println(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(a);
    System.out.println(b);
}
};

```

**Calling of static variables:-**

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

- a. Directly possible.
- b. By using class name.
- c. By using reference variable.

```

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

```

```

        Test t=new Test();
        System.out.println(t.a);      //3-way(By using reference variable)
    }
};


```

**Variables VS default values:-**

Case 1:- for the instance variables the JVM will set the default values.

```

class Test
{
    //for the instance variables JVM will assign default values.
    int a;
    boolean b;
    public static void main(String[] args)
    {
        //access the instance members by using object name(reference variable)
        Test t=new Test();           //creates object of Test class
        System.out.println(t.a);     //instance variable calling
        System.out.println(t.b);     //instance variable calling
    }
};

```

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

```

class Test
{
    //for the static variables JVM will assign default values.
    static int a;
    static float b;
    public static void main(String[] args)
    {
        //access the static members by using class Names
        System.out.println(Test.a);   //static variables calling
        System.out.println(Test.b);   //static variables calling
    }
};

```

Case 3:- for the local variables the JVM won't provide the default values before using those variables must initialize 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,b;      //for the local variables JVM does not provide default values
        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);

**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.
- ❖ Instance variables memory is allocated on object basis it means if we are creating 10 objects for every object one copy of instance variable created .
- ❖ Static variables for all objects same copy is maintained. One Object change the value next created objects are affected.
- ❖ Static variables memory is allocated on class based it means for single class one copy is created and all objects of that class uses same copy.

**Class Vs Object:-**

- Class is a logical entity it contains logics where as object is physical entity it is representing memory.
- Class is used to declare logics for that logics memory is allocated by creation of Object.
- Class is blur 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 required memory.
- Civil engineer based on blue print of house it is possible to create multiple houses but every house required place.
- We will discuss object creation in detailed in constructor concept.

**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.a=333;      t1.b=444;
          Test t2 = new Test();//10 444
          System.out.println(t2.a);//10
          System.out.println(t2.b);//444
          t2.b=555;
          Test t3 = new Test();//10 555
          System.out.println(t3.a);//10
          System.out.println(t3.b);//555
      }
}

```

**Example :- ( variables summary)**

```

class Test
{
    int x=100;          //instance variable(use this variable inside the class in multiple methods)
    static int y=1000; // static variable (use this variable inside the class in multiple methods)
    //instance method
    void m1() //user defined method must called by user in main method
    {
        boolean b=true;           //local variable declaration
        System.out.println(b);   //printing local variable
        System.out.println(x);   //printing instance variable
        System.out.println(Test.y); //printing static variable
    }
    //static method
    static void m2() //user defined method must called by user in main method
    {
        double d =10.5;           //local variable
        System.out.println(d);   //printing local variable
        Test t = new Test();
        System.out.println(t.x);   //printing instance variable
        System.out.println(Test.y); //printing static variable
    }
    //static method
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();                  //instance method calling
        Test.m2();                //static method calling
    }
}

```

**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 or constructor starts and memory id destroyed when method ends and constructor 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 at object creation time and these variables are stored in heap memory & call the instance members (variables & methods) by using object name(reference variable).

**Note 3 :-** The variables which are declared inside the class and outside of the method with static modifier those variables are called static variables and we are able to access (usage) instance members inside the class , for the static variables memory is allocated at .class loading time and these variables are stored in non-heap memory & call the static members(variables & methods) by using class Name.

**Characteristics of variables:-**

<b>Characteristic</b>	<b>Local variable</b>	<b>instance variable</b>	<b>static variables</b>
<b>where declared</b>	inside method or constructor or block.	inside the class outside Of methods	inside the class outside of methods.
<b>Use</b>	within the method	inside the class more than one method and constructor.	inside the class more than one method and constructor.
<b>Lifetime</b>	created method or constructor entered, destroyed on exit.	created when object of class created with new & destroyed when object is destroyed.	created when the program starts and destroyed when the Program ends.
<b>Initial value</b>	none, must initialize the value before first use.	default values are assigned based on data -types and objects.	default values are assigned based on Data-types & objects.
<b>Object creation</b>	no way related to object	for every object one copy Of instance variable created It means memory.	for all objects one copy is created. Single memory.
<b>Accessing</b>	directly possible.	By using object name. <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>Memory</b>	stored in stack memory.	Stored in heap memory	non-heap memory.

**+ operator(for String concatenation):-**

'+' is a overloaded operator in the java language the + is acting as a addition of the two numbers and the '+' is acting as a concatenation of the two strings.

- 1) if both operands are numbers then "+" perform addition .
- 2) if at least one operand is String then "+" perform concatenation .

```
class Test
```

```
{           public static void main(String[] args)
{           int a=10;           int b=20;
String str1="durga";
String str2=40+50+str1+"ratan"+60+70;
System.out.println(str2);
System.out.println(a+"----"+b);
System.out.println(str1+"----"+a+"----"+b);
}
}
```

**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):-**

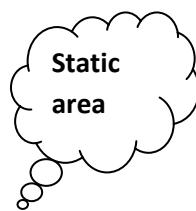
- 1) Methods are used to write the business logic of the project that logics will be executed whenever we are calling that method.
- 2) Coding conversion of method is method name starts with lower case letter if method contains more than one word then every inner word starts with uppercase letter.  
Ex:- post() , charAt() , toUpperCase() , compareToIgnoreCase().....etc
- 3) There are two types of methods
  - a. Instance method
  - b. Static method

**Instance method:-**

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

Instance method

**Static Method:-**

```
Static Void m1()
{
    
}
}
```

Static method

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

**Note:** - for the instance methods memory is allocated at object creation time but for the static methods memory is allocated at .class file loading.

**Note:** - access (call) the instance members(variables & methods) by using object name and call the static members(variables & methods) by using class name.

**Syntax:-**

```
Void m1() { } //instance method
Static void m2() { }//static method
Test obj = new Test();
Classname.staticmethod(); // call static method by using class name
Test.m2();
Objectname.nonstatic method(); or Objectname.instancemethod(); //calling instance method
Obj.m1();
```

**Syntax:-**

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

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

Ex:-   **Public void m1()**  
**Private int m2(int a,int b)**  
**public boolean createNewFile() throws java.io.IOException**  
**public abstract void commit() throws java.sql.SQLException;**  
**public abstract void rollback() throws java.sql.SQLException;**

**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.

- a. **Method declaration**
- b. **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 durga( ) { System.out.println("durga"); }
    void soft( ) { System.out.println("software solutions"); }
    public static void main(String[] args)
    {
        Test t=new Test();
        t.durga(); //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 is executed.
- Parameters of methods is nothing but input 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-5 :-while calling methods it is possible to provide input to method in two ways.**

1) By passing constant values.                      *Obj.m1(10,20)*

2) By passing variables as a argument values.    *Obj.m1(a,b);*

*import java.util.\*;*                      //to import Scanner class

```

class Test
{
    //static method
    static void status(int age)      //local variable
    {        if (age>18) {        System.out.println("eligible for voting"); }
            else {        System.out.println("not eligible for voting"); }
    }
    public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in); //used to take dynamic input from keyboard
        System.out.println("enter u r age");
        int age = s.nextInt();           //used to get the value from command prompt
        Test.status(age);             //static method calling by passing variable as a argument
    }
}

```

**Example-6 :- 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-7:-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-8:-** 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-9 :-**

- 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.

```

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

```

**Example-10 :-**

- 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-11 :- methods vs return type.**

1. Every functionality is able to return some value like 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.....).
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 must return the 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 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");
}
```

5. Every method is able to return value and holding 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-12:- methods vs. return variables****Returns local variable as a return value**

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

**Returns instance variable as a return value**

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

**If both local & instance variables having same name then to return instance value use this keyword.**

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

**Example 13 :- 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 14:- The java class is able to return user defined class as a return value.**

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

class Heroin
{
    void age(){System.out.println("Anushka age is:30");}
};

class Test
{
    //instance method return class as a return type
    Person m1() //m1() return type is class hence return Person object
    {
        Person p = new Person();
        return p;
    }
    Heroin m2() //m2() return type is Heroin class hence return Heroin object
    {
        Heroin h = new Heroin();
        return h;
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        Person p = t.m1();      p.eat();
        Heroin h = t.m2();     h.age();
    }
}
```

**Example 15:**

This keyword representing current class objects.

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

- 1) By creating object and return reference variable.
- 2) **This** keyword representing current class object hence by using this keyword we are to return current class object.

```
class Test
{
    Test m1()      //first approach to return same class(Test) object
    {
        Test t = new Test();
        return t;
    }
    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:- 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);
    }
};
```

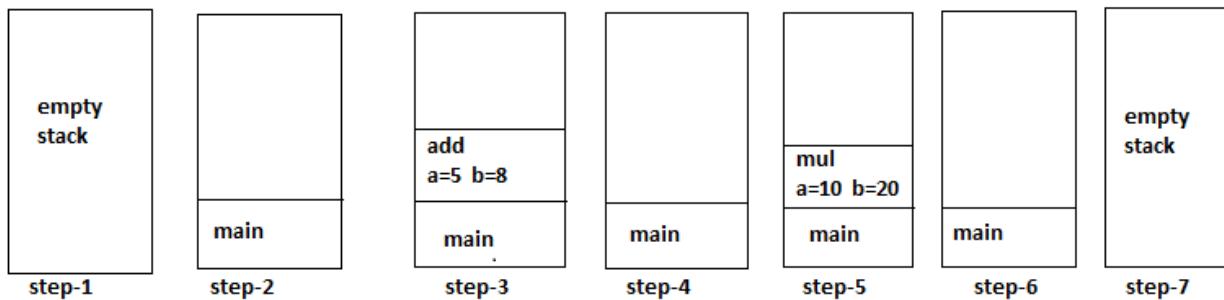
**Example 2:- (StackOverFlowError)**

```
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);
    }
};
```

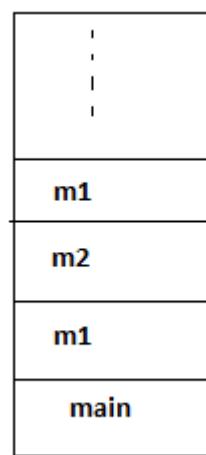
**Example 17 :- Stack Mechanism:-**

- 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("durga");
        m1();
    }
    public static void main(String[] args)
    {
        Test t=new Test();      t.m1();
    }
}
```



**Example 19:- For java methods it is possible to provide Objects as a parameters.**

```
class A
{
}
class B
{
}
class Test
{
    void testMethod(A a,B b) //method is taking objects as a parameters list.
    {
        System.out.println(a.getClass());
        System.out.println(b.getClass());
    }
    public static void main(String[] args)
    {
        A a = new A();
        B b = new B();
        Test t = new Test();
        t.testMethod(a,b); //calling method by passing two objects
    }
}
```

**this keyword:-**

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

- Current class variables.
- Current class methods.
- 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 instancevariables:-****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
}//end class

```

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();
    }
};
```

### Class vs Object:-

- Class is a group of objects that have common property. Java is classes based language we are able to design the program by using classes and objects.
- Object is a real world entity. Object orientation is methodology to design a program by using classes and objects.
- Object is physical entity where as class is a logical entity.
- Object creation
  - Test t=new Test(); (Test object creation)
  - Ratan r=new Ratan(10,20); (Ratan object creation)
  - Employee e=new Employee("ratan",111,10000); (Employee Object creation)
- A class provides the blueprint for Objects.
- Object is nothing but instance of a class.

### Every objects contains 3 characteristics

1. State(represent data of an object)
2. Behavior(represent behavior of an object)
3. Identity(used to identify the objects uniquely).

### PEN(object):-

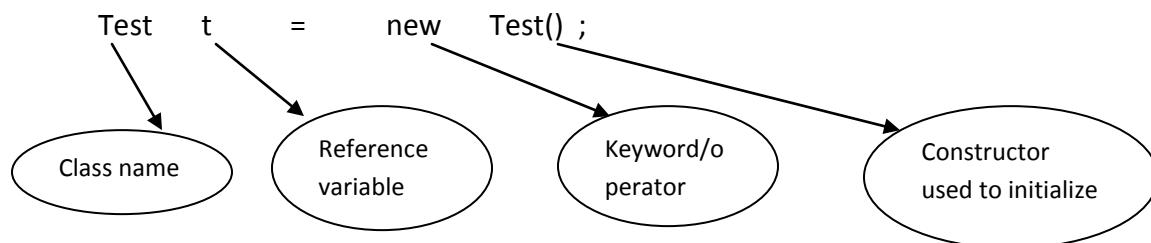
State:- name raynolds,color red etc.....

Behavior:- used to write

**Declaration** :- it is representing the variable associated with object.

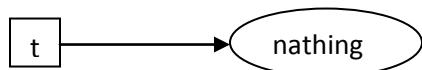
**Instantiation** :- The new keyword is a Java operator that creates the object.

**Initialization** :- The new operator is followed by a call to a constructor, which initializes the new object.



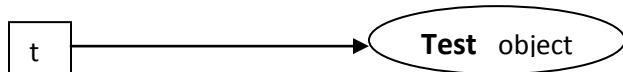
### Test t;

- a. It notifies the compiler refers the Test data with t reference variable.
- b. Declaring reference variable doesn't mean creating object we must use new operator to create object. **t is reference variable pointing to nothing.**



### Test t=new Test();

- a. The new operator instantiating class by allocating memory for new object and the reference variable pointing to that object.



**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

**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) Compiler generates Default constructor inside the class when we are not providing any type of constructor (0-arg or parameterized).
- 2) Inside the class if we declaring at least one constructor (0-arg or parameterized) the compiler won't generate default constructor.
- 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 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.

**Exampl-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.**

**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,"durga",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-3:-**

- if we are trying to compile below application the compiler will generate error message "can not 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-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);
}
}

```

**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 contructor 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 costructor 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 contains 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 :- 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
{
    int sid;
    String sname;
    int smarks;
    Student(int sid,String sname,int smarks)
    //conversion
        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);
    //constructor is taking Student object
    Student s1 = new Student(s);
    s.disp();
    s1.disp();
}
```

**Difference between methods and constructors:-****Methods**

1. Methods are used to write the functionality that functionality is executed whenever we are calling that method.
2. Method name starts with lower case letter , if the method contains more than one word that inner word starts with uppercase letter.
3. For the methods return type is mandatory.
4. There are two types of methods
  - a. Instance method
  - b. Static method
5. Inside the class it is possible to write multiple instance methods and static methods.
6. It is possible to call methods directly without using **this** keyword.
7. One method is able to call multiple methods at a time.
8. Method calling is any statement inside the method.
9. Parameters of methods are local variables.

Void m1(int a,int c) { } // local variables

10. To call the super class methods user **super** keyword.

**constructors**

- 1) Constructors are used to write the functionality and assign the values to variables that functionalities are executed t object creation time automatically.
  - 2) Constructor name and class name must be same.
  - 3) Constructor doesn't contains return types(return type is not allowed).
  - 4) There are two types of constructors
    - a. Default constructor.
    - b. Parametarized constructor.
  - 5) Inside the constructor if we are declaring constructor compiler not generate default constructor but if we are not declaring constructor compiler generate default constructor.
  - 6) To call the current class constructors **this** keyword mandatory.
  - 7) One constructor is able to call only one constructor t a time.
  - 8) Constructor calling is must be first line of the constructor.
  - 9) Parameters of constructor are local variables.
- Ex : Test( int a, int b, int c) { }  
//local variables
- 10) To call super class constructors use **super** keyword.

**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**

**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 variables
    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 metod --->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 vriables
    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:-**

The process of acquiring fields(variables) and methods(behaviors) from one class to another class is called inheritance.

1. The main objective of inheritance is code extensibility whenever we are extending class automatically the code is reused.
2. In inheritance one class giving the properties and behavior & another class is taking the properties and behavior.
3. Inheritance is also known as is-a relationship. By using extends keyword we are achieving inheritance concept.
4. extends keyword is providing relationship between two classes when you make relationship able to reuse the code.
5. The class is derived from another class is called sub class (or derived class, child class, extended class).
6. The class from which the sub class is derived is called super class(or base class, Parent class).
7. In java parent class is giving properties to child class and Child is acquiring properties from Parent.
8. 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. if we are creating object for child class, by using that object reference variable it is possible to call both parent and child specific methods.

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

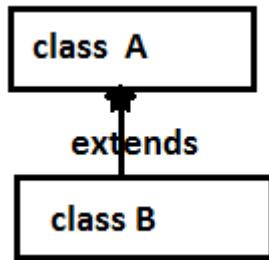
### **Types of inheritance :-**

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:-**

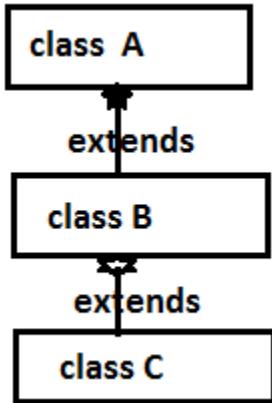
```

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]

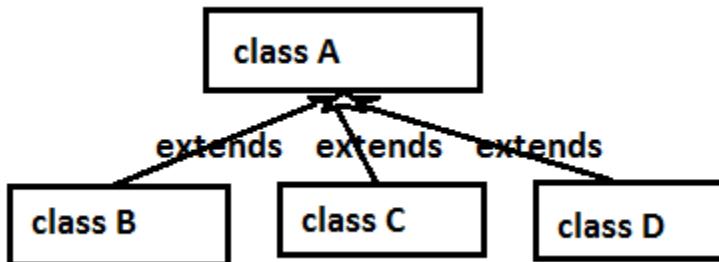
**Example:-**

```

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();
    }
}

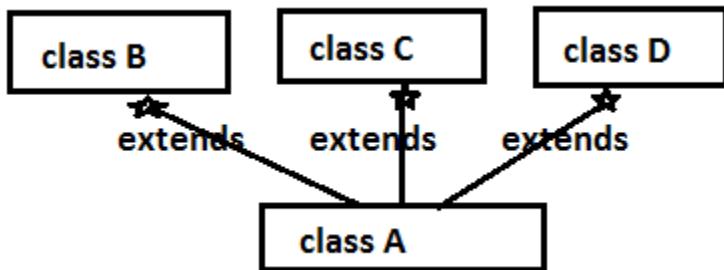
```

**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:-**

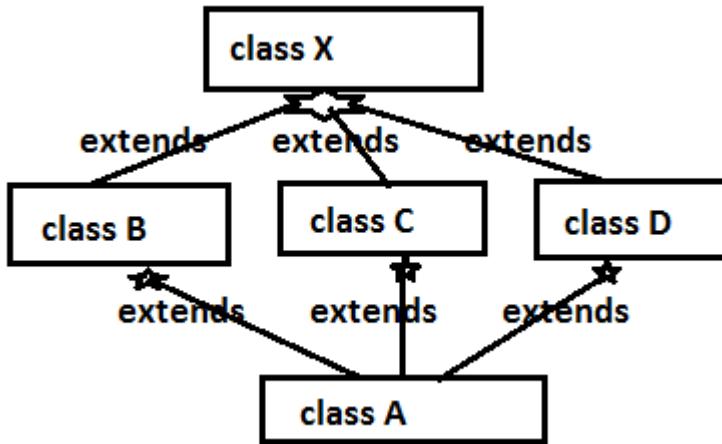
```

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:- Except for the Object class , a class has one direct super class.**

**Note :- a class inherit fields and methods from all its super classes whether directly or indirectly.**

**Note : an abstract class can only be sub classed but cannot be instantiated.**

**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 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**

### 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 Durga
{
    void disp(){System.out.println("durga : 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();
        Durga d = new Durga(); 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.

**Super keyword 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.

- a. Super class methods use super keyword.
- b. 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 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();
    }
}
```

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

```
class Parent
{
    Parent() { System.out.println("parent
0-arg cons"); }
}
class Child extends Parent
{
    /* below code is generated by compiler
    Child()
    {
        super();
    } */
    public static void main(String[] args)
    {
        Child c = new Child();
    }
}
```

**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

**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("chld 0-arg cons");
    }
    Child(int a)
    {
        //super(); generated by compiler
        System.out.println("chld 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
chld 0-arg cons
parent instance block
parent cons
Child instance block
```

**chld 1-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 } explicitly by the programmer
- b. Constructor overloading } implicitly by the JVM('+' addition& concatenation)
- c. Operator overloading }

**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();
     t.m1('a');  t.m1(10);  t.m1(10,20); //three methods execution decided at compilation time
    }
}
```

**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 two 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 can perform more than one operation 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.

