

ratna5256@gmail.com

Corejava

Corejava



Bv
By

Mr. Ratan



Core java syllabus

1. Introduction 5-26

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

2. Flow control statements 27-38

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

3. Java class 39-76

*Variables
Methods
Constructors
Instance blocks
Static blocks*

4. Operator 77-81

*Unary Operator
Arithmetic Operator
shift Operator
Relational Operator
Bitwise Operator
Logical Operator
Ternary Operator
Assignment Operator.*

5. Oops 82-123

*Inheritance
Aggregation
Composition
Polymorphism
Abstraction
Encapsulation*

6. Packages 124-136

*Predefined packages
User defined packages
Importing packages
Project modules
Source file declaration
Normal import
Static import*

7. Modifiers

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

8. Interface 137- 153

*Interface declarations
Marker interface
Extends vs implements
Nested interface
Adaptor classes
Interface vs. inheritance
Cloning process
Serialization process
Deserialization process
Transient modifier.*

9. Garbage Collector 154-159

*Different ways to destroy object
Finalize() method
System gc() method
Runtime gc() method*

10. String manipulations 160-172

*String
StringBuffer
StringBuilder
 StringTokenizer
compreTo() vs equals()
length() vs length
method chaining
toString() implementation*

11. Wrapperclass 173-178

*Data types vs Wrapper classes
toString()
ParseXXX()
valueOf()
XXXValue().
Auto boxing vs Autounboxing*

12. Exception handling 179-207

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

13. java.io package 208-217

*File creation
Directory creation
Byte channel
Character channel
Writing and reading operations
Normal streams
Buffered streams
charArrayWriter*

14. Multithreading 218-241

*Thread info
Single Threaded model vs multithreaded model
Main Thread vs user Thread
Creation of user defined Thread
Life cycle stages of Thread
Thread naming
Thread priority
Thread synchronization
Inter Thread communication
Hook Thread
Daemon Thread
Difference between wait() notify() notifyAll()*

15. Nested classes 242-252

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

16. Lambda expressions 253-257**17. Annotations 258 - 261**

Advantages of annotations
Different annotations working
@Suppress Warnings
@Functional Interface
@Deprecated
@Override

20. Collection framework & Generics 275-323

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

21. INTERNATIONALIZATION (I18N) 324-334

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

18. Enumeration 262-267

Introduction
Advantages of enumeration
Values() vs Ordinal()
Enum vs enum
Diff between enum vs class

19. Arrays 268-274

Introduction
Declaration of Arrays
Object data & primitive data.

22. JVM architecture 335-340

What is JVM
Structure of the JVM
Components of JVM

23. Networking 341 - 344

Introduction
Socket and ServerSocket
URL info
Client-Server programming

24. AWT 345-367

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

25. Swings 368-376

Awt vs. swings
Advantages of swings
Different components of Swings(TextField , Checkbox.etc)
Event handling in Swings

26. Applet in java 377-382

JAVA introduction:

<i>Author</i>	:	<i>James Gosling</i>
<i>Vendor</i>	:	<i>Sun Micro System(which has since merged into Oracle Corporation)</i>
<i>Initial Name</i>	:	<i>OAK language</i>
<i>Present Name</i>	:	<i>java</i>
<i>Development starts</i>	:	<i>1991</i>
<i>Beta version</i>	:	<i>1995</i>
<i>Initial version</i>	:	<i>JDK 1.0 (January 23, 1996) (java development kit)</i>
<i>Present version</i>	:	<i>Java SE 13(September 25, 2018)</i>
<i>Stable version</i>	:	<i>Java SE 8 (March 18, 2014)</i>
<i>Type of the software</i>	:	<i>open source</i>
<i>Extensions</i>	:	<i>.java , .class</i>
<i>Operating System</i>	:	<i>multi Operating System</i>
<i>Implementation Lang</i>	:	<i>c, cpp.....</i>
<i>Symbol</i>	:	<i>coffee cup with saucer</i>
<i>SUN</i>	:	<i>Stanford Universally Network</i>
<i>Slogan/Motto</i>	:	<i>WORA(write once run anywhere)</i>
<i>Compilation</i>	:	<i>java compiler</i>
<i>Execution</i>	:	<i>JVM(java virtual machine)</i>

Importance of core java: According to the SUN 3 billion devices run on the java language only.

- ✓ Java is used to develop Desktop Applications such as MediaPlayer, Antivirus etc.
- ✓ Java is Used to Develop Web Applications such as durgasoft.com, irctc.co.in etc.
- ✓ Java is Used to Develop Enterprise Application such as Banking applications.
- ✓ Java is Used to Develop Mobile Applications.
- ✓ Java is Used to Develop Embedded System.
- ✓ Java is Used to Develop SmartCards.
- ✓ Java is Used to Develop Robotics.
- ✓ Java is used to Develop Gamesetc.