➤ res

**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 check the modifiers permission like the sub class method modifier is having same permission or increasing level of parent class method but not decreasing order.
- 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)

#### **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 java 5 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();
    }
}

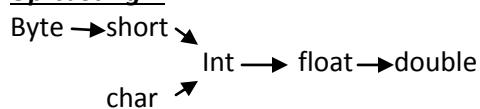
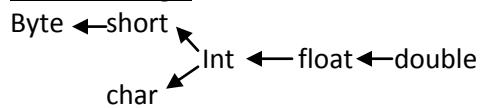
class Dog extends Animal
{
    Dog m1()
    {
        return new Dog();
    }
    public static void main(String[] args)
    {
        Dog d = new Dog();      d.m2();
        Dog d1 = d.m1();        // [d.m1() returns Dog object]
        d1.m2();
        Animal a = new Animal();
        a.m2();
        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");} //override method
    //direct method of child class
    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:-method overriding vs. multilevel inheritance.**

```

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

class Ratan extends Person
{
    void eat(){System.out.println("ratan takes :- 4-apples+4promogranate+3glassmilk ");}
};

class RatanKid extends Ratan
{
    void eat(){System.out.println("ratan kid takes :- 6-times milk");}
    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();
        Ratan p1 = new RatanKid();//[creates object of RatanKid class]
        p1.eat();
    }
};

```

**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) { }

**Method overriding :-**

- 2) To achieve overloading one java class sufficient.
- 3) It is also known as Compile time polymorphism/static binding/early binding.

- 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());
    }
}
class String extends Object
{
    //it is overriding toString()
    public String toString()
    {
        return "String-content";
    }
}
class StringBuffer extends Object
{
    //it is overriding toString()
    public String toString()
```

```

    {
        return "String-content";
    }
};

```

**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]

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

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

```

**GC():-**

- 1) Internally the garbage collector is running to destroy the useless objects.
- 2) By using gc() method we are able to call Garbage Collector explicitly by the developer.
- 3) gc() present in System class and it is a static method.

**Syntax:- System.gc();**

- 3) Whenever garbage collector is destroying useless objects just before destroying the objects the garbage collector is calling finalize() method on that object to perform final operation of particular object.

**Ex:-**

```

class Test
{
    public void finalize()
    {
        System.out.println("ratan sir destroyed");
    }

    public static void main(String[] args)

```

```

    {
        Test t1=new Test();
        Test t2=new Test();
        System.out.println(t1);
        System.out.println(t2);
        t1=null;
        t2=null;
        System.gc();
    }
};

```

**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();
    }
}

```

**Instanceof operator:-**

it is used check the type of the object.

Instanceof operator return Boolean value as a return value.

**Syntax:- Reference-variable instanceof class-name**

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

**while using instanceof operator the reference variable and class Name must have some relationship (Parent-child or child-parent) otherwise compiler will raise compilation error.**

if the reference variable is child of the specified classes at that situation instanceof return true  
 if the reference variable is parent of the specified classes at that situation instanceof return false.

class Fruit

```
{
};
```

class Apple extends Fruit

```
{
```

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

```
        Apple a=new Apple();
        Object o=new Object();
    }
```

**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();
    }
};

```

```
String str="ratan";
Throwable t=new Throwable();
System.out.println(a instanceof Fruit);
System.out.println(a instanceof Object);
System.out.println(str instanceof Object);
System.out.println(o instanceof Throwable);
System.out.println(t instanceof Throwable);
//System.out.println(t instanceof Apple);//compilation error
}
}
class Animal
{}
class Dog extends Animal
{
    public static void main(String[] args)
    {
        Animal a = new Animal();
        Dog d = new Dog();
        Object o = new Object();
        String str="ratan";
        Throwable t = new Throwable();
        /*[child-ref vs Parent-class --->true
           parent-ref vs Child-class ---->false
        }*/
        System.out.println(a instanceof Object);//true [child-ref vs Parent class]
        System.out.println(d instanceof Object);//true [child-ref vs Parent class]
        System.out.println(a instanceof Dog);//false [parent-ref Vs Child-class]
        System.out.println(t instanceof Object);//true [child-ref vs Parent class]
        System.out.println(o instanceof String);//false [parent-ref Vs Child-class]
        //System.out.println(d instanceof String);norelation b/w Dog&String classes
    }
};
```

**Abstraction:-**

There are two types of methods in java

- a. Normal methods
- b. Abstract methods

**Normal methods:- (component method/concrete method)**

Normal method is a method which contains method declaration as well as method implementation.

**Example:-**

```
void m1() --->method declaration
{
    body; --->method implementation
}
```

**Abstract methods:-**

- 1) The method which is having only method declaration but not method implementations such type of methods are called abstract Methods.
- 2) In java every abstract method must end with “ ; ”.

**Example :-**      ***abstract void m1(); ----→method declaration***

Based on above representation of methods the classes are divided into two types

- 1) Normal classes.
- 2) Abstract classes.

**Normal classes:-**

Normal class is a ordinary java class it contains only normal methods if we are trying to declare at least one abstract method that class will become abstract class.

**Example:-**

```
class Test //normal class
{
    void m1() { body ; } //normal method
    void m2() { body ; } //normal method
    void m3() { body ; } //normal method
};
```

**Abstract class:-**

Abstract class is a java class which contains at least one abstract method(wrong definition).

**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.**

```
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 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
};

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();
    }
};
```

**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 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();
    abstract void m4();
};

abstract class Test1 extends Test
{
    void m1() { System.out.println("m1 method"); }
};

abstract class Test2 extends Test1
{
    void m2() { System.out.println("m2 method"); }
};

abstract class Test3 extends Test2
{
    void m3(){ System.out.println("m3 method"); }
};

class Test4 extends Test3
{
    void m4(){ System.out.println("m4 method"); }

    public static void main(String[ ] args)
    {
        Test4 t = new Test4(); //created object of normal class
        t.m1(); t.m2(); t.m3(); t.m4();
    }
};

```

**Example-4 :-**

The **abstract class reference variable is able to hold child class objects.**

```

abstract class Test
{
    abstract void m1();
    abstract void m2();
    abstract void m3();
};

abstract class Test1 extends Test
{
    void m1() { System.out.println("m1 method"); }
};

abstract class Test2 extends Test1
{
    void m2() { System.out.println("m2 method"); }
};

abstract class Test3 extends Test2
{
    void m3(){ System.out.println("m3 method"); }
};

class Test4 extends Test3
{
    public static void main(String[ ] args)
    {
        Test t1 = new Test4(); t1.m1(); t1.m2(); t1.m3(); //parent reference variable child object
        Test1 t2 = new Test4(); t2.m1(); t2.m2(); t2.m3(); //parent reference variable child object
        Test2 t3 = new Test4(); t3.m1(); t3.m2(); t3.m3(); //parent reference variable child object
    }
};

```

**Example-5 :-**

**for the abstract methods it is possible to provide any return type(void, int, char, Boolean.....etc)**

```

abstract class Test1
{
    abstract int m1(char ch);
    abstract boolean m2(int a);
}
class Test2 extends Test1
{
    int m1(char ch)
    {
        System.out.println("char value is:-"+ch);
        return 100;
    }
    boolean m2(int a)
    {
        System.out.println("int value is:-"+a);
        return true;
    }
    public static void main(String[] args)
    {
        Test2 t=new Test2();
        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);
    }
};

```

**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 7:-**

**While providing implementation of abstract methods must check modifier accessing only for (public, private, protected ,default).**

**Note :- in java parent and child classes it is possible to maintains same level modifiers or possible to increase the permissions but it is not possible to decrease the permissions, if we trying to decrease the permission compiler generate error message “attempting to assign weaker access privileges”.**

<u>Parent-class method</u>	<u>child-class method</u>	
<b>Default</b>	default	-->valid (same level)
	protected , public	--> valid (increasing permission )
	Private	--> invalid (decreasing permission)
<b>Public</b>	public	--> valid (same level)
	Default,private,protected	-->invalid(decreasing permission)
<b>Private</b>	private	--> valid (same level)
	Public ,default,protected	--> valid (increasing permission)
<b>Protected</b>	protected	--> valid (same level)
	Public	-->valid (increasing permission)
	Default,private	---->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
}
```

**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 write the functionality that functionality 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 executed.

**Note :- in below example abstract class constructor is executed but object is not created.**

```
abstract class Test
{
    abstract void m1();           //abstract method
    Test()                      //0-arg constructor
    {
        System.out.println("Test abstract class constructor");
    }
};

class Test1 extends Test
{
    void m1(){System.out.println("m1 method");} //normal method
    Test1()                     //1-arg constructor
    {
        super();
        System.out.println("Test1 normal class constructor");
    }
    public static void main(String[] args)
    {
        Test1 t = new Test1();
        t.m1();
    }
};
```

D:\>java Test1

**Test abstract class constructor  
Test1 normal class constructor**

**m1 method**

**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 fallowing 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[] durga)** --->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");
    }
};
```

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

class Child extends Parent
{
};
```

*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 command line arguments
        System.out.println(ratan[0]);
        System.out.println(ratan[1]);
    }
};
```

```
};

D:\>java Test corejava ratan
corejava
ratan
D:\>java Test core java ratan
core
java
D:\>java Test "core java" ratan
core java
ratan
```

**Var-arg method:-**

1. 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","durga"); //String var-arg calling
        t.m1();//var-arg method vs ambiguity [compilation error ambiguous]
    }
}

```

## Packages

### Information regarding packages:-

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

### Syntax:-

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

The packages are divided into two types

- 1) Predefined packages
- 2) User defined packages

### Predefined packages:-

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

Ex:- java.lang,Java.io,Java.awt,Java.util,Java.net.....etc

### Java.lang:-

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

```
java
|-----→lang
      |--→ String(class)
      |--→ StringBuffer(class)
      |--→ Object(class)
      |--→ Runnable(interface)
      |--→ Cloneable(interface)
```

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

### Java.io package:-

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

```
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 those classes are present in the java.net package.

```
java
|-----→net
    |-->Socket(class)
    |-->ServerSocket(class)
    |--> InetAddress(class)
    |-->URL(class)
    |--> SocketOption(interface)
```

**Java.awt package:-**

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

```
java
|-----→awt
    |--> Button(class)
    |--> Checkbox(class)
    |--> Choice(class)
    |--> Adjustable(interface)
```

**User defined packages:-**

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

**Ex 1:-valid**

```
package pack1;
import java.lang.*;
```

**Ex 2:-invalid**

```
package pack1;
package pack2;
```

**Ex3:valid**

```
package pack2;
import java.io.*;
import java.aet.*;
```

**ex 4:-invalid**

```
import java.io.*;
import java.lang.*;
package pack3;
```

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

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

- 1) The package name is must reflect with your organization name. the name is reverse of the organization domain name.

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

**Package name:-** **Package com.dss;**

- 2) Whenever we are working in particular project(Bank) at that moment we have to take the package name is as fallows.

**Project name :-** **Bank**

**package :-** **Package com.dss.Bank;**

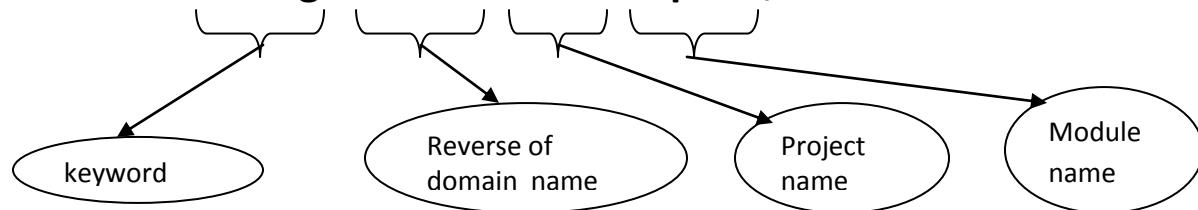
- 3) The project contains the module (deposit) at that situation our package name should reflect with the module name also.

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

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

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

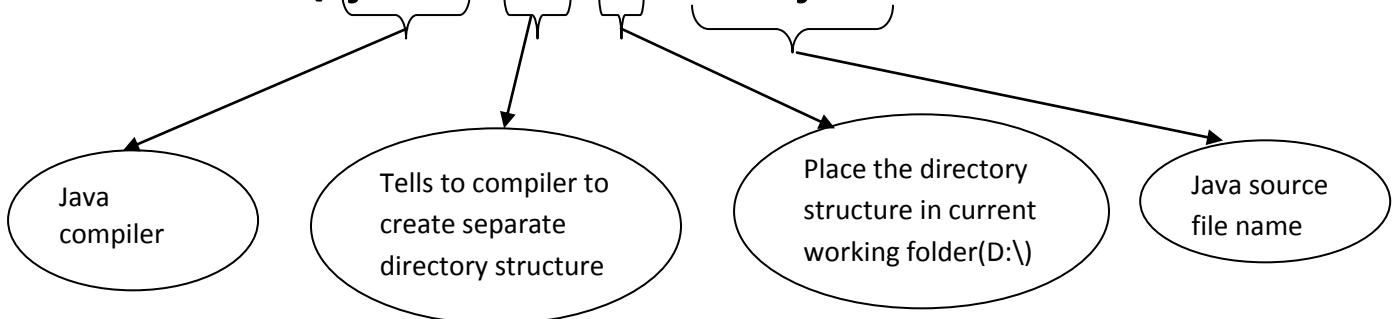
**Package com.dss.bank.deposit;**



**Note:-**

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

**D:\>javac -d . Test.java**



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

```

Com
|-----dss
|-----bank
|-----deposit
|-----(number of .classes will be generated )
  
```

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

Compilation                   :- javac -d . Test.java

```

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

Execution                   :- java com.tcs.onlineexam.Test

**Note:-**

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

#### **Import session:-**

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

#### **Predefined packages support:-**

Ex1:-

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

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

Ex 2:-

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

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

#### **User defined packages support:-**

I am taking two user defined packages are

1) Package pack1;

Class A

{ }

Class B

{ }

2) Package pack2

Class D

{ }

Ex 1:-

```
Import pack1.A;
```

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

Ex 2:-

```
Import pack1.*;
```

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

#### **Note:-**

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

#### **Public:-**

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

#### **Example:-**

**any package can access public classes [public class name source file name must be same]**

```
public class Test
```

```

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

```

**Default modifier:-**

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

**Example:-**

```

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

```

**Note :-**

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

**Private modifier:-**

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

**Example:-**

```

class Parent
{
    private int a=10; //we are able to access only within the Parent class
    private void m1(){System.out.println("parent m1() method");}
};

class Child extends Parent
{
    void m2()
    {
        m1(); //m1() is private here accessing not possible
        System.out.println(a); //a variables is private Child class unable to access
    }
}

```

```

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

```

**Protected modifier:-**

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

**Example :-****A.java:-**

```

package app1;
public class A
{
    protected int fee=1000;
    protected void course() { System.out.println("corejava]"); }
};

```

**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);
        b.course();
    }
};

```

**Example :-demonstrate the user defined packages and user defined imports.****Durga.java:-**

```

package com.dss.states.info;
public class Durga
{
    public void ap() { System.out.println("jai samaikandra"); }
    public void tl() { System.out.println("jai jai telengana"); }
    public void others() { System.out.println("jai jai jai others"); }
};

```

**Tcs.java:-**

```

package com.tcs.states.details;
import com.dss.states.info.Durga; //importing package
class Tcs
{
    public static void main(String[] args)
    {
        Durga d = new Durga();
        d.ap();
        d.tl();
        d.others();
    }
}

```

```
D:\durgasoft>javac -d . Durga.java
D:\durgasoft>javac -d . Tcs.java
D:\durgasoft>java com.tcs.states.details.Tcs
jai samaikandra
jai jai telengana
jai jai jai others
```

**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();
    }
};
```

modifier	Private	no-modifier	protected	public
Same class	yes	yes	yes	yes
Same package sub class	no	yes	yes	yes
Same package non sub class	no	yes	yes	yes
Different package sub class	no	no	yes	yes
Different package non sub class	no	no	no	yes

**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 java programming.

**Ex:-without static import**

```
import java.lang.*;
class Test
{
    public static void main(String[] args)
    {
        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");
    }
};
```

**Ex:-**

```
package com.dss.java.corejava;
public class Durga
{
    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");
}
};
```

**File -2 without static import**

```

package com.tcs.course.coursedetails;
import com.dss.java.corejava.*;
class Tcs
{
    public static void main(String[] args)
    {
        System.out.println(Durga.fee);
        Durga.course();
        Durga.duration();
        Durga.trainer();
    }
}
```

**File -2 with static import**

```

package com.tcs.course.coursedetails;
import static com.dss.java.corejava.Durga.*;
class Tcs
{
    public static void main(String[] args)System.out.println(fee);
        course();
        duration();
        trainer();
    }
}
```

**Example :-**

- when you import main package we are unable to use sub package classes at that situation to use sub package classes must import sub packages also.

**A.java:-**

```

package jav.corejava;
public class A
{
    public void m1()
    {System.out.println("core java World!");}
}
```

**Package structure:-**

```

jav
|-->corejava
    |--->A.class
```

**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
```

**C.java:-**

```
package jav.corejava.advjava.structs;
public class C
{
    public void m1()
    {System.out.println("Structs World!");
    }
}
```

**Package structure :-**

```
jav
|-->corejava
    |--->A.class
    |--->advjava
        |--->B.class
        |--->structs
            |--->C.class
```

**MainTest.java:**

```
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();
    }
}
```

**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 userdefined API (application programming interface document):-**

1. API document nothing but user guide.
2. Whenever we are buying any product the manufacturing people provides one document called user guide. By using userguide we are able to use the product properly.
3. James gosling is developed java product whenever james gosling is 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 userdefined api document for the userdefined 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 at least 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 by default 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 completely then we should go for abstract class**

**Note-3 :if you know the implementation completely 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();
}
}

```

**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 it
{
    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();
    }
};

```

**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();
    }
};
```

**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();
    }
};
```

**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:-**

```

class extends class
class implements interface
interface extends interface

```

<b>class A extends B</b>	<b>=====&gt;valid</b>
<b>class A extends B,C</b>	<b>=====&gt;invalid</b>

<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 (extends must be first)
<i>class A implements it1 extends B</i>	=====> invalid

**Adaptor class:-**

It is an intermediate class between the interface and user defined class. And it contains empty implementation of interface methods.

**Limitation of interface**

```
interface it
{
    void m1();
    void m2();
    ;
    ;
    void m100()
}
Class Test implements it
{
    Must provide implementation of 100 methods
    otherwise compiler raise compilation error
}
```

```
interface it
{
    void m1();
    void m2();
    ;
    ;
```

**advantage of adaptor classes**

```
void m100()
}
class Adaptor implements it
{
    void m1(){}
    void m2(){}
    ;
    ;
    void m100(){}
};
class Test implements it
{
    must provide the 100 methods implementation
};
class Test extends Adaptor
{
    provide the implementation of required
    methods.
};
```

**Ex:-**

```
interface it1
{
    void m1();
    void m2();
}
class X implements it1
{
    public void m1(){}
    public void m2(){}
};
class Test1 implements it1
{
```

```

public void m1()
{
    System.out.println("m1 method");
}
public void m2()
{
    System.out.println("m2 method");
}
public static void main(String[] args)
{
    Test1 t=new Test1();
    t.m1();
    t.m2();
}
};

class Test2 extends X
{
    public void m1()
    {
        System.out.println("adaptor m1
method");
    }
    public static void main(String... arhs)
    {
        Test2 t=new Test2();
        t.m1();
    }
};

```

**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***

#### **Java.lang.String:-**

- String is a final class it is present in java.lang package.
- String is nothing but a group of characters or character array enclosed with in double quotes..

**Constructors of string class:-**

```
String str=new String(java.lang.String);
String str=new String(char[]);
String str=new String(char[],int,int);
```

#### **Example :-**

```
class Test
{
    public static void main(String[] args)
    {
        String str="ratan";
        System.out.println(str);
        String str1=new String("ratan");
        System.out.println(str1);
        String str2=new String(str1);
        System.out.println(str2);
        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);
    }
}
```

#### **Java.lang.StringBuffer:-**

- StringBuffer is a final class present in the java.lang package.
- StringBuffer is used to take group of characters or character array with in double quotes.

**Constructors of StringBuffer :-**

```
StringBuffer sb=new StringBuffer();
StringBuffer sb1=new StringBuffer(int capacity);
StringBuffer sb2=new StringBuffer(String str);
```

```
class Test
{
    public static void main(String[] args)
    {
        StringBuffer sb=new StringBuffer();
        System.out.println(sb.capacity());           //default capacity 16
        StringBuffer sb1=new StringBuffer(5);
        System.out.println(sb1.capacity());          //your provided capacity
        StringBuffer sb2=new StringBuffer("rattaiah");
        System.out.println(sb2.capacity());          //initial capacity+ provided string length 24
        System.out.println(sb2.length());            //8
    }
}
```

**Java.lang.String object creation:-**

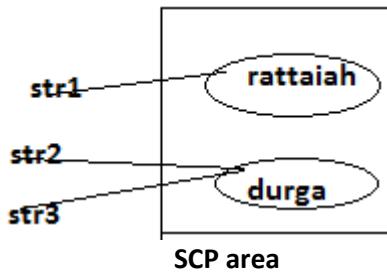
There are two approaches to create String class object.

- 1) without using new operator(by using literal)
- 2) by using new operator

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

**Creating a string object without using new operator (by using literal):-**

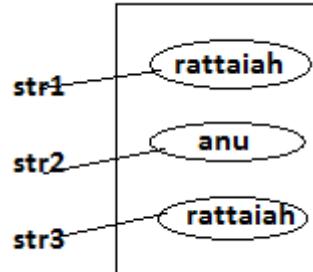
- Whenever we are creating String object without using new operator the objects are created in SCP (String constant pool) area.
- ```
String str1="rattaiah";
String str2="durga";
String str3="durga";
```



- When we create object in SCP area then just before object creation it is always checking previous objects.
  - If the previous object is there with the same content then it won't create new object the reference variable pointing to existing object.
  - If the previous objects are not matched then it create new object.
- SCP area does not allow duplicate objects.

**Creating a string object by using new operator**

- Whenever we are creating String object by using new operator the object created in heap area.
- ```
String str1=new String("rattaiah");
String str2 = new String("anu");
String str3 = new String("rattaiah");
```



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

**Java.lang.StringBuffer:-**

only one approach to create StringBuffer object  
*ex:- (by using new operator)*

```
StringBuffer sb1 = new StringBuffer("ratan");
```

**Example:-**

```

class Test
{
    public static void main(String[] args)
    {
        //two approaches to create a String object
        String str1 = "ratan";           //without using new operator
        System.out.println(str1);
        String str2 = new String("anu"); //by using new operator
        System.out.println(str2);

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

```

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

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

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

**Example :-**

```

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

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

```

**Java.lang.String 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("durga");
        String str2 = new String("durga");
        //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.

**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
    }
};
```

**== 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("durga");
        String str4 = new String("durga");
        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
    }
}

```

**Java.lang.String class methods:-****1) CompareTo() & compareTolgnoreCase():-**

- By using compareTo() we are comparing two strings character by character, such type of checking is called lexicographically checking or dictionary checking.
- compareTo() is return type is integer and it returns three values
  - 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.
- compareTolgnoreCase() method comparing two strings character by character by ignoring case.

```
class Test
{
    public static void main(String... ratan)
    {
        String str1="ratan";
        String str2="durga";
        String str3="ratan";
        String str4="Durga";
        String str5="ratna";
        System.out.println(str1.compareTo(str2));//14
        System.out.println(str1.compareTo(str4));//46
        System.out.println(str2.compareTo(str4));//32
        System.out.println(str4.compareTo(str2));//-32
        System.out.println(str1.compareTo(str3));//0
        System.out.println(str1.compareTo(str5));//-13
        System.out.println(str1.compareToCase(str5));//-13
        System.out.println(str2.compareToCase(str4));//0
        System.out.println(str1.compareToCase(str3));//0
    }
};
```

**Concat():-**

Concat() method present in the String class and it is used to combine the two String.

Ex :-

```
class Test
{
    public static void main(String[] args)
    {
        String age="22";
        String s="he is "+age+" years old.";
        System.out.println(s);

        String str1="durga";
```

```

String str2="ratan";
String str3=str1.concat(str2);
System.out.println(str3);

int a=22;
String s1="he is "+a+" years old.";
System.out.println(s1);

String s2="six "+2+2+2;
System.out.println(s2);

String s3="six "+(2+2+2);
System.out.println(s3);
}
}

```

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

length( )----→method    length--→variable

length variable used to find length of the array

```

int [] a={10,20,30};
System.out.println(a.length); //3
length() is method used to find the length of the given string.
String str="rattaiah";
System.out.println(str.length());//8

```

**charAt(int):-**

By using above method we are able to extract the character from particular index position.

Ex:-

```

class Test
{
    public static void main(String[] args)
    {
        String str="ratan";
        System.out.println(str.charAt(1));

        char ch="ratan".charAt(2);
        System.out.println(ch);
    }
}

```

**Split(String):-**

By using split() method we are dividing string into number of tokens.

Ex:-

```

class Test
{
    public static void main(String[] args)
    {
        String str="hi rattaiah how r u";
        String[] str1=str.split(" ");

```

```

        for (int i=0;i<str1.length ;i++ )
        {
            System.out.println(str1[i]);
        }
    };
getBytes():-

```

By using this method we are converting String into the byte[] .the main aim of the converting String into the byte[] format is some of the networks are supporting to transfer the data in the form of bytes only at that situation is conversion is mandatory.

**Ex:-**

```

class Test
{
public static void main(String[] args)
{
String str="rattaiah";
byte[] b=str.getBytes();
System.out.println(b);
String str1=new String(b);
System.out.println(str1);
}

```

**trim():-**

- 1) trim() is used to remove the trail and leading spaces
- 2) this method always used for memory saver

**Ex:-**

```

class Test
{
public static void main(String[] args)
{
String str="      ratan      ";
System.out.println(str.length());//7
System.out.println(str.trim());//ratan
System.out.println(str.trim().length());//5
System.out.println("      ratan      ".trim());//ratan
}
}

```

**replace(char oldchar,char newchar) replace(Stirng oldString,Stirng newString):-**

by using above method we are replacing the particular character of the String. And particular portion of the string.

**Ex:-**

```

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
}

```

```

}

Ex :-
class Test
{
    public static void main(String[] args)
    {
        String str1="durga software solutions";
        System.out.println(str1);
        System.out.println(str1.replace("software","hardware")); // Durga hardware solutions
    }
}

```

**toUpperCase() and toLowerCase():-**

The above methods are used to convert the lower case to the uppercase and uppercase to lowercase character.

Ex:-

```

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

```

**Java.lang.String.endsWith() and Java.lang.String.startsWith():-**

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.

```

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
    }
}

```

**substring(int startingposition) & substring(int startingposition,int endingposition):-**

By using above method we are getting substring from the whole String.

In the above methods

starting position parameter value is including

ending position parameter value is excluding

```

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

```

```

    {
        String str="ratan how r u";
        System.out.println(str.substring(2));//tan how r u
        System.out.println(str.substring(1,7));//atan h
        System.out.println("ratansoft".substring(2,5));//tan
    }
}

```

**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=30000;
        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.

Ex:-

```

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());
        }

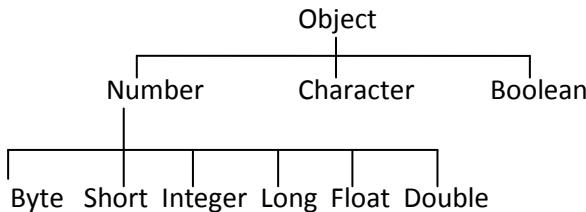
        //used our string to split giver String
        String str1 = "hi,rata,mf,sdfsdf,ara"; StringTokenizer st1 = new
        StringTokenizer(str1,"");
        while (st1.hasMoreElements())
        {
            System.out.println(st1.nextElement());
        }

    }
}
```

## Wrapper classes

- To represent primitive data types as a Object form we required some classes these classes are called wrapper classes.
- All wrapper classes present in the java.lang package.
- We are having 8 primitive data types hence sun peoples are providing 8 wrapper classes.

### **Wrapper classes hierarchy:-**



### **Constructors of wrapper classes:-**

To create objects of wrapper classes All most all wrapper classes contain two constructors but float contains three constructors & char contains one constructor.

```

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');
  
```

### **Constructor approach to create wrapper Object:-**

<u>datatype</u>	<u>wrapper-class</u>	<u>constructors</u>
byte	Byte	byte, String
short	Short	short, String
int	Integer	int, String
long	Long	long, String
float	Float	double, float, String
double	Double	double, String
char	Character	char
boolean	Boolean	boolean, String

### **Example :-**

```

class Test
{
    public static void main(String[] args)
    {
        //primitive variables
        int a=10;
        System.out.println(a);
        char ch = 'a';
        System.out.println(ch);
        //wrapper objects
        Integer i1 = new Integer(10);
        System.out.println(i1);
        Integer i2 = new Integer("1000");
      
```

```

        System.out.println(i2);
        Character ch1 = new Character('a');
        System.out.println(ch1); //a
        /*Integer x = new Integer("ten"); "ten" is unable to convert into integer
        System.out.println(x); //NumberFormatException
        */
    }
}

```

**Methods :-**

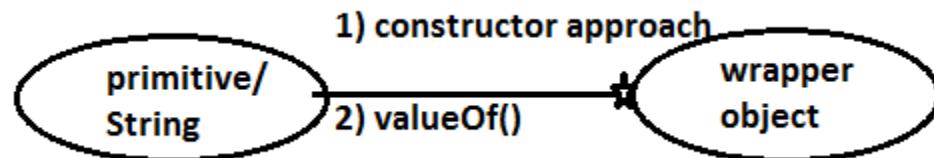
- 1) **valueOf()**
- 2) **XXXValue()** here xxx = datatype
- 3) **parseXXX()** here xxx = datatype
- 4) **toString()**

**The main importance of wrapper classes:-**

1. To convert a data types into the object means we are giving object from data types by using constructor.
2. To convert String into the data types by using parsexxx() method.

**1) valueOf():-**

By using valueof() we are creating wrapper object and it is a alternative to the constructor.

**Example:-**

```

class Test
{
    public static void main(String[] args)
    {
        //constructor approach create wrapper objects
        Integer a1 = new Integer(10);
        System.out.println(a1);
        Integer a2 = new Integer("100");
        System.out.println(a2);
        //valueOf() approach to create wrapper Object
        Integer i1 = Integer.valueOf(10);
        System.out.println(i1);
        Integer i2 = Integer.valueOf("1000");
        System.out.println(i2);
        Character ch = Character.valueOf('a');
        System.out.println(ch);
    }
}

```

**Example :-conversion of primitive to String**

```
class Test
{
    public static void main(String[] args)
    {
        //conversion of primitive to String
        int a=10;
        int b=20;
        String str1=String.valueOf(a);
        String str2=String.valueOf(b);
        System.out.println(str1+str2);
    }
}
```

**XxxValue():-**

By using XXXValue() method we are converting wrapper objects into the corresponding primitive values.

**Example :-**

```
class Test
{
    public static void main(String[] args)
    {
        Integer i=Integer.valueOf(150);
        System.out.println("byte value :" +i.byteValue());//-106
        System.out.println("short value :" +i.shortValue());//150
        System.out.println("int value :" +i.intValue());//150
        System.out.println("long value :" +i.longValue());//150
        System.out.println("float value :" +i.floatValue());//150.0
        System.out.println("double value :" +i.doubleValue());//150.0
        Character c=new Character('s');
        char ch=c.charValue();
        System.out.println(ch);
        Boolean b=new Boolean(false);
        boolean bb=b.booleanValue();
        System.out.println(bb);
    }
}
```

**parseXXX():-**

By using parseXXX() method we are converting String into the corresponding primitive.



**Example :-**

```
class Test
{
    public static void main(String[] args)
    {
        String str1="10";
        String str2="20";
        System.out.println(str1+str2);//1020
        int a=Integer.parseInt(str1); //converting String into integer value
        float f=Float.parseFloat(str2); //converting String into float value
        System.out.println(a+f);//30.0
    }
}
```

**toString():-**

- It prints String representation of Object.
- Always The `toString()` method return type is String Object.

```
class Test
{
    public static void main(String[] args)
    {
        Test t =new Test();
        System.out.println(t.toString()); //Object class toString is executed

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

        StringBuffer sb = new StringBuffer("ratan");
        System.out.println(sb.toString()); //StringBuffer class toString is executed

        Integer i = new Integer(100);
        System.out.println(i.toString()); //Integer class toString is executed

        //Integer toString returns String class Object
        String strObject = i.toString();
        System.out.println(strObject);
    }
}
```

**Example :- all classes `toString()` method returns String class object.**

```
class Test
{
    public static void main(String[] args)
    {
        Integer i = new Integer(10);
        String str = i.toString(); //returns String class Object
        System.out.println(str);
        StringBuffer sb = new StringBuffer("ratan");
        System.out.println(i.toString());
        String str1 = sb.toString(); //returns String class Object
        System.out.println(str1);
        Float f = new Float("10.5");
        String str2 = f.toString(); //returns String class Object
        System.out.println(str2);
    }
}
```

**Example :-**

- 1) When we print reference variables internally it calls `toString()` method.
- 2) On primitive variables we are unable to call `toString()` method if we are trying to call `toString()` compiler generate error message "int cannot be dereferenced".

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

**Compilation error:-**

D:\morn11>javac Test.java

*Test.java:6: int cannot be dereferenced*

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

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

Until 1.4 version we are not allowed to place primitive in the palc wrapper and wrapper in the place of primitive. The programmer is responsible person to do the explicit conversion primitive to the wrapper and wrapper to the primitive.

**Autoboxing:-**

```
Integer i=10;
System.out.println(i);
```

The above statement does not work on the 1.4 and below versions. The auto conversion of the primitive into the Wrapper object is called the autoboxing these conversions done by compiler at the time of compilation.

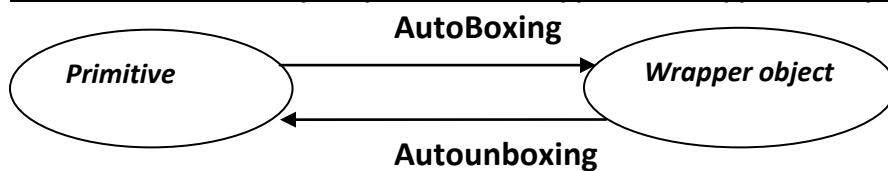
**Autounboxing:-**

```
int a=new Integer(100);
System.out.println(a);
```

The auto conversion of the wrapper object to the primitive value is called autounboxing and these conversions are done by compiler at the time of compilation.

**Example:-**

```
class Test
{
    public static void main(String[] args)
    {
        //autoboxing [internal conversion of primitive to wrapper object]
        Integer i = 100;
        System.out.println(i);
        //autounboxing [internal conversion of wrapper object to primitive]
        int a = new Integer(1000);
        System.out.println(a);
    }
}
```

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

## Java .io package

The data stored in computer memory in two ways.

**1) Temporary storage.**

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

**2) Permanent storage.**

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

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

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

**Java.io.File:-**

By using file class we are able to find information about file like.

- a) Size of the file.
- b) Recent modify date.
- c) Whether the file exists or not.

File object creation

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

<b><u>boolean canRead()</u>:-</b>	Returns true if a file is readable
<b><u>boolean canWrite()</u>:-</b>	Returns true if a file is writable.
<b><u>boolean exists()</u> :-</b>	Returns true if file exists.
<b><u>String getName()</u> :-</b>	Returns file name
<b><u>String getPath()</u> :-</b>	Returns the file's path.
<b><u>String getParent()</u> :-</b>	Returns the name of the folder in which the file can be found.
<b><u>long length()</u> :-</b>	Returns the file's size.
<b><u>long lastModified()</u> :-</b>	Returns the time the file was last modified.

**C:\home\user\Report**

The character used to separate the directory names (also called the *delimiter*) is specific to the file system: The Solaris OS uses the forward slash (/), and Microsoft Windows uses the backslash slash (\).

**InputStream** → abstract class contains methods to perform input operations.

**OutputStream** → abstract class contains methods to perform output operations.

**FileInputStream** → Child of InputStream that provides capability to read the data from disk file.

**FileOutputStream** → Child of OutputStream that provides capability to write data to disk file.

**PrintStream** → Child of FileOutputStream it is child of OutputStream

**Stream :-**

Stream is a channel it supports continuous flow of data from one place to another place  
Java.io is a package which contains number of classes by using those 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:-

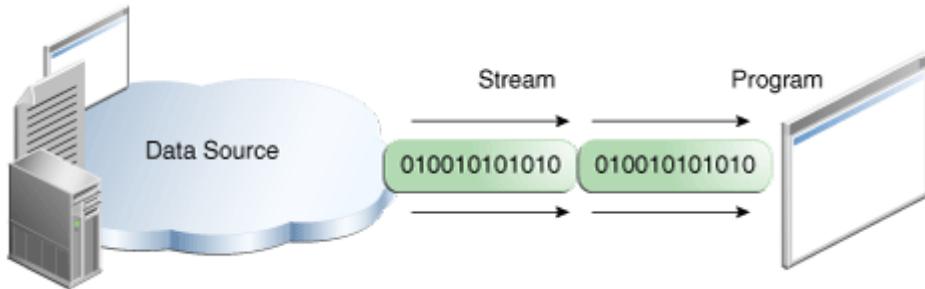
1. Byte oriented streams.(supports byte formatted data to transfer)
2. Character oriented stream.(supports character formatted data to transfer)

### Byte oriented streams:-

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

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

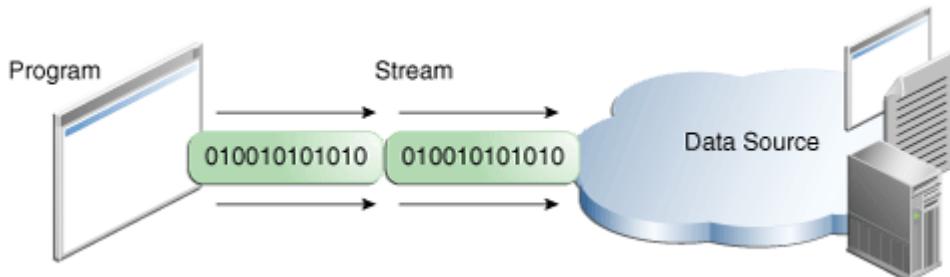
To read the data from the .txt file we have to read() method.



#### **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
{
    static FileInputStream fis;
    static FileOutputStream fos;
    public static void main(String[] args)
    {
        try{
            fis=new FileInputStream("get.txt");
```

```

fos=new FileOutputStream("set.txt",true);
int c;
while ((c=fis.read())!=-1)
{
    fos.write(c);
}
fis.close();
fos.close();
}
catch(IOException io)
{
    System.out.println("getting IOException");
}
}

Ex:-it will support one character at a time.
import java.io.*;
class Test
{
    static FileReader fr;
    static FileWriter fw;
    public static void main(String[] args)
    {
        try{
            fr=new FileReader("get.txt");
            fw=new FileWriter("set.txt",true);

            int c;
            while ((c=fr.read())!=-1)
            {
                fw.write(c);
            }
            fr.close();
            fw.close();
        }
        catch(IOException io)
        {
            System.out.println("getting IOException");
        }
    }
}

```

**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.

```

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)
            {
                pw.println(line);
            }
            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 fallowing classes.

1. FileInputStream
2. ObjectInputStream

Ex:-Student.java

```
import java.io.Serializable;
public class Student implements Serializable
{
    int id;
    String name;
    int marks;

    public Student(int id, String name,int marks)
    {
        this.id = id;
        this.name = name;
        this.marks=marks;
    }
}
```

**To perform serialization :- we are writing the object data to the file called abc.txt file we are transferring that file across the network.**

```
import java.io.*;
class Serializable1
{
    public static void main(String args[])throws Exception
    {
        Student s1 =new Student(211,"ravi",100);
        FileOutputStream fos=new FileOutputStream("abc.txt",true);
        ObjectOutputStream oos=new ObjectOutputStream(fos);
        oos.writeObject(s1);
        oos.flush();
        System.out.println("Serializable process success");
    }
}
```

**To perform deserialization:- in the network the file is available with java data to read the data we have to go for deserialization.**

```
import java.io.*;
class Deserialization
{
    public static void main(String args[])throws Exception
    {
        //deserialization process
        FileInputStream fis=new FileInputStream("abc.txt");
```

```

ObjectInputStream ois=new ObjectInputStream(fis);
Student s=(Student)ois.readObject();

System.out.println("the student name is:"+s.name);
System.out.println("the stuent id is:"+s.id);
System.out.println("the student marks:"+s.marks);
System.out.println("deserialization success");

}

}

import java.io.*;
class Student implements Serializable
{
    String sname;
    int smarks;
    int sno;
    Student(String sname,int smarks,int sno)
    {
        this.sname = sname;
        this.smarks = smarks;
        this.sno = sno;
    }
    public static void main(String[] args)throws Exception
    {
        Student s=new Student("ratan",100,111);

        //serialization
        FileOutputStream fos = new FileOutputStream("venkat.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(s);
        System.out.println("*****serialization completed*****");

        //deserialization
        FileInputStream fis = new FileInputStream("venkat.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Student s1 = (Student)ois.readObject();
        System.out.println("student name--> "+s1.sname);
        System.out.println("student marks--> "+s1.smarks);
        System.out.println("student number--> "+s1.sno);
        System.out.println("*****Deserialization 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.*;
import java.io.Serializable;
class Student implements Serializable
{
    transient int id=100;
    transient String name="ravi";
}
class Serializable1
{
    public static void main(String args[])throws Exception
    {
        Student s1=new Student();
        System.out.println("the stuent id is:"+s1.id);
        System.out.println("the student name is:"+s1.name);
        FileOutputStream fos=new FileOutputStream("ratan.txt",true);
        ObjectOutputStream oos=new ObjectOutputStream(fos);
        oos.writeObject(s1);
        FileInputStream fis=new FileInputStream("ratan.txt");
        ObjectInputStream ois=new ObjectInputStream(fis);
        Student s=(Student)ois.readObject();
        System.out.println("the stuent id is:"+s.id);
        System.out.println("the student name is:"+s.name);
    }
}
```

## Exception Handling

### Information regarding Exception :-

- 1) Dictionary meaning of the exception is abnormal termination.
- 2) Exception is an event which occurs during program execution time that disturb the normal flow of execution.
- 3) An exception is a problem occurred during execution time of the program.
- 4) An unwanted unexpected event that disturbs or terminates normal flow of execution called exception.
- 5) Every Exception is a predefined class present in some packages.
  - a. **ArithmaticException**-----→java.lang
  - b. **SQLException**-----→java.sql
  - c. **ServletException**-----→javax.servlet
  - d. **IOException**-----→java.io
- 6) 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 sun micro systems standards The Exceptions are divided into three types

- 1) **Checked Exception.**
- 2) **Unchecked Exception.**
- 3) **Error.**

### checkedException:-

1. The Exceptions which are checked by the compiler at the time of compilation is called Checked Exceptions.  
Ex:- *IOException,SQLException,InterruptedException etc.....*
2. In our java application whenever we are getting checked Exception the compiler is able to give intimation to developer regarding handle the Exception.
3. If the application contains checked exception the code wont be compiled so must handle the checked Exception in two ways
  - a. By using try-catch block.
  - b. By using throws keyword.
4. In java must handle the checked Exception by using try-catch or throws keyword then only code compiled it means for the checked Exception try-catch or throws keywords mandatory to handle the Exception.
5. The checked Exception caused due to predefined methods it means whenever we are using predefined methods(**not all predefined only for exceptional methods**) in our application we are getting checked Exception.

#### **There are two types of predefined methods**

- a. Exceptional methods
  - i. public static native void sleep(long) throws java.lang.InterruptedException
  - ii. public boolean createNewFile() throws java.io.IOException

- iii. public abstract java.sql.Statement createStatement() throws java.sql.SQLException.....etc
- b. Normal methods
  - i. public long length();
  - ii. public java.lang.String toString();
  - iii. public void destroy();
- 6. In our application whenever we are using exceptional methods the code won't be compiled because these methods throws checked exception hence must handle the exception by using try-catch or throws keyword.

Ex:- in below example we are using sleep() method it is a exceptional methods it throws checked Exception(**InterruptedException**) hence must handle that checked exception by using try-catch or throws keyword then only code is compiled otherwise compilation error "**unreported exception InterruptedException; must be caught or declared to be thrown**"

#### **Example:-**

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("ratan-1");
        Thread.sleep(4000);
        System.out.println("ratan-2");
        Thread.sleep(4000);
        System.out.println("ratan-3");
        Thread.sleep(4000);
        System.out.println("ratan-4");
    }
}
```

#### **Compilation Error:-**

Test.java:6: unreported exception InterruptedException; must be caught or declared to be thrown

    Thread.sleep (4000);

**Note: - Whenever we are getting above message application having checked Exception must handle that exception by using try-catch blocks or throws keyword.**

#### **Unchecked Exception:-**

1. The exceptions which are not checked by the compiler at the time compilation are called uncheckedException.  
Ex:-     ArithmaticException,NullPointerException, etc.....
2. If the application contains checked Exception the compiler is unable to give the information it means code is compiled but at runtime program terminated abnormally and JVM(default Exceptional handler )give exception information.
3. Whenever we are getting unchecked exception the code is compiled but at runtime program terminated abnormally hence handle the exception by using try-catch blocks or throws keyword to get normal termination.

**Note-1:-**

**For the checked Exception try-catch blocks or throws keyword are mandatory then only code is compiled but for the unchecked Exception try-catch blocks or throws keyword optional it means code is compiled but at runtime program 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 to get normal termination of application.**

**Error:-**

- Exceptions are caused by developer's mistakes these are recoverable. But errors are caused by lack of system resources these are non recoverable.

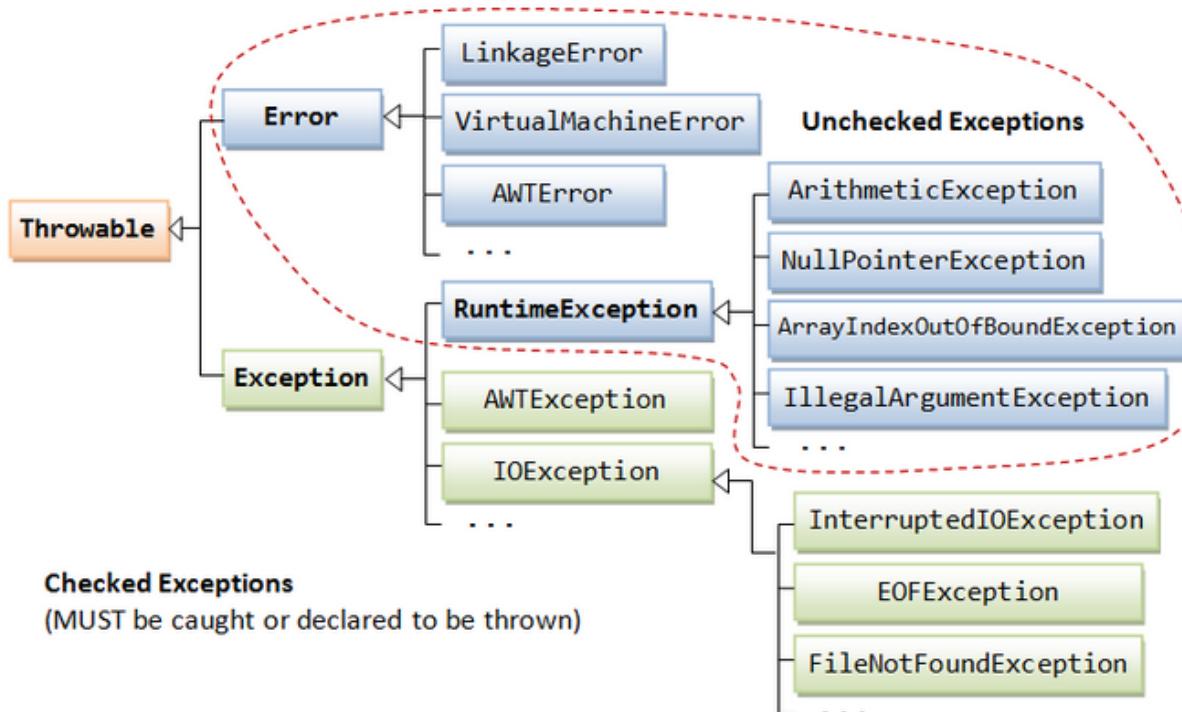
Ex:- StackOverflowError,OutOfMemoryError,AssertionError etc.....

**Example:-**

```
class Test
{
    public static void main(String[] args)
    {
        Test[] t = new Test[100000000];
    }
}
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
at Test.main(Test.java:5)
```

**Exception Handling Tree Structure:-**

Root class of exception handling is Throwable class



- 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)); //Exception statement
        System.out.println("rattan kids");
    }
}
```

D:\>java Test

rattan 1st class

rattan 2st class

rattan inter

rattan trainer

Exception in Thread “main” java.lang.ArithmeticException: / 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 -2:-**

- 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("program starts");
        try
        {
            int[] a={10,20,30};
            System.out.println(a[3]);
        }
        catch(ArithmaticException ae)
        {
            System.out.println("we are getting exception");
        }
        System.out.println("rest of the app");
    }
}
```

```
D:\>java Test
program starts
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
at Test.main(Test.java:9)
```

**Example-3:-**

If there is no exception in try block the catch blocks are won't be executed.

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("ratan task starts");
        try
        {
            System.out.println("durgasoft");
            System.out.println("ratansoft");
        }
        catch(ArithmetricException ae)
        {
            System.out.println("catch block");
        }
        System.out.println("rest of the code");
    }
}
```

**Example -4:-in Exception handling independent try blocks are not allowed.**

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println ("ratan task starts");
        try
        {
            System.out.println ("durgasoft");
            System.out.println ("ratansoft");
        }
        System.out.println ("rest of the code");
    }
}
```

D:\>javac Test.java

Test.java:6: 'try' without 'catch' or 'finally'

**Example-5:-**

In between try-catch independent statements are not allowed. If we are declaring independent statements the compiler will raise compilation error “ 'try' without 'catch' or 'finally' ”.

```
class Test
{
    public static void main(String[] args)
    {
        try{
            System.out.println("durga");
            System.out.println(10/0);
        }
        System.out.println("ratan");
        catch(ArithmetricException e)
        {
            System.out.println(10/2);
        }
        System.out.println("rest of the app");
    }
}
```

```

        }
    }
}
```

**Example-6:-**

If the exception is raised in other than try block it is always abnormal termination.

The exception raised in catch block it is always abnormal termination.

```

class Test
{
    public static void main(String[] args)
    {
        try{
            System.out.println("ratan");
            int a=10/0;
        }
        catch(ArithmaticException ae)
        {
            System.out.println(10/0);
        }
        System.out.println("rest of the code");
    }
}

```

D:\>java Test

ratan

Exception in thread "main" java.lang.ArithmaticException: / by zero  
at Test.main(Test.java:11)

**Example -7:-**

- 1) If the exception raised in try block remaining code of try block won't be executed.
- 2) Once the control is out of the try block the control never entered into try block once again.

**1 & 2 won't be executed**

```

class Test
{
    public static void main(String[] args)
    {System.out.println("program starts");
        try
        {
            int a=10/0;
        (1)     System.out.println("ratan");
        (2)     System.out.println("anu");
        }
        catch(ArithmaticException e)
        {
            int a=10/5;
            System.out.println(a);
        }
        System.out.println("rest of the code ");
    }
}

```

**1 & 2 will be executed.**

```

class Test
{
    public static void main(String[] args)
    {System.out.println("program starts");
        try
        {
            System.out.println("ratan");(1)
            System.out.println("anu");(2)
            int a=10/0;
        }
        catch(ArithmaticException e)
        {
            int a=10/5;
            System.out.println(a);
        }
        System.out.println("rest of the code ");
    }
}

```

**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("getting Exception" +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");  
    }  
}  
  
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");  
    }  
}
```

**Possibilities of try-catch blocks:-****Possibility-1**

```
try { }
catch () { }
```

**Possibility-2**

```
try
{     }
catch ()
{     }
try
{     }
catch ()
{     }
```

**Possibility-3**

```
try
{     }
catch () { }
catch () { }
```

**Possibility-4**

```
try
{     try
{         }
catch ()
{     }
}
catch () { }
```

**Possibility-5**

```
try
{     }
catch ()
{     try
{         }
catch ()
{     }
}
```

**Possibility-6**

```
try
{     try
{         }
catch ()
{     }
}
catch ()
{     try
{         }
catch ()
{     }
}
```

**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.
  - b. streams closing.
  - c. Object destruction .
- 3) It is not possible to write finally alone.
 

a. <i>try-catch-finally</i>	-- →valied
b. <i>try-catch</i>	-- →valied
c. <i>catch-finally</i>	-- →invalied
d. <i>try-catch-catch-finally</i>	-- →valied
e. <i>try-finally</i>	-- →valied
f. <i>catch-catch-finally</i>	-- →invalied
g. <i>Try</i>	-- →invalied
h. <i>Catch</i>	-- →invalied
i. <i>Finally</i>	- →invalied

**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.

**Case 1:-**

```

class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("try");
        }
        catch (ArithmaticException ae)
        {
            System.out.println("catch");
        }
        finally
        {
            System.out.println("finally");
        }
    }
}

```

D:\morn11>java Test

**try**

**finally**

**case 2:-**

```

class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println(10/0);
        }
        catch (ArithmaticException ae)
        {
            System.out.println("catch");
        }
        finally
        {
            System.out.println("finally");
        }
    }
}

```

D:\morn11>java Test

**catch**

**finally**

**case 3:-**

```
class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println(10/0);
        }
        catch (NullPointerException ae)
        {
            System.out.println("catch");
        }
        finally
        {
            System.out.println("finally");
        }
    }
}
```

D:\morn11>java Test

**finally**

**Exception in thread "main" java.lang.ArithmaticException: / by zero  
at Test.main(Test.java:4)**

**case 4:-**

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

**case 5:-**

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

```

        finally
        {
            System.out.println(10/0);
        }
        System.out.println("rest of the code");
    }
}
D:\>java Test
try
Exception in thread "main" java.lang.ArithmetricException: / by zero
at Test.main(Test.java:15)

```

**case 6:-it is possible to provide try-finally.**

```

class Test
{
    public static void main(String[] args)
    {
        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)
    {
        int a=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 Test
{
    void m1()
    {
        m2();
    }
    void m2()
    {
        m3();
    }
    void m3()
    {
        try{
            System.out.println(10/0);
        catch(ArithmetricException ae)
        {
            System.out.println(ae);           //java.lang.ArithmetricException: / by zero
            System.out.println(ae.getMessage()); // / by zero
            ae.printStackTrace();
            /* java.lang.ArithmetricException: / by zero
            at Test.m3(Test.java:10)
            at Test.m2(Test.java:6)
            at Test.m1(Test.java:3)
            at Test.main(Test.java:22)*/
        }
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
    }
};

```

**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())
    {
        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)
    {
    }
}
```

**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();
    }
}
```

```

    {
        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 User defined checked Exception:-***

*There are two approaches to create User defined checked Exception*

1. Default constructor approach
2. Parameterized constructor approach

**Creation of user defined 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:-**

```
class InvalidAgeException extends Exception
{
    //default constructor
};
```

Note: - in this example we are creating user definid 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**

```
import java.util.*;
class Test
```

```

{      static void validate(int age) throws InvalidAgeException
    {      if (age>18)
        {          System.out.println("welcome to the voting");
        }
    else
        {// InvalidAgeException 0-argument constructor executed (default constructor)
            throw new InvalidAgeException();
        }
    }
public static void main(String[] args) throws InvalidAgeException
{
    Scanner s=new Scanner(System.in);
    System.out.println("please enter age");
    int age=s.nextInt();
    validate(age);
}
}

D:\>java Test
please enter age
26
welcome to the voteing
D:\>java Test
please enter age
12
Exception in thread "main"
InvalidAgeException
at Test.validate(Test.java:12)
at Test.main(Test.java:20)

```

**creation of userdefined checked exception by using parametrized constructor approach:-**

step-1:- create the userdefined exception class.

**InvalidAgeException.java**

```

public class InvalidAgeException extends Exception
{
    InvalidAgeException(String str)
    {
        super(str); //calling super class constructor by passing our content as a input
    }
}

```

Step-2:- use user created Exception in our project.

**Project.java**

```

import java.util.*;
class Test
{
    static void validate(int age) throws InvalidAgeException
    {
        if (age>18)
        {
            System.out.println("welcome to the voting");
        }
    else
        {// InvalidAgeException parameterized constructor executed
            throw new InvalidAgeException("not eligible for vote");
        }
    }
public static void main(String[] args) throws InvalidAgeException
{
    Scanner s=new Scanner(System.in);
    System.out.println("please enter age");
}

```

```

        int age=s.nextInt();
        validate(age);
    }
}

```

**D:\>java Test**  
**please enter age**  
**26**  
**welcome to the voting**

**D:\>java Test**  
**please enter age**  
**12**  
**Exception in thread "main"**  
**InvalidAgeException: not eligible for vote**  
**at Test.validate(Test.java:12)**  
**at Test.main(Test.java:20)**

#### **Ex:- creation of user defined un-checked exception by using default constructor approach:-**

**Step-1:- create userdefined exception.**

##### **InvalidAgeException.java**

```
public class InvalidAgeException 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**

```
import java.util.*;
class Test
{
    static void validate(int age)
    {
        if (age>18)
        {
            System.out.println("welcome to the voting");
        }
        else
        {
            throw new InvalidAgeException("not eligible for vote");
        }
    }
    public static void main(String[] args)
    {
        Scanner s=new Scanner(System.in);
        System.out.println("please enter age");
        int age=s.nextInt();
        validate(age);
    }
}
```

#### **Ex:- creation of user defined un-checked exception by using parameterized constructor approach:-**

##### **InvalidAgeException.java**

```
public class InvalidAgeException extends RuntimeException
{
    InvalidAgeException(String str)
    {
        super(str);
    }
}
```

```

}

Project.java
import java.util.*;
class Test
{
    static void validate(int age)
    {
        if (age>18)
        {
            System.out.println("welcome to the voting");
        }
        else
        {
            throw new InvalidAgeException("not eligible for vote");
        }
    }
    public static void main(String[] args)
    {
        Scanner s=new Scanner(System.in);
        System.out.println("please enter age");
        int age=s.nextInt();
        validate(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 durgasoft");
    }
    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:-**

<b><i>Checked Exception</i></b>	<b><i>Description</i></b>
ClassNotFoundException	If the loaded class is not available
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	If the requested method is not available.
<b><i>UncheckedException</i></b>	<b><i>Description</i></b>
ArithmaticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.(out of range)
InputMismatchException	If we are giving input is not matched for storing input.
ClassCastException	If the conversion is Invalid.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.

## Multi Threading

### **Information about multithreading:-**

- 1) The earlier days the computer's memory is occupied only one program after completion of one program it is possible to execute another program is called uni programming.
- 2) Whenever one program execution is completed then only second program execution will be started such type of execution is called co operative execution, this execution we are having lot of disadvantages.
  - a. Most of the times memory will be wasted.
  - b. CPU utilization will be reduced because only program allow executing at a time.
  - c. The program queue is developed on the basis co operative execution

**To overcome above problem a new programming style will be introduced is called multiprogramming.**

- 1) Multiprogramming means executing the more than one program at a time.
- 2) All these programs are controlled by the CPU scheduler.
- 3) CPU scheduler will allocate a particular time period for each and every program.
- 4) Executing several programs simultaneously is called multiprogramming.
- 5) In multiprogramming a program can be entered in different states.
  - a. Ready state.
  - b. Running state.
  - C. Waiting state.
- 6) Multiprogramming mainly focuses on the number of programs.

### **Advantages of multiprogramming:-**

1. CPU utilization will be increased.
2. Execution speed will be increased and response time will be decreased.
3. CPU resources are not wasted.

### **Thread:-**

- 1) Thread is nothing but separate path of sequential execution.
- 2) The independent execution technical name is called thread.
- 3) Whenever different parts of the program executed simultaneously that each and every part is called thread.
- 4) The thread is light weight process because whenever we are creating thread it is not occupying the separate memory it uses the same memory. Whenever the memory is shared means it is not consuming more memory.
- 5) Executing more than one thread a time is called multithreading.

**Information about main Thread:-**

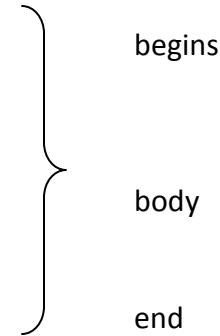
When a java program started one Thread is running immediately that thread is called main thread of your program.

1. It is used to create a new Thread(child Thread).
  2. It must be the last thread to finish the execution because it perform various actions.
- It is possible to get the current thread reference by using `currentThread()` method it is a static public method present in Thread class.

```
class CurrentThreadDemo
{
    public static void main(String[] arhgs)
    {
        Thread t=Thread.currentThread();
        System.out.println("current Thread-->" +t);
        //change the name of the thread
        t.setName("ratan");
        System.out.println("after name changed--> " +t);
    }
};
```

**Single threaded model:-**

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



In the above program only one thread is available is called main thread to know the name of the thread we have to execute the fallowing code.

**The main important application areas of the multithreading are**

1. Developing video games
2. Implementing multimedia graphics.
3. Developing animations

**There are two different ways to create a thread.**

- 1) You can extends the `java.lang.Thread` Class
- 2) You can implement the `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.**

```
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 a Thread object**

```
MyThread t=new MyThread();
```

**Step 3:- Starts the execution of a thread.**

```
t.start();
```

**Example :-**

```
class MyThread extends Thread          //defining a Thread
{
    public void run()   //business logic of thread
    {
        System.out.println("Rattaiah from durgasoft");
        System.out.println("body of the thread");
    }
};

class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
    }
}
```

**Flow of execution:-**

- 1) Whenever we are calling t.start() method the JVM search for the start() in the MyThread class since not present in the MyThread class so JVM will execute parent class(**Thread**) start() method is executed.
- 2) Whenever Thread class start method is executed then Thread Scheduler perform following actions.
  - a. Our thread is registered into Thread Scheduler then only new Thread is created.
  - b. The Thread class start() automatically calls run() to execute logics.

**Thread Scheduler:-**

- Thread scheduler is a part of the JVM. It decides which thread is executed first and which thread is executed next.
- Only one thread is executed at a time.
- We can't expect exact behavior of the thread scheduler it is JVM vendor dependent. So we can't expect output of the multithreaded examples we can say the possible outputs.
- Thread Scheduler mainly uses preemptive (or) time slicing to schedule the threads.

**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.

**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 durgasoft");
        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()
    {
        try
        {
            for (int i=0;i<5 ;i++ )
                {System.out.println("hi ratan");
            Thread.sleep(3000);    //it throws InterruptedException hence handle the Exception
            }
        }
        catch (InterruptedException ie)          {System.out.println(ie); }
    }
};

class Test
{
    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
        try
        {
            for (int i=0;i<10;i++ )
            {
                System.out.println("hi anu");
                Thread.sleep(2000);
            }
        }
        catch (InterruptedException ie)          {      System.out.println(ie); }
    }
};

```

**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.

**Internal Implementation of multiThreading:-**

```

interface Runnable
{
    public abstract void run();
}

class Thread implements Runnable
{
    public void run()
    {
        //empty implementation
    }
};

```

```

class MyThread extends Thread
{
    public void run()      //overriding run() to write business logic
    {
        for (int i=0;i<5 ;i++ )
        {
            System.out.println("user implementation");
        }
    }
}

```

**Difference between t.start() and t.run():-**

- In the case of t.start(), Thread class start() is executed a new thread will be created that is responsible for the execution of run() method.
- But in the case of t.run() method, no new thread will be created and the run() is executed like a normal method call by the main thread.

**Note :- Here we are not overriding the run() method so thread class run method is executed which is having empty implementation so we are not getting any output.**

```

class MyThread extends Thread
{
}
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
        for (int i=0;i<5;i++ )
        {
            System.out.println("main thread");
        }
    }
}

```

**Note :- If we are overriding start() method then JVM is executes override start() method at this situation we are not giving chance to the thread class start() hence n new thread will be created only one thread is available the name of that thread is main thread.**

```

class MyThread extends Thread
{
    Public void start()
    {
        System.out.println("override start method");
    }
}
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
        for (int i=0;i<5 ;i++ )
        {
            System.out.println("main thread");
        }
    }
}

```

**Different Threads are performing different tasks:-**

- 1) Particular task is performed by the number of threads here number of threads(t1,t2,t3) are executing same method (functionality).

- 2) In the above scenario for each and every thread one stack is created. Each and every method called by particular Thread the every entry stored in the particular thread stack.

```

class MyThread1 extends Thread
{
    public void run()
    {
        System.out.println("ratan task");
    }
}
class MyThread2 extends Thread
{
    public void run()
    {
        System.out.println("durga 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("durgasoft 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 has some name if may be default name provided by the jvm or customized name provided by the programmer.

The following methods are useful to set and get the name of a Thread.

**Public final String getName()**  
**Public final void setName(String name)**

#### **Example:-**

```
class MyThread extends Thread
{
class Test
{
    public static void main(String args[])
    {
        System.out.println(Thread.currentThread().getName());
        MyThread t=new MyThread();
        System.out.println(t.getName());
        Thread.currentThread().setName("meena");
        System.out.println(Thread.currentThread().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".

#### **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() and javanlang.Thread.isAlive() :-**

To know the thread whether the another thread is ended or not use isAlive() method.

And its return type is Boolean

If isAlive() return true Thread is running mode and if it returns false Thread execution completed.

Join() is used to stop the execution of the thread until completion of some other Thread task.

**if a t1 thread is executed t2.join() at that situation t1 must wait until completion of the t2 thread.**

1. Public final void join()throws InterruptedException
2. Public final void join(long ms) throws InterruptedException
3. Public final void join(long ms, int ns) throws InterruptedException

**Java.lang.Thread.getId():-**

getId() is used to generate id value for each and every thread.

**Public long getId()**

**Methods of Thread class:-**

```
class MyThread extends Thread
{
    public void run()
    {
        for (int i=0;i<10;i++)
        {
            Thread.sleep(2000);
            System.out.println("durgasoft task");
        }
    }
}
```

```

        }
    }
};

class ThreadDemo
{
    public static void main(String[] args) throws Exception
    {
        MyThread1 t1=new MyThread1();
        MyThread2 t2=new MyThread2();
        MyThread3 t3=new MyThread3();
        t1.start();
        t2.start();
        t3.start();//4-threads
        t1.join();
        System.out.println(t1.getName()); //thread-0
        System.out.println(t2.getName());
        t1.setName("sneha");
        System.out.println(t1.getName()); //sneha
        System.out.println(Thread.currentThread().getName()); //main
        Thread.currentThread().setName("poornima");
        System.out.println(Thread.currentThread().getName()); //poornima
        System.out.println(Thread.activeCount()); //number of Threads
        System.out.println(t1.isAlive());//to check thread Running or not
        System.out.println(t1.getId()); //to get id of Thread
        System.out.println(t2.getId());
        System.out.println(Thread.currentThread().getPriority());
        System.out.println(t1.getPriority());
        Thread.currentThread().setPriority(10);
        System.out.println(Thread.currentThread().getPriority());
        for (int i=0;i<5;i++)
        {
            Thread.sleep(5000);
            Thread.yield();
            System.out.println("main thread");
        }
    }
};

```

**Interrupted():-**

- ❖ A thread can interrupt another sleeping or waiting thread.
- ❖ For this Thread class defines 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();
    }
};

```

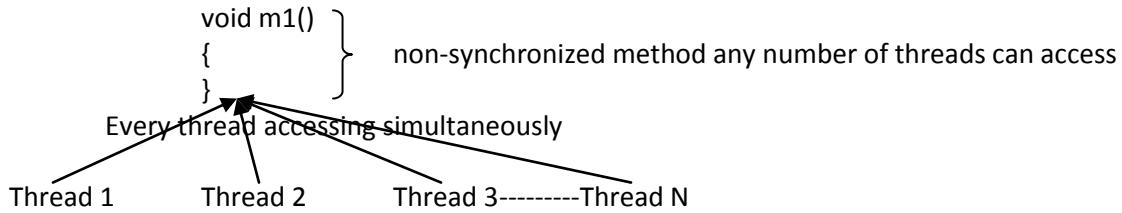
**NOTE:-** The interrupt() is working good whenever our thread enters into waiting state or sleeping state.

The interrupted call will be wasted if our thread doesn't enter into the waiting/sleeping state.

**Synchronized :-**

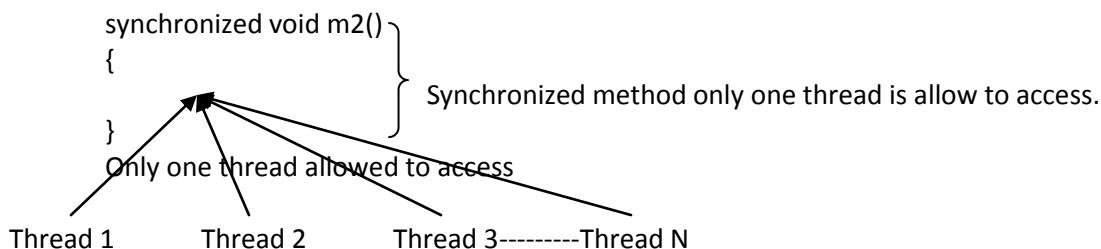
- Synchronized modifier is the modifier applicable for methods but not for classes and variables.
- If a method or a block declared as synchronized then at a time only one Thread is allowed to operate on the given object.
- The main advantage of synchronized modifier is we can resolve data inconsistency 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.

**Non-synchronized methods**



- 1) In the above case multiple threads are accessing the same methods hence we are getting data inconsistency problems. These methods are not thread safe methods.
- 2) But in this case multiple threads are executing so the performance of the application will be increased.

### Synchronized methods



- 1) In the above case only one thread is allowed to operate on particular method so the data inconsistency problems will be reduced.
- 2) Only one thread is allowed to access so the performance of the application will be reduced.
- 3) If we are using above approach there is no multithreading concept.

Hence it is not recommended to use the synchronized modifier in the multithreading programming.

### Example :-

```
class Test
{
    public static synchronized void x(String msg)      //only one thread is able to access
    {
        try{
            System.out.println(msg);
            Thread.sleep(4000);
            System.out.println(msg);
            Thread.sleep(4000);
        }
        catch(Exception e)
        {e.printStackTrace();}
    }
}
class MyThread1 extends Thread
{
    public void run( ) { Test.x("ratan"); };
}
class MyThread2 extends Thread
{
    public void run( ){Test.x("anu");}
}
class MyThread3 extends Thread
{
    public void run( ){Test.x("banu");}
}
class TestDemo
```

```
{     public static void main(String[] args) //main thread -1
    {
        MyThread1 t1 = new MyThread1();
        MyThread2 t2 = new MyThread2();
        MyThread3 t3 = new MyThread3();
        t1.start();      //2-Threads
        t2.start();      //3-Threads
        t3.start();      //4-Threads
    }
}
```

**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.

**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.

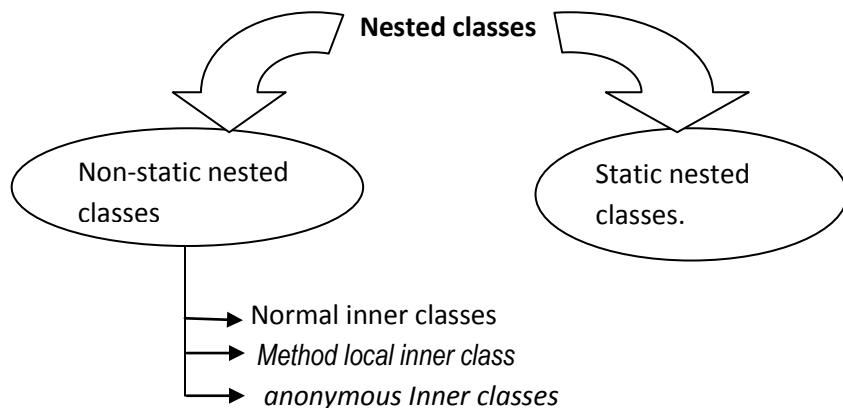
## Nested classes

- 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();                                o.m1();
        Outer.Inner i = o.new Inner();                      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 it

```
{
    void m1();
    void m2();
    .....
    void m100();
}
```

//adaptor class(contains empty implementation of interface methods)

class X implements it

```
{
    void m1(){}
    void m2(){}
    .....
    void m100(){}
};
```

//userdefined class extending adaptor class

class Test extends X

```
{
    //all methods are visible here
};
```

//using 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("Anonomous 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 ; } enum Week {
    public static final smantha;
    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 :-Semicolon optional**

```
enum Heroin
{
    samantha,tara,anu,ubanu
}
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");
}
class Test
{
    public static void main(String... ratan)
    {
        Heroin s=Heroin.samantha;
    }
};
```

**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[] arhss)  
    {        //accessing enum constants  
        Heroin[] h = Heroin.values();  
        for (Heroin h1 : h)  
        {  
            System.out.println(h1+"---"+h1.ordinal());  
        }  
    }//end main  
};//end class
```

### Collections(java.util)

- 1) Collection is a object that group multiple elements into a single unit and collections are used to store, retrieve, manipulate data.
- 2) Collection represent group of objects as a single entity.

#### **Limitations of array:-**

- 1) Array is indexed collection o fixed number of homogeneous data elements
- 2) Arrays can hold homogeneous data only
- 3) Once we created an array no chance of increasing o decreasing size of array

Ex:-

```
Student[ ] s=new Student[100];
S[0]=new Student();
S[1]=new Student();
S[2]=new Customer();-----compilation error
```

To overcome the above limitations of array the sun peoples are introduced collections concept

#### **Collections:-**

- 1) collection can hold both homogeneous data and heterogeneous data
- 2) collections are growable in nature
- 3) Memory wise collections are good. Recommended to use.
- 4) Performance wise collections are not recommended to use .

#### **Collections:-**

If we want to represent group of as a single entity then we should go for collection.

#### **In the collection framework we having 9 key interfaces:-**

1. Collection
2. List
3. Set
4. SortedSet
5. NavigableSet
6. Queue
7. Map
8. SortedMap
9. NavigableMap

The interface contains abstract method and for that interfaces object creation is not possible hence think about implementation classes of that interfaces.

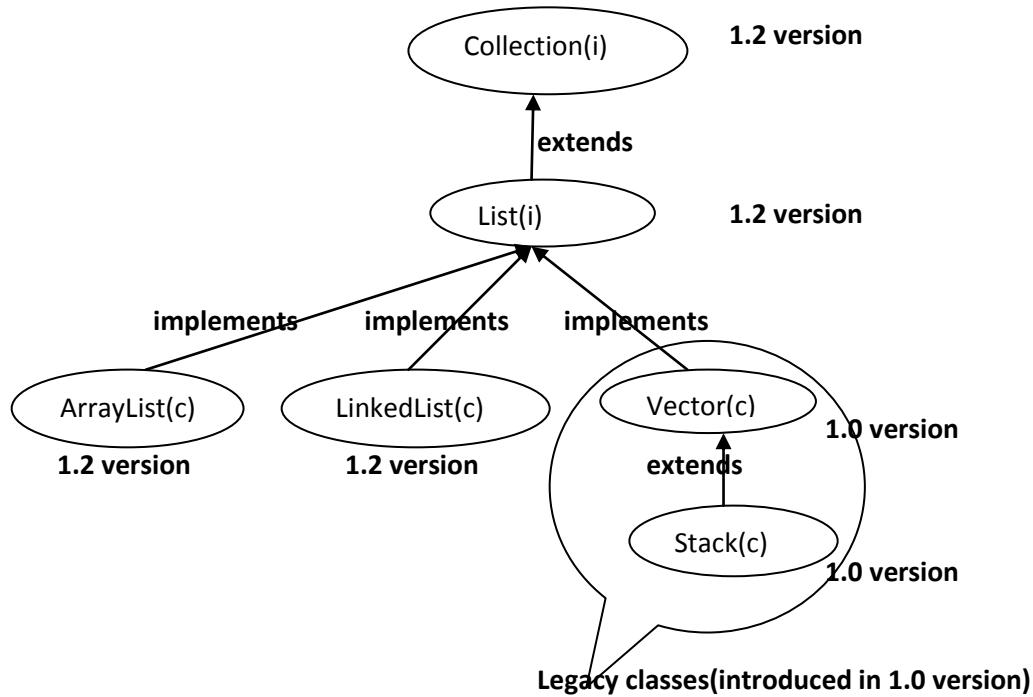
#### **Collection interface methods:-**

```
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();

```



I -----→Interface

c-----→class

#### Legacy class:-

The java classes that are introduced in 1.0 version are called legacy classes.

Ex :- Vector , Stack , HashTable.....etc

#### Deprecated methods:-

The methods of java not used in present projects those methods are called deprecated methods.

Ex :- compareTo() ....etc

#### In Ratan sir class collections are three steps:-

Step 1:- Represent group of objects as a single entity.

Step 2:- retrieve the objects of collections classes by using cursors.

Step 3:- represent [key,value] pairs.

#### characteristics of Collection classes:-

- 1) The collection classes are introduced in different Versions
- 2) Heterogeneous data allowed or not allowed.
- 3) Null insertion is possible or not possible.
- 4) Insertion order is preserved or not preserved.
- 5) Collection classes' methods are synchronized or non-synchronized.
- 6) Duplicate objects are allowed or not allowed.

- 7) Collections classes underlying data structures.**  
**8) Collections classes supported cursors.**

**Java.util.ArrayList:-**

the collection classes stores only objects but we are passing primitives these primitives are automatically converts into objects is called auto-boxing.

- 1) Introduced in 1.2 version.
- 2) Heterogeneous objects are allowed.
- 3) Null insertion is possible.
- 4) Insertion order is preserved.
- 5) ArrayList methods are non-synchronized methods.
- 6) Duplicate objects are allowed.
- 7) The underlying data structure is growable array.
- 8) Cursors :-Iterator , ListIterator

**Different methods of ArrayList:-**

<b>public boolean add(E);</b>	----> used to add objects in ArrayList.
<b>public void add(int, E);</b>	----> used to add the objects in ArrayList in specified index.
<b>public E remove(int);</b>	----> used to remove objects in specified index.
<b>public boolean remove(java.lang.Object);</b>	----> remove the specified object.
<b>public E set(int, E);</b>	----> used to replace the object in specified index.
<b>public void clear();</b>	---> remove all objects of collections classes.
<b>public E get(int);</b>	----> used to retrieve the object in specified index.
<b>public int size();</b>	----> used to find size of ArrayList.
<b>public boolean isEmpty();</b>	- ----> used to check whether ArrayList is empty or not.
<b>public boolean contains(java.lang.Object);</b>	---> to check whether the ArrayList contains specified Object or not.
<b>public int indexOf(java.lang.Object);</b>	---> used to check particular object index position in forward direction.
<b>public int lastIndexOf(java.lang.Object);</b>	---> used to check particular object index position in reverse direction.

**Example:- Normal version 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 removing elements ArrayList size " + al.size());
    }
}
```

```

        System.out.println(al);
    }
}

```

**All collection classes are having 2-versions:-**

- 1) Normal version.
- 2) Generic version.

**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

```

**Example :-retrieving different object form normal version of ArrayList**

```

import java.util.ArrayList;
class Emp
{
    //instance variable
    int eid;
    Emp(int eid) //local variable
    {
        this.eid=eid; } //conversion of local variable to instance variable
};
class Student
{
    //instance variable
    String sname;
    Student(String sname) //local variable
    {
        //conversion [passing local variables values to instance variables]
        this.sname = sname;
    }
};
class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        Emp e1 = new Emp(111); //creates object of Emp class
        Emp e2 = new Emp(222); //creates object of Student class
        Student s = new Student("anushka");
        //auto-boxing [automatic conversion of primitive to wrapper object]
        al.add(10); //integer object (auto boxing)
        al.add('a'); //character object (auto boxing)
        al.add(10.5); //double object (auto boxing)
        al.add("ratan"); //String object
        al.add(e1); //adding Emp object
        al.add(e2); //adding Emp object
        al.add(s); //adding student object
    }
}

```

```
for (Object o : al)
{
    //instanceof is checking type of the object and printing corresponding objects
    if (o instanceof Integer)
        System.out.println(o.toString());
    if (o instanceof Double)
        System.out.println(o.toString());
    if (o instanceof Character)
        System.out.println(o.toString());
    if (o instanceof String)
        System.out.println(o.toString());
    if (o instanceof Emp){
        Emp e = (Emp)o;
        System.out.println(e.eid);
    }
    if (o instanceof Student)
        System.out.println(s.sname);
}
}
```

**Normal version of ArrayList**

- 1) Normal version is able to hold any type of data(heterogeneous data).

```
Ex:- ArrayList al = new ArrayList();
      al.add(10);
      al.add('a');
      al.add(10.5);
      SOP(al);
```

- 2) In normal it is holding different types of data hence while accessing data must perform type casting.
- 3) If we are using normal version while compilation compiler generate warning message like 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**

- 1) Generic version is able to hold specified type of data.

```
Ex:- ArrayList<type-name> al = new
      ArrayList<type-name>();
      ArrayList<Integer> al = new
      ArrayList<Integer>();
      al.add(10);
      al.add(20);
      al.add(30);
      SOP(al);
```

- 2) While retrieving object from generic version typecasting is not required because it is holding only specified type of data.
- 3) If we are using generic version compiler wont 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);
    }
}
```

**we are able to retive objects from collection classes in two ways:-**

- 1) by using for-each loop.
- 2) by using cursors.

**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

### **Enumeration:-**

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.

### **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 fallowing 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:-**

<b><u>Characterstics</u></b>	<b><u>Enumeration</u></b>	<b><u>Iterator</u></b>	<b><u>ListIterator</u></b>
<b>1. Version</b>	1..0 version	1.2 version	1.2 version
<b>2.Legacy or not</b>	legacy	Not a legacy	Not a legacy
<b>3. How to get object</b>	By using elements() method	By using iterator() method	By using listIterator()method
<b>4. How many versions</b>	Normal version & generic version	Normal version & generic version	Normal version & generic version
<b>5. Methods</b>	2 methods	3 methods	9 methods
<b>6. Operations</b>	Read operations	Read & remove	Read & remove &update
<b>7.Class or interface</b>	interface	interface	interface
<b>8.Applicable to which type of classes</b>	legacy classes	all collection classes	only for List type of classes
<b>9.Cursor moment</b>	only forward direction	only forward direction.	both forward and backward directions.
<b>10.Return which object</b>	implementation class of Enumeration Interface	Implementation class of Iterator interface.	Implementation class of ListIterator interface.

**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$1
java.util.Vector$Itr
java.util.Vector$ListItr

```

**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]**

<b>add(E);</b>	----->to add the Object
<b>remove(java.lang.Object);</b>	----->to remove the object
<b>size();</b>	----->to find size
<b>isEmpty();</b>	----->returns true if empty otherwise false
<b>clear();</b>	----->to remove all objects
<b>addAll();</b>	----->to add one collection object into another collection
<b>removeAll();</b>	----->to remove all the elements of particular collection
<b>retainAll();</b>	----->to remove all elements except particular collections

**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,"durga");
        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 last Object
    l.removeLast(); //remove first 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.

#### 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 )

Ex:-

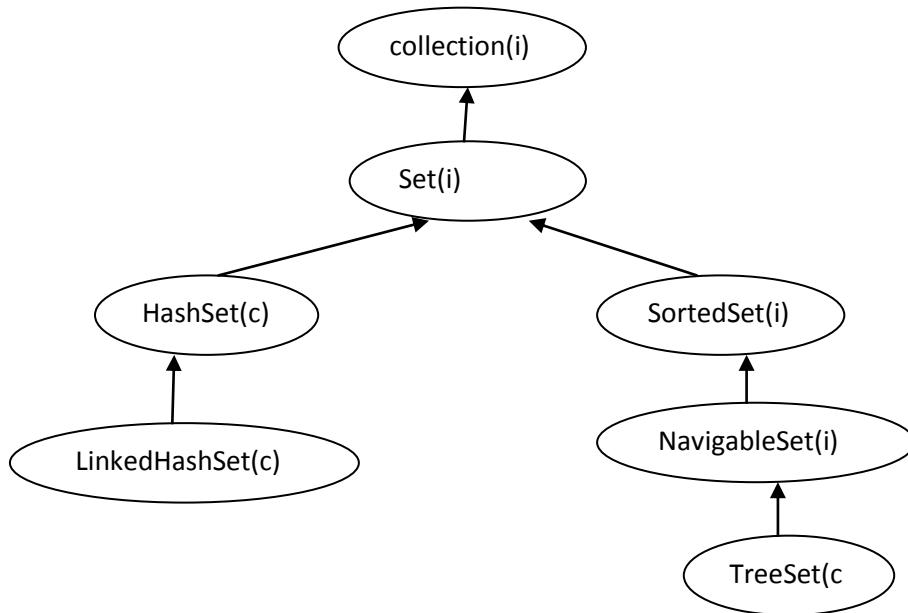
```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Stack s=new Stack();
        s.push("A");
        s.push(10);
        s.push("aaa");
        System.out.println(s);
        s.pop();
        System.out.println(s);
        System.out.println(s.search("A"));
    }
}

```

**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

**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 false(nothing).
3. Null insertion is possible.
4. Heterogeneous objects are allowed.
5. The underlying data structure is hashTable.
6. 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<String> h = new HashSet<String>();
        h.add("A");
        h.add("B");
        h.add("C");
        h.add("D");
        h.add("D");
        Iterator<String> itr = h.iterator();
        while (itr.hasNext())
        {
            String str = itr.next();
            System.out.println(str);
        }
    }
}
  
```

```

        }
    }
}
```

**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, Enumeration.

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

**TreeSet:-**

1. The underlying data Structure is BalancedTree.
2. Insertion order is not preserved it is based some sorting order.
3. Heterogeneous data is not allowed.
4. Duplicate objects are not allowed
5. Null insertion is possible only once.

**Ex:-**

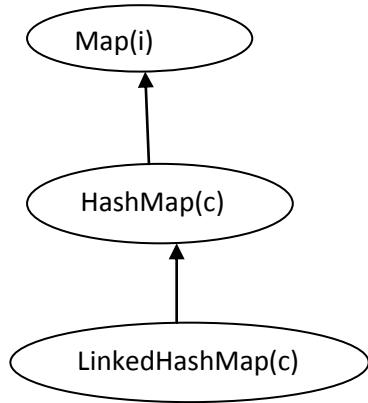
```

import java.util.*;
class Test
{
```

```

public static void main(String[] args)
{
    TreeSet t=new 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]
}
}

```

**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:-****HashMap:-**

- 
1. **Hetrogeneous data allowed.**
  2. **Underlying data Structure is HashTable.**
  3. **Duplicate keys are not allowed but values can be duplicated.**
  4. **Insertion order is not preserved.**

5. Null is allowed for key(only once)and allows for values any number of times.

6. Every method is non-synchronized so multiple Threads are operate at a time hence permanence is high.

7.interduced in 1.2 version

8 . cursor :- Iterator

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        HashMap h = new HashMap();
        h.put("ratan",111); //h.put(key,value);
        h.put("anu",111);
        h.put("banu",111);
        //public java/util/Set<K> keySet(); method syntax
        Set s1=h.keySet();           //used to get all keys
        System.out.println("all keys--->" + s1);
        // public java/util/Collection<V> values(); method syntax
        Collection c = h.values(); //used to get all values
        System.out.println("all values--->" + c);
        // public java/util/Set<java/util/Map$Entry<K, V>> entrySet(); method syntax
        Set ss = h.entrySet(); //it returns all entries nathing but [key,value]
        System.out.println("all entries--->" + ss);
        Iterator itr = ss.iterator();
        while (itr.hasNext())
        {
            Map.Entry m = (Map.Entry)itr.next(); //next() method retrun first entry to represent
            that entry do typeCasting
            System.out.println(m.getKey()+"----"+m.getValue()); //printing key and value
        }
    }
}
```

**LinkedHashMap:-**

1. Hetrogeneous data allowed.
2. Underlying data Structure is HashTable & linkedlist.
3. Duplicate keys are not allowed but values can be duplicated.
4. Insertion order is preserved.
5. Null is allowed for key(only once)and allows for values any number of times.
6. Every method is non-synchronized so multiple Threads are operate at a time hence permanence is high.

7.interduced in 1.4 version

8 . cursor :- Iterator

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        LinkedHashMap h = new LinkedHashMap();
        h.put("ratan",111); //h.put(key,value);
        h.put("anu",111);
        h.put("banu",111);
        //public java/util/Set<K> keySet(); method syntax
```

```

Set s1=h.keySet();           //used to get all keys
System.out.println("all keys--->"+s1);
// public java/util/Collection<V> values(); method syntax
Collection c = h.values(); //used to get all values
System.out.println("all values--->"+c);
// public java/util/Set<java/util/Map$Entry<K, V>> entrySet(); method syntax
Set ss = h.entrySet(); //it returns all entries nothing but [key,value]
System.out.println("all entries--->"+ss);
Iterator itr = ss.iterator();
while (itr.hasNext())
{
    Map.Entry m= (Map.Entry)itr.next(); //next() method retrun first entry to represent
that entry do typeCasting
    System.out.println(m.getKey()+"----"+m.getValue()); //printing key and value
}
};

1. It used to hold key value pairs
2. Underlying data Structure is HashTable.
3. Duplicate keys are not allowed but values can be duplicated.
4. Insertion order is not preserved.
5. Null is allowed for key (only once)and allows for values any number of times.
6. Every method is non-synchronized so multiple Threads are operate at a time hence
permanence is high.

```

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        HashMap h=new HashMap();
        h.put("samba",100);
        h.put("veeru",100);
        h.put("durga",100);
        System.out.println(h);
        Set s=h.keySet();
        System.out.println(s);
        Collection c=h.values();
        System.out.println(c);
        Set s1=h.entrySet();
        System.out.println(s1);

        Iterator itr=s1.iterator();
        while (itr.hasNext())
        {
            Map.Entry m1=(Map.Entry)itr.next();
            System.out.println(m1.getKey()+"----"+m1.getValue());
            if (m1.getKey().equals("samba"))
            {

```

```

        m1.setValue("gayan TeamLead");
    }
}
System.out.println(s1);
}
}

```

**HashTable:-**

1. It is a legacy class introduced in the 1.0 version.
2. Every method is synchronized hence only one thread is allow to access.
3. The performance of the application is low.
4. Null insertion is not possible if we are trying to insert null values we are getting NullPointerException.

Ex:-

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Hashtable h=new Hashtable();
        h.put("hyd",100);
        h.put("bang",200);
        h.put("pune",300);
        System.out.println(h);
        System.out.println(h.contains(300));//true
        System.out.println(h.containsValue(500));//false
        Collection c=h.values();
        System.out.println(c);
        Set c1=h.keySet();
        System.out.println(c1);
    }
}

```

**LinkedHashMap:-**

1. It used to hold key value pairs
2. Underlying data Structure is HashTable & LinkedList.
3. Duplicate keys are not allowed but values can be duplicated.
4. Insertion order is preserved.

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        LinkedHashMap h=new LinkedHashMap();
        h.put("sambha",100);
        h.put("veeru",100);
        h.put("durga",100);
        System.out.println(h);
        Set s=h.keySet();
        System.out.println(s);
    }
}

```

```

Collection c=h.values();
System.out.println(c);
Set s1=h.entrySet();
System.out.println(s1);
Iterator itr=s1.iterator();
while (itr.hasNext())
{
    Map.Entry m1=(Map.Entry)itr.next();
    System.out.println(m1.getKey()+"-----"+m1.getValue());
    if (m1.getKey().equals("sambha"))
    {
        m1.setValue("gayan TeamLead");
    }
}
System.out.println(s1);
}

```

**Java.util.Properties:-**

**abc.properties**

<b>username</b>	<b>=</b>	<b>root</b>
<b>password</b>	<b>=</b>	<b>ratan</b>
<b>durga</b>	<b>=</b>	<b>java</b>

**Test.java (uses abc.properties file)**

```

import java.util.*;
import java.io.*;
public class Test
{
    public static void main(String[] args) throws Exception
    {
        //locate jdbc properties file
        FileInputStream fis = new FileInputStream("abc.properties");
        //create properties file object(java.util)
        Properties p = new Properties();
        //load the jdbc properties into Properties class
        p.load(fis);
        //read values from java.util.Properties class Object
        String username = p.getProperty("username");
        String password = p.getProperty("password");
        String name = p.getProperty("durga");
        System.out.println("user name--->" + username);
        System.out.println("password--->" + password);
        System.out.println("language--->" + name);
    }
}

```

**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 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](http://www.google.com)

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.

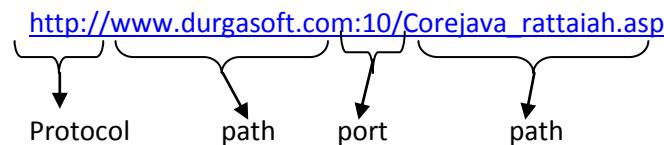
The IP Address is : local host/we are getting IP Address of the system

#### 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.durgasoft.com](http://www.durgasoft.com)
  - c. Port number of the particular application and it is optional(:10)
  - d. File name or directory name **Corejava\_rattaiah.asp**
- 4) To crate the object for URL we have to use the fallowing 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.durgasoft.com:10/index.html");
        System.out.println("protocal 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)

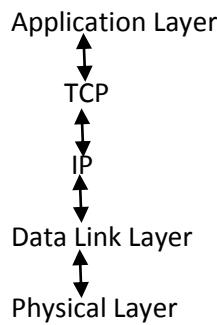
#### **Connection Oriented:-**

- 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 fallowing communication the java peoples are provided the fallowing 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 in to 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 is coming from TCP and prepare envelop 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 sent that 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 compare to the TCP/IP communication.
- 4) This protocol not sending the data inn 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 peoples are provided the fallowing 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 class present in the java.net package
- 5) It is acting as a communicator between the client and server.
- 6) Whenever if we want to send the data first we have to create a socket that is 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(Durgasoft,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**

1. Java.awt is a package it will provide very good environment to develop graphical user interface applications.
2. AWT means (Abstract Window Toolkit). AWT is used to prepare the components but it is not providing any life to that components means by using AWT it is possible to create a static components.
3. To provide the life to the static components we need to depends upon some other package is called java.awt.event package.
4. This application not providing very good look and feel hence the normal users facing problem with these types of applications.
5. 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 :-**

Component is an object which is displayed pictorially on the screen.

Ex:-

Button,Label,TextField.....etc

**Container:-**

Container is a GUI component, it is able to accommodate all other GUI components.

Ex:-

Frame,Applet.

**Event:-**

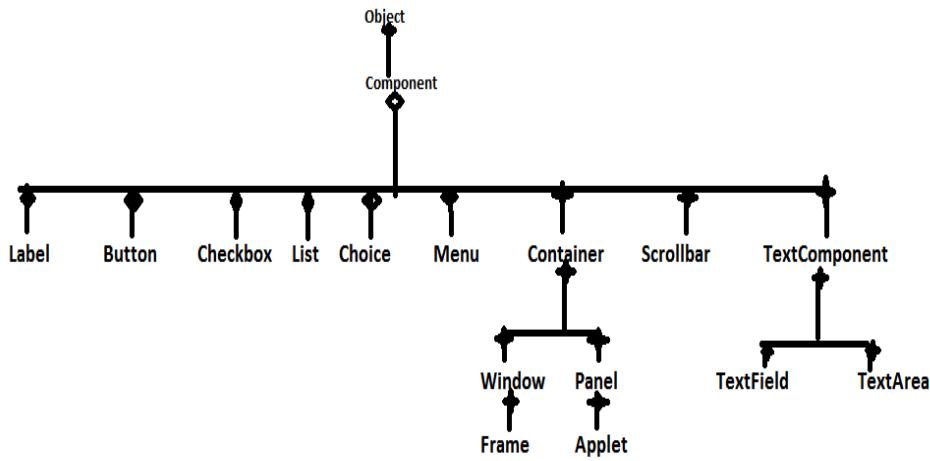
The event nothing but a action generated on the component or the change is made on the state of the object.

Ex:-

Button clicked, Checkboxchecked, Itemselected in the list, Scrollbar scrolled horizontal/vertically.

**Classes of AWT:-**

The classes present in the AWT package.

**Frame:-**

- 1) Frame is a class which is present in java.awt package.
- 2) Frame is a Basic component in AWT, because all the components displayed in a Frame.
- 3) We are displaying pictures on the Frame.
- 4) It is possible to display some text on the Frame.

Based on the above reasons the frame will become basic component in AWT.

**Constructors:-**

- \* create a Frame class object.  
Frame f=new Frame();
- \* create a Frame class object and pass file  
Frame f=new Frame("MyFrame");
- \* Take a subclass to the Frame and create object to the subclass.  
class MyFrame extends Frame  
MyFrame f=new MyFrame();

**Characteristics of the Frame:-**

- 1) When we create a Frame class object the Frame will be created automatically with the invisible mode. To provide visible mode to following method.

**public void setVisible(boolean b)**

where b==true means visible mode.

where b==false means invisible mode.

Ex: f.setVisible(true);

- 2) When we created a Frame the initial size of the Frame is :

0 pixel height

0 pixel width

So it is not visible to use.

- To provide particular size to the Frame we have to use following method.

**public void setSize(int width,int height)**

Ex: f.setSize(400,500);

- 3) To provide title to the Frame explicitly we have to use the following method

**public void setTitle(String Title)**

Ex: f.setTitle("MyFrame");

- 4) When we create a Frame, the default background color of the Frame is white. If you want to provide particular color to the Frame we have to use the following method.

**public void setBackground(color c)**

Ex: f.setBackground(Color.red);

\*\*\*\*\*CREATION OF FRMAE\*\*\*\*\*

```
import java.awt.*;
class Demo
{
    public static void main(String[] args)
    {
        //frame creation
        Frame f=new Frame();
        //set visibility
        f.setVisible(true);
        //set the size of the frame
        f.setSize(400,400);
        //set the background
        f.setBackground(Color.red);
        //set the title of the frame
        f.setTitle("myframe");
    }
}
```

```
};
```

**\*\*\*CREATION OF FRAME BY TAKING USER DEFINED CLASS\*\*\***

```
import java.awt.*;
class MyFrame extends Frame
{
    MyFrame()
    {
        setVisible(true);
        setSize(500,500);
        setTitle("myframe");
        setBackground(Color.red);
    }
}
class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};
```

**To display text on the screen:-**

1. If you want to display some textual message or some graphical shapes on the Frame then we have to override paint(), which is present in the Frame class.

```
public void paint(Graphics g)
```

2. To set a particular font to the text, we have to use Font class present in java.awt package

```
Font f=new Font(String type,int style,int size);
```

```
Ex: Font f= new Font("arial",Font.Bold,30);
```

Ex :-

```
import java.awt.*;
class Test extends Frame
{
    public static void main(String[] args)
    {
        Test t=new Test();
        t.setVisible(true);
        t.setSize(500,500);
        t.setTitle("myframe");
        t.setBackground(Color.red);
    }
    public void paint(Graphics g)
    {
        Font f=new Font("arial",Font.ITALIC,25);
```

```

        g.setFont(f);
        g.drawString("hi ratan how r u",100,100);
    }
}

```

**Note:-**

1. When we create a MyFrame class constructor,jvm executes MyFrame class constructor just before this JVM has to execute Frame class zero argument constructor.
2. In Frame class zero argument constructor repaint() method will be executed, it will access predefined Frame class paint() method. But as per the requirement overriding paint() method will be executed.
3. Therefore the paint() will be executed automatically at the time of Frame creation.

**Preparation of the components:-****Label:-**

- 1) Label is a constant text which is displayed along with a TextField or TextArea.
- 2) Label is a class which is present in java.awt package.
- 3) To display the label we have to add that label into the frame for that purpose we have to use add() method present in the Frame class.

**Constructor:-**

```

Label l=new Label();
Label l=new Label("user name");

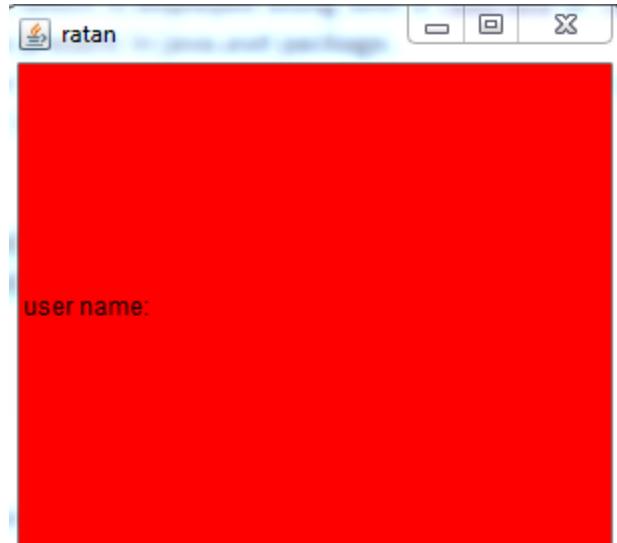
```

**Ex :-**

```

import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);
        Label l=new Label("user name:");
        f.add(l);
    }
}

```

**TextField:-**

- 1) TextField is an editable area.
  - 2) In TextField we are able to provide single line of text.
  - 3) Enter Button doesn't work on TextField. To add TextField into the Frame we have to use add() method.
1. To set Text to the textarea we have to use the following method.

- ```
t.setText("Durga");

2. To get the text form the TextArea we have to use fallowing method.
String s=t.getText();

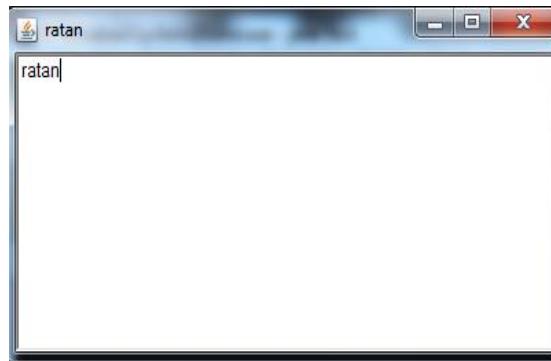
3. To append the text into the TextArea.
t.appendText("ratan");
```

**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:-**

- 1) TextArea is a class present in java.awt.package.
- 2) TextArea is a Editable Area. Enter button will work on TextArea.
- 3) To add the TextArea into the frame we have to use the add()

**Construction:-**

```
TextArea t=new TextArea();
TextArea t=new TextArea(int rows,int columns);
```

4. To set Text to the textarea we have to use the following method.
 

```
t.setText("Durga");
```
5. To get the text form the TextArea we have to use fallowing method.
 

```
String s=t.getText();
```

6. To append the text into the TextArea.

```
t.appendText("ratan");
```

```
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);
        f.setLayout(new FlowLayout());
        Label l=new Label("user name:");
        TextArea tx=new TextArea(4,10); //4 character height 10 character width
        tx.appendText("ratan");
        tx.setText("aruna");
        System.out.println(tx.getText());
        f.add(l);
        f.add(tx);
    }
}
```



#### **Choice:-**

- 1) Choice is a class present in java.awt package.
- 2) List is allows to select multiple items but choice is allow to select single Item.

#### **Constructor:-**

```
Choice ch=new Choice();
```

#### **Methods :-**

1. To add items to the choice we have to use following method.  

```
ch.add("HYD");
ch.add("Chennai");
ch.add("BANGALORE");
```
2. To remove item from the choice based on the string.  

```
ch.remove("HYD");
ch.remove("BANGALORE");
```

3. To remove the item based on the index position  
ch.remove(2);
4. To remove the all elements  
ch.removeAll();
5. To inset the data into the choice based on the particular position.  
ch.insert(2,"ratan");
6. To get selected item from the choice we have to use following method.  
String s=ch.getSelectedItem();
7. To get the selected item index number we have to use fallowing method  
int a=ch.getSelectedIndex();

**ex:-**

```
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);

        Choice ch=new Choice();
        ch.add("c");
        ch.add("cpp");
        ch.add("java");
        ch.add(".net");
        ch.remove(".net");
        ch.remove(0);
        ch.insert("ratan",0);
        f.add(ch);

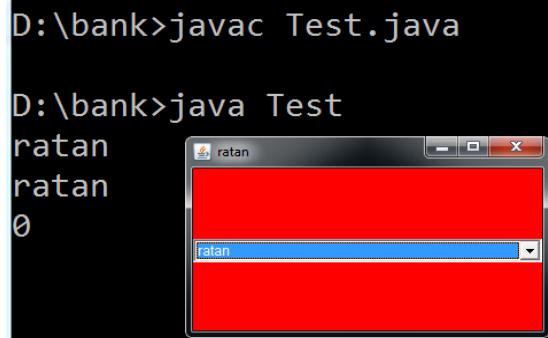
        System.out.println(ch.getItem(0));
        System.out.println(ch.getSelectedItem());
        System.out.println(ch.getSelectedIndex());
        //ch.removeAll();
    }
}
```

**List:-**

- 1) List is a class it is present in `java.awt.package`
- 2) List is providing list of options to select. Based on your requirement we can select any number of elements. To add the List to the frame we have to use `add()` method.

#### **CONSTRUCTOR:-**

- 1) List l=new List();



It will creates the list by default size is four elements. And it is allow selecting the only one item at a time.

2) List l=new List(3);

It will display the three items size and it is allow selecting the only single item.

3) List l=new List(5,true);

It will display the five items and it is allow selecting the multiple items.

#### Methods:-

1. To add the elements to the List we have to use following method.

```
l.add("c");
l.add("cpp");
l.add("java");
l.add("ratan",0);
```

2. To remove element from the List we have to use following method.

```
l.remove("c");
l.remove(2);
```

3. To get selected item from the List we have to use following method.

```
String x=l.getSelectedItem();
```

4. To get selected items from the List we have to use following method.

```
String[] x=s.get.SelectedItems()
```

Ex:-

```
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);
        f.setLayout(new FlowLayout());

        List l=new List(4,true);
        l.add("c");
        l.add("cpp");
        l.add("java");
        l.add(".net");
        l.add("ratan");
        l.add("arun",0);
        l.remove(0);
        f.add(l);
        System.out.println(l.getSelectedItem());
    }
}
```

**Checkbox:-**

- 1) Checkbox is a class present in java.awt package.
- 2) The user can select more than one checkbox at a time. To add the checkbox to the frame we have to use add() method.

**Constructor:-**

- 1) Checkbox cb1=new CheckBox();  
cb1.setLabel("BTECH");
- 2) Checkbox cb1=new CheckBox("MCA");
- 3) Checkbox cb3=new CheckBox("BSC",true);

**Methods:-**

1. To set a label to the CheckBox explicitly and to get label from the CheckBox we have to use the following method.  
cb.setLabel("BSC");
2. To get the label of the checkbox we have to use fallowing method.  
String str=cb.getLabel();
3. To get state of the CheckBox and to set state to the CheckBox we have to use following method.  
Boolean b=ch.getState();

Ex:-

```
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);
```

```

        Checkbox cb1=new Checkbox("BTECH",true);
        f.add(cb1);
        System.out.println(cb1.getLabel());
        System.out.println(cb1.getState());
    }
}

```

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

```
D:\bank>java Test
```

BTECH  
true

**RADIO BUTTON:-**

- 1) AWT does not provide any predefined support to create RadioButtons.
- 2) It is possible to select Only item is selected from group of item. To add the RadioButton to the frame we have to use add() method.

By using two classes we create Radio Button those are

- a)CheckBoxgroup
- b)CheckBox

step 1:- Create CheckBox group object.

```
CheckBoxGroup cg=new CheckBoxGroup();
```

step 2:- pass Checkbox object to the CheckboxGroup class then the radio buttons are created.

```
CheckBox cb1=new CheckBox("male",cg,false);
CheckBox cb2=new CheckBox("female",cg,false);
```

**Methods:-**

- 1) To set status and to get status we have to use setState() and getState() methods.

```
String str=cb.getState();
cb.setState();
```

- 2) To get Label and to set Label we have to use following methods.

```
String str=getLabel()
setLabel("female").
```

Ex:-

```
import java.awt.*;
class Test
{
    public static void main(String[] args)
```

```
{  
    Frame f=new Frame();  
    f.setVisible(true);  
    f.setTitle("ratan");  
    f.setBackground(Color.red);  
    f.setSize(400,500);  
    CheckboxGroup cg=new CheckboxGroup();  
    Checkbox cb1=new Checkbox("male",cg,true);  
    f.add(cb1);  
    System.out.println(cb1.getLabel());  
    System.out.println(cb1.getState());  
}  
}  
D:\bank>javac Test.java
```

```
D:\bank>java Test
```

```
male  
true
```



#### Layout Managers:-

```
import java.awt.*;  
class Test  
{  
    public static void main(String[] args)  
    {  
        Frame f=new Frame();  
        f.setVisible(true);  
        f.setTitle("ratan");  
        f.setBackground(Color.red);  
        f.setSize(400,500);  
        Label l1=new Label("user name:");  
        TextField tx1=new TextField();  
        Label l2=new Label("password:");  
        TextField tx2=new TextField();  
        Button b=new Button("login");  
        f.add(l1);  
        f.add(tx1);  
        f.add(l2);  
        f.add(tx1);
```

```

        f.add(b);
    }
}

```

### **Event delegation model:-**

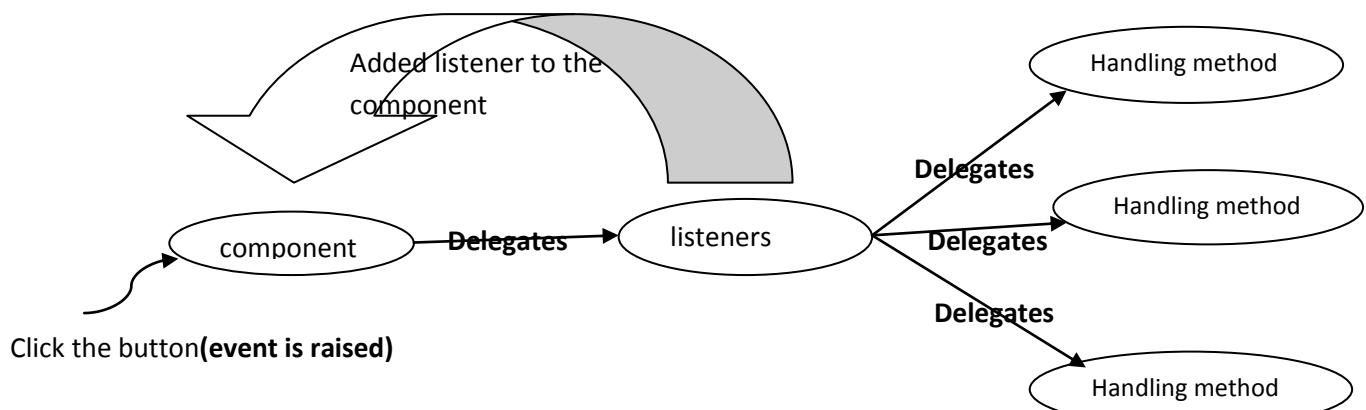
1. When we create a component the components visible on the screen but it is not possible to perform any action for example button.
2. Whenever we create a Frame it can be minimized and maximized and resized but it is not possible to close the Frame even if we click on Frame close Button.
3. The Frame 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. ScrollBar is a class present in the java.qwt.package
2. 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)<br>2.Public Void WindowActivated(WindowEvent e)<br>3.Public Void WindowDeactivated(WindowEvent e)<br>4.Public Void WindowClosing(WindowEvent e)<br>5.Public Void WindowClosed(WindowEvent e)<br>6.Public Void WindowIconified(WindowEvent e)<br>7.Public Void WindowDeiconified(WindowEvent e) |
| 2.Textfield | ActionEvent  | ActionListener  | 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<br>(AdjustementEvent e)                                                                                                                                                            |
| 11.Mouse     | MouseEvent      | MouseListener      | 1.Public Void MouseEntered(MouseEvent e)<br>2.Public Void MouseExited(MouseEvent e)<br>3.Public Void MousePressed(MouseEvent e)<br>4.Public Void MouseReleased(MouseEvent e)<br>5.Public Void MouseClicked(MouseEvent e) |
| 12.Keyboard  | KeyEvent        | KeyListener        | 1.Public Void KeyTyped(KeyEvent e)<br>2.Public Void KeyPressed(KeyEvent e)<br>3.Public Void KeyReleased(KeyEvent e)                                                                                                      |

\*\*\*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("durga")&&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();
    }
};
```

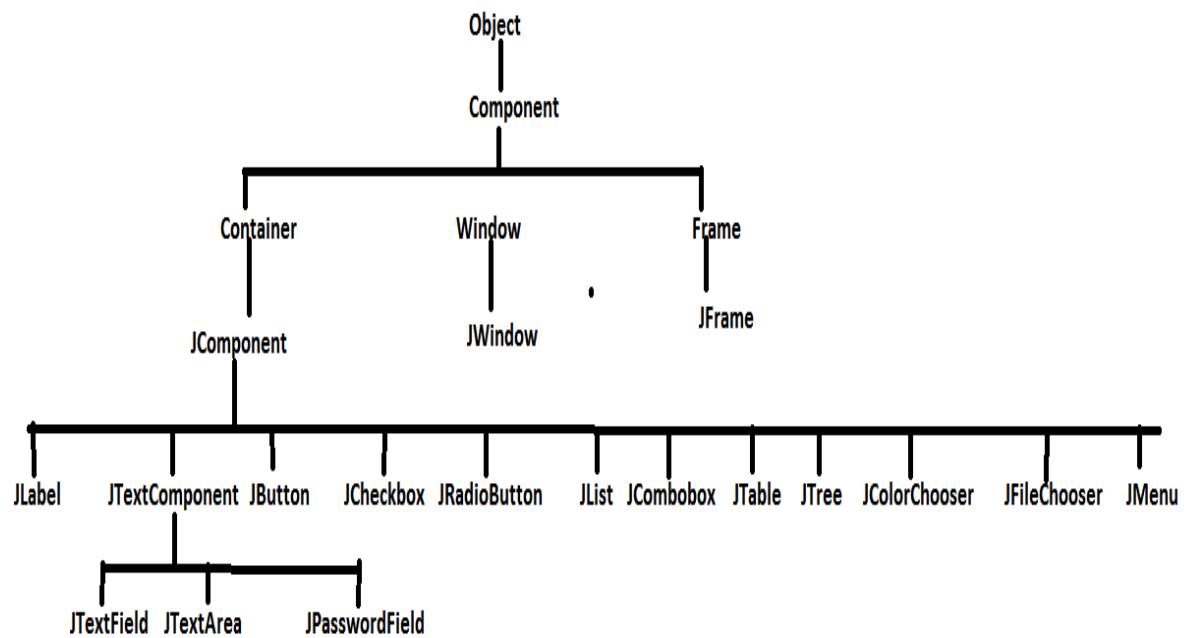
## SWINGS

1. Sun Micro Systems introduced AWT to prepare GUI applications.
2. awt components not satisfy the client requirement.
3. An alternative to AWT Netscape Communication has provided set of GUI components in the form of IFC(Internet Foundation Class).
4. IFC also provide less performance and it is not satisfy the client requirement.
5. In the above contest[sun+Netscape] combine and introduced common product to design GUI applications.

### Differences between awt and Swings:

1. AWT components are heavyweight component but swing components are light weight component.
2. AWT components consume more number of system resources Swings consume less number of system resources.
3. AWT components are platform dependent but Swings are platform independent.
4. AWT is provided less number of components where as swings provides more number of components.
5. AWT doesn't provide Tooltip Test support but swing components have provided Tooltip test support.
6. in awt for only window closing :      windowListener      windowAdaptor  
In case of swing use small piece of code.
  - i. f.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);
7. AWT will not follow MVC but swing follows MVC Model View Controller It is a design pattern to provide clear separation b/w controller part,model part,view part.
  - a. Controller is a normal java class it will provide controlling.
  - b. View part provides presentation
  - c. Model part provides required logic.
8. In case of AWT we will add the GUI components in the Frame directly but Swing we will add all GUI components to panes to accommodate GUI components.

### **Classes of swing:-**



*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(l1);           c.add(l6);           c.add(cb);           c.add(l7);
c.add(ta);          c.add(b);           

}

class SwingDemo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};

*****JCOLORCHOOSEN*****
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
class MyFrame extends JFrame implements ChangeListener
{
    JColorChooser cc;
    Container c;
    MyFrame()
    {
        this.setVisible(true);
        this.setSize(500,500);
        this.setTitle("SWING GUI COMPONENTS EXAMPLE");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        c=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();
    }
};
```

```
*****JFILECHOOSEN*****
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
class MyFrame extends JFrame implements ActionListener
{
    JFileChooser fc;
    Container c;
    JLabel l;
    JTextField tf;
    JButton b;
    MyFrame()
    {
        this.setVisible(true);
        this.setSize(500,500);
        this.setTitle("SWING GUI COMPONENTS EXAMPLE");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        c=getContentPane();
        l=new JLabel("Select File:");
        tf=new JTextField(25);
        b=new JButton("BROWSE");
        this.setLayout(new FlowLayout());
        b.addActionListener(this);
        c.add(l);           c.add(tf);           c.add(b);
    }
    public void actionPerformed(ActionEvent ae)
    {
        class FileChooserDemo extends JFrame implements ActionListener
        {
            FileChooserDemo()
            {
                Container c=getContentPane();
                this.setVisible(true);
                this.setSize(500,500);
                fc=new JFileChooser();
                fc.addActionListener(this);
                fc.setLayout(new FlowLayout());
                c.add(fc);
            }
            public void actionPerformed(ActionEvent ae)
            {
                File f=fc.getSelectedFile();
                String path=f.getAbsolutePath();
                tf.setText(path);
                this.setVisible(false);
            }
        }
        new FileChooserDemo();
    }
}
```

```
}

class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};

*****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);

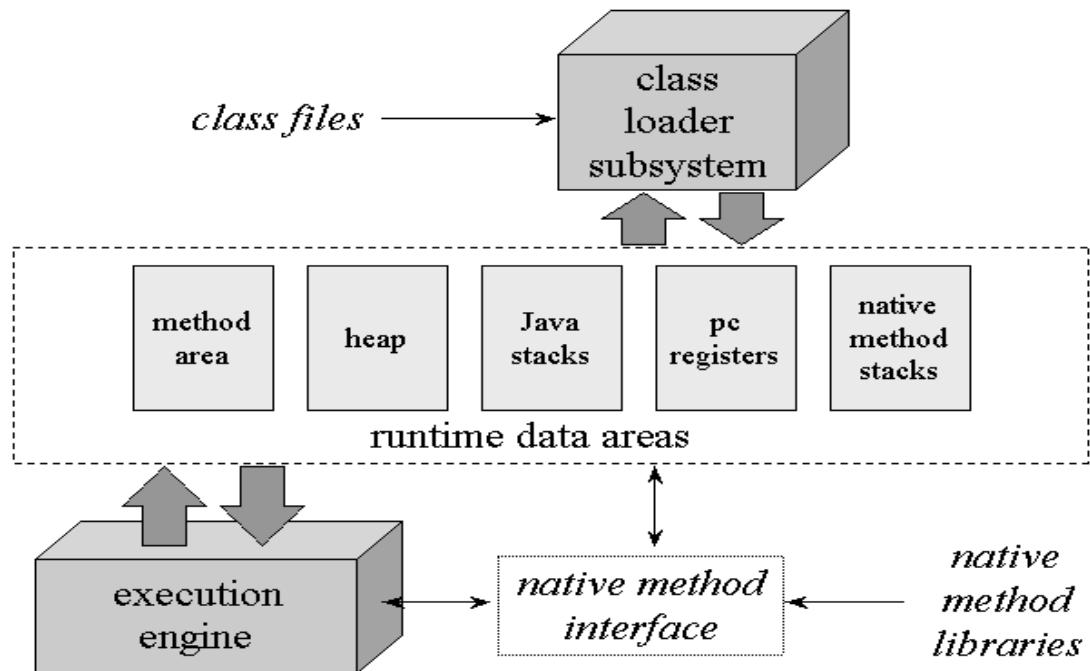
        g.drawString("Durga Software Solutions",100,200);
    }
};
```

**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("Durga 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>
```

### JVM Architecture:-



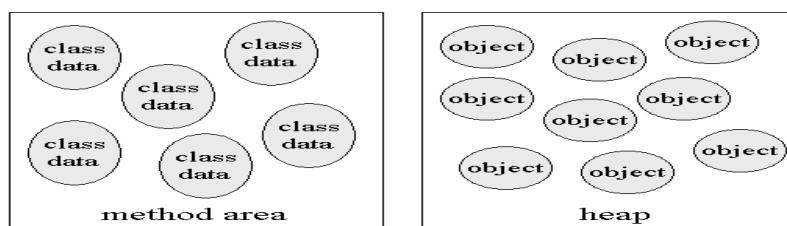
### Class loader subsystem:-

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.

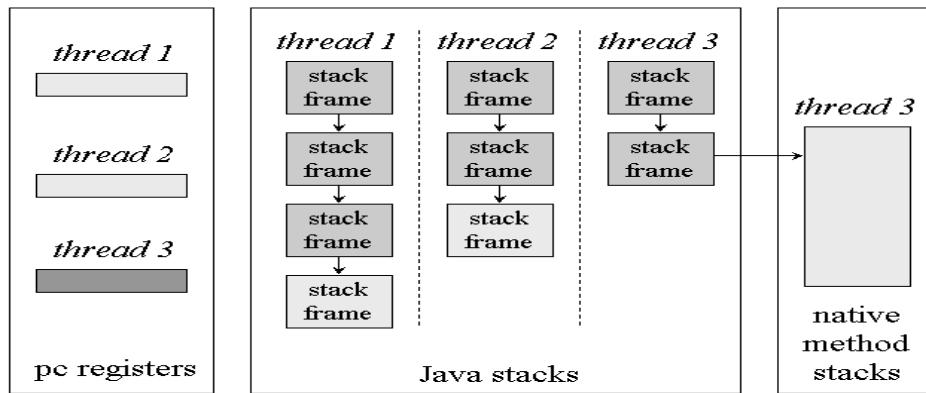
**Heap area:-**It is used to store the Objects.



**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:-**

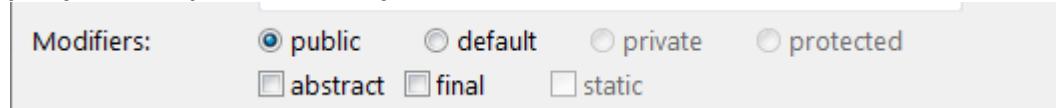
- In java no concept like “access specifiers and access modifiers” and only one concept is there modifiers concept.
- How many Modifiers in java means don't say 3 or 4 or 5 , in java 11 modifiers are there.
- The default modifier in java is “default”.
- The most restricted modifier in java is private (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.**

<u>modifier</u>	<u>classes</u>	<u>methods</u>	<u>variables</u>
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:-**

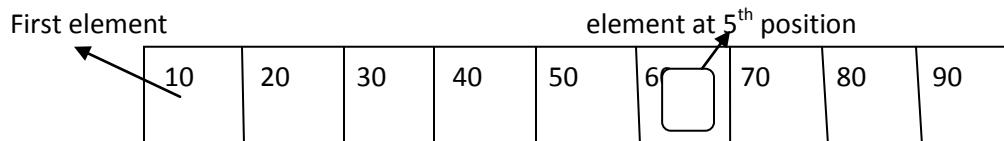
1. By using new operator.
2. by using clone() method.
3. without using new operator by using **String str="ratan"; [by using String content]**.
4. at the time of deserialization we are getting the data from file we are stored in object form.
5. By using factory method
  - a. Instance factory method
  - b. Static factory method
6. By using newInstance method. (used by servers).....etc

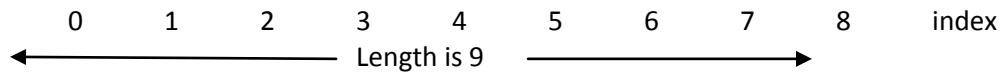
**Arrays:-**

- 1) Array is a final class inheritance is not possible.
- 2) Arrays are used to store the multiple numbers of elements of single type.
- 3) The length of the array is established at the time of array creation. After creation the length is fixed.
- 4) The items presented in the array are classed elements. Those elements can be accessed by index values. The index is begins from (0).

**Advantages of array:-**

- 1) Length of the code will be decreased
- 2) We can access the element present in the any location.
- 3) Readability of the code will be increased.



**Root structure:-**

```
java.lang.Object
|
+--java.lang.reflect.Array
```

public final class **Array** extends Object

**single dimensional array declaration:-**

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

**declaration & instantiation & initialization :-**

approach 1:- int a[]={10,20,30,40};  
 approach 2:- int[] a=new int[100];  
               a[0]=10;        a[1]=20;       .....     a[99]=40;

**Example :-**

**printing the array elements by using for loop**

```
class Test
{
    public static void main(String[] args)
    {
        int[] a={10,20,30,40};
        for (int i=0;i<a.length;i++)
        {
            System.out.println(a[i]);
        }
    }
}
```

**Example :-**

```
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)
        {

```

**Example :-**

**printing the array elements by using for-each loop(1.5 version)**

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

```

        System.out.println(a1);
    }
}
}

Ex:-when you print array values out of range value then JVM will print
ArrayIndexOutOfBoundsException
class Test
{
    public static void main(String[] args)
    {
        int[] a=new int[100];
        System.out.println(a.length);
        System.out.println(a[99]);
        byte[] bb=new byte[100];
        System.out.println(bb[100]);//ArrayIndexOutOfBoundsException
    }
}

```

**Example:-**

```

class Test
{
    public static void main(String[] args)
    {
        int[] a=new int[4];// allocates memory for 4 elements
        a[0]=10;
        a[1]=100;
        a[2]=1000;
        a[3]=10000;
        System.out.println(a.length);
        for (int i=0;i<a.length;i++)
        {
            System.out.println(a[i]);
        }
    }
}

```

**To get the class name of the array:-**

- *getClass() method is used to get the class.*
- *getName() method is used to print the name of the class.*
- *The above two methods are present in java.lang.Object class.*

```

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

```

**declaration of two dimensional array:-**

```

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

```

**Example :-**

```

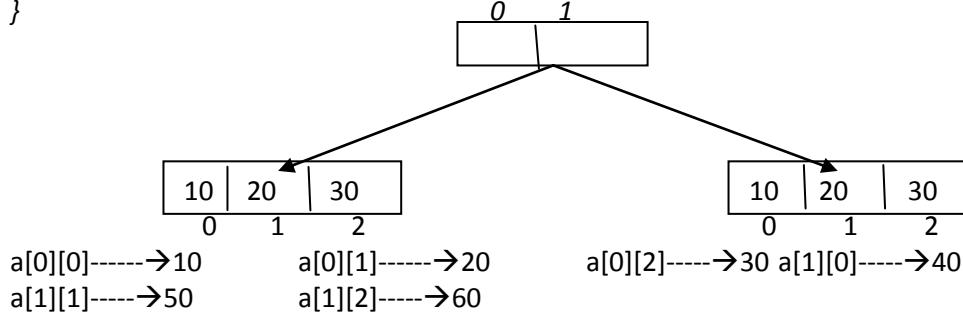
class Test
{
    public static void main(String[] args)
    {

```

```

{
    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
}

```



### 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 form 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 en 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) What is the purpose of variables in java?
- 35) How many types of variables in java and what are those variables?
- 36) What is the life time of static variables and where these variables are stored?
- 37) What is the life time of instance variables and where these variables are stored?
- 38) What is the life time of local variables and where these variables are stored?
- 39) For the static members when memory is allocated?
- 40) Where we declared local variables & instance variables & static variables
- 41) For the instance members when memory is allocated?
- 42) For the local variables when memory is allocated?
- 43) What is the difference between instance variables and static variables?
- 44) Can we declare instance variables inside the instance methods and static variables inside the static method?
- 45) 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?
- 46) What do you mean by method signature?
- 47) What do you mean by method implementation?
- 48) How many types of methods in java and how many types of areas in java?
- 49) What is the purpose of template method?
- 50) Can we have inner methods in java?
- 51) One method is able to call how many methods at time?
- 52) For java methods return type is mandatory or optional?
- 53) Who will create and destroy stack memory in java?
- 54) When we will get StackOverflowError?
- 55) Is it possible to declare return statement any statement of the method or any specific rule is there?
- 56) When we will get “variable might not have been initialized” error message?
- 57) What are the coding conventions of classes and interfaces?
- 58) What are the coding conventions of methods and variables?
- 59) What is the default package in java programming?
- 60) Platform dependent vs platform independent?
- 61) Is java a object oriented programming language?
- 62) By using which keyword we are creating object in java?
- 63) Object creation syntax contains how many parts?
- 64) How many types of constructors in java?
- 65) How one constructor is calling another constructor? One constructor is able to call how many constructors at time?
- 66) What do you mean by instantiation?
- 67) What is the difference between object instantiation and object initialization?
- 68) How many ways to create a object in java?
- 69) What is the purpose of this keyword?
- 70) What is the need of converting local variables to instance variables?
- 71) Is it possible to convert instance variables to local variables yes → how no → why?
- 72) When we will get compilation error like “call to this must be first statement in constructor”?
- 73) When we will get compilation error line “can not find symbol”?

- 74) What do u mean by operator overloading, is it java supporting operator overloading concept?
- 75) What is the purpose of scanner class and it is present in which package and introduced in which version?
- 76) What do you mean by constructor?
- 77) Who is generating default constructor and at what time?
- 78) What is the difference between named object and nameless object and write the syntax ?
- 79) What is object and what is relationship between class and Object?
- 80) Is it possible to execute default constructor and user defined constructor time?
- 81) If we are creating object by using new operator at that situation for every object creation how many constructors are executed?
- 82) What do you mean by object delegation?
- 83) What is the purpose of instance blocks when it will execute?
- 84) Inside class it is possible to declare how many instance blocks and what is syntax?
- 85) What is execution flow of method VS constructor Vs instance blocks Vs static blocks?
- 86) When instance blocks and static blocks are executed?
- 87) 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 curlybrases 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 difference between child class and parent class?
- 7) What is the root class for all java classes?
- 8) What is the execution process of constructors if two classes are there in inheritance relationship?
- 9) What is the execution process of instance blocks if two classes are there in inheritance relationship?
- 10) What is the execution process of static blocks if two classes are there in inheritance relationship?
- 11) Inside the constructor if we are not providing this() and super() keyword the compiler generated which type of super keyword?
- 12) How to call super class constructors?
- 13) Is it possible to use both super and this keyword inside the method?
- 14) Is it possible to use both super and this keyword inside the constructor?
- 15) If the child class and parent class contains same variable name that situation how to call parent class variable in child class?
- 16) One class able to extends how many classes at a time?
- 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) What is the root class for all java classes?
- 21) Which approach is recommended to create object either parent class object or child class object?
- 22) What is the purpose of instance of keyword in java?
- 23) What do you mean by polymorphism?
- 24) What do you mean by method overloading and method overriding?
- 25) How many types of overloading in java?
- 26) Is it possible to override variable in java?
- 27) What do you mean by constructor overloading?
- 28) What are rules must fallow while performing method overriding?
- 29) When we will get compilation error like “overridden method is final”?
- 30) What is the purpose of final modifier java?
- 31) Is it possible to override static methods yes→how no→why?
- 32) Parent class reference variable is able to hold child class object?
- 33) How many types of polymorphism in java?
- 34) What do you mean by dynamic method dispatch?
- 35) The applicable modifiers for local variables?
- 36) Is it all methods present in final class is always final and is it all variables present final class is always final?
- 37) 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?
- 38) Object class contains how many methods?
- 39) When we will get compilation error like “con not inherit from final parent”?
- 40) How many types of type casting in java?

- 41) What do you mean by co-varient return types?
- 42) What do u mean by method hiding?
- 43) What do you mean by abstraction?
- 44) How many types of classes in java?
- 45) Normal class is also known as ?
- 46) What is the difference between normal method and abstract method?
- 47) What is the difference between normal class and abstract class?
- 48) Is it possible to create a object for abstract class?
- 49) What do you mean by abstract variable?
- 50) Is it possible to override non-abstract method as a abstract method?
- 51) What is the purpose of abstract modifier in java?
- 52) How to prevent object creation in java?
- 53) What is the definition of abstract class?
- 54) In java is it abstract class reference variable is able to hold child class object?
- 55) What do you mean by encapsulation?
- 56) What do you mean by tightly encapsulated class?
- 57) What do you mean accessor method and mutator method ?
- 58) How many ways area there to set some values to class properties?
- 59) Can we overload method?
- 60) Can we inherit main method in child class?
- 61) In java main method is called by ?
- 62) The applicable modifiers on main method?
- 63) While declaring main method public static modifiers order mandatory or optional?
- 64) What is the argument of main method?
- 65) What is the return type of main method?
- 66) What are the mandatory modifiers for main method and optional modifiers of main method?
- 67) Why min method is static?
- 68) What do you by command line arguments?
- 69) Is it possible to pass command line arguments with space symbol no → good yes → how ?
- 70) What is the purpose of strictfp classes?
- 71) What is the purpose of strictfp modifier?
- 72) What is the purpose of native modifier?
- 73) What do you mean by native method and it also known as?
- 74) What do you mean by javaBean class?
- 75) The javabean class is also known as?
- 76) Applicable modifiers of local variables?
- 77) Applicable modifiers of instance variables?
- 78) Applicable modifiers of static variables?

## **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 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, database closing 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?

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 difference between length vs length()?
- 20) What is the default capacity of StringBuffer?
- 21) What do you mean by factory method?
- 22) Concat() method is a factory method or not?
- 23) What is the difference between heap memory and String constant pool memory?
- 24) String is a final class or not?
- 25) StringBuilder and StringTokenizer introduced in which versions?
- 26) What do you mean by legacy class & can you please give me one example of legacy class?
- 27) How to apply StringBuffer class methods on String class Object content?
- 28) When we use String & StringBuffer & String
- 29) What do you mean by cloning and use of cloning?
- 30) Who many types of cloning in java?
- 31) What do you mean by cloneable interface present in which package and what is the purpose?
- 32) What do you mean by marker interface and Cloneable is a marker interface or not?
- 33) 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 two create wrapper objects?
4. When we will get NumberFormatException?
5. How many constructors are present to create Character Wrapper class?
6. How many constructors are present to create Integer Wrapper class?
7. How many constructors are present 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 Autounboxing & introduced in which version?
14. Purpose of parseXXX() & xxxValue() method?
15. What are the Wrapper classes are direct child class of Object class?
16. What are the 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 framework 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 collections interfaces 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 is the difference between HashMap & LinkedHashMap?
- 36) What is the difference between comparable vs Comparator interface?
- 37) What is the difference between TreeSet andTtreeMap?
- 38) What is the difference between HashTable and Properties file key=value pairs?
- 39) What do you mean by properties file and what are the advantages of properties file?
- 40) Properties class present in which package?
- 41)

thank you  
thank you

thank you  
thank you

thank you  
thank you