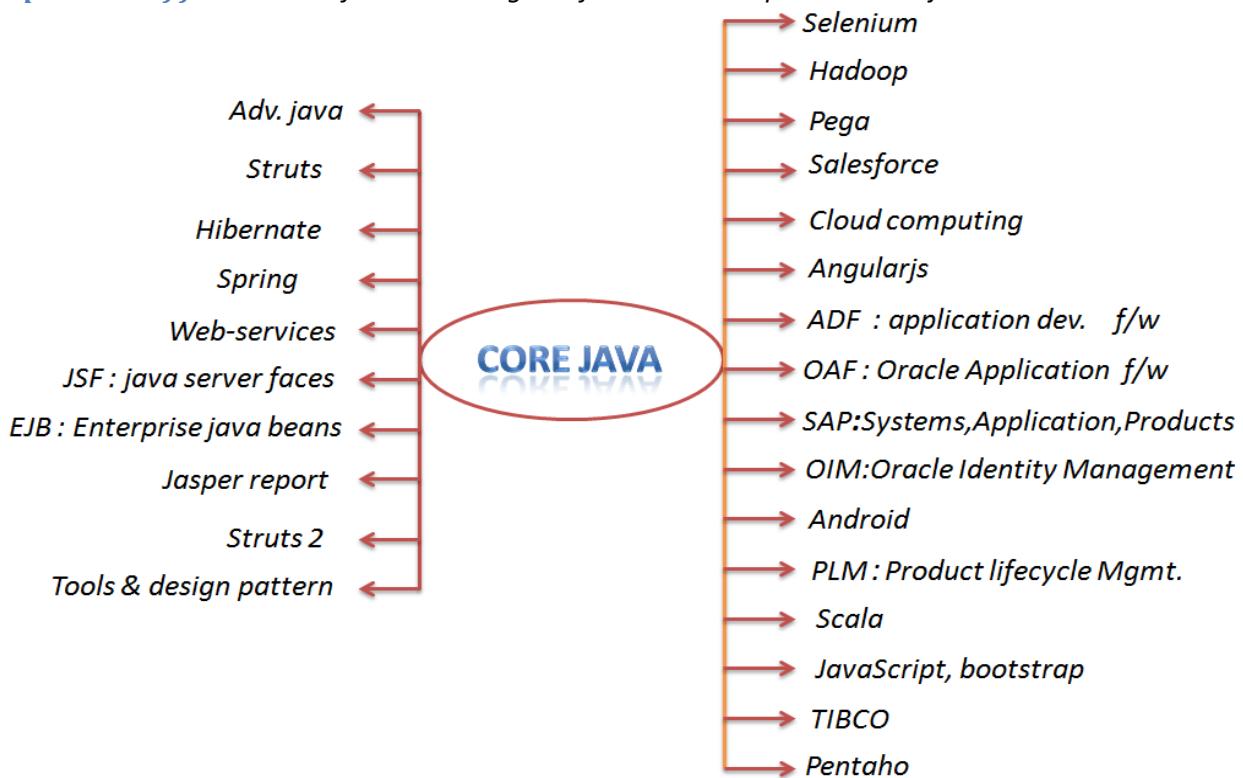
Java is a high level, robust, object-oriented and secure programming language.

Parts of the java:

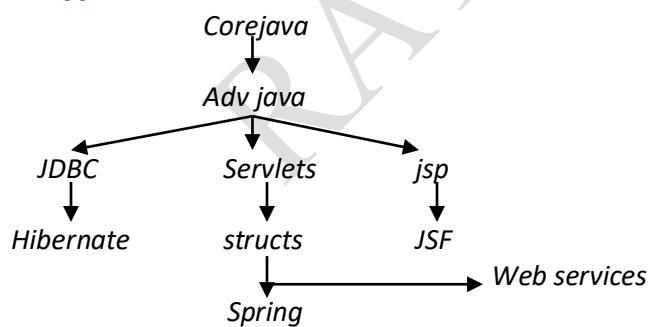
Core java & adv. java not official words, As per the **sun micro system** standard the java language is divided into three parts.

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

Importance of java: Most of the technologies & frame works depend on Core java.



Learning process of java:



Version Name	Code Name	Release Date
JDK 1.0	Oak	23 January 1996
JDK 1.1	(none)	19 February 1997
J2SE 1.2	Playground	4 December 1998
J2SE 1.3	Kestrel	8 May 2000
J2SE 1.4	Merlin	13 February 2002
J2SE 5.0	Tiger	29 September 2004
Java SE 6	Mustangs	11 December 2006
Java SE 7	Dolphins	28 July 2011
Java SE 8	(Not available)	18 March 2014
Java SE 9 (September 21, 2017)	Java SE 10 (March 20, 2018)	Java SE 11 (September 25, 2018)

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

byte

short

int

long

float

double

char

boolean

(8)

Flow-Control:

if

else

switch

case

default

break

for

while

do

continue

(10)

method-level:

void

return

(2)

Object-level:

new

this

super

instanceof

(4)

source-file:

class

extends

interface

implements

package

import

(6)

Exception handling:

try

catch

finally

throw

throws

(5)

1.5 version:

enum

assert

(2)

unused:

goto

const

(2)

Modifiers:

public

private

protected

abstract

final

static

strictfp

native

transient

volatile

synchronized

(11)

Predefined cons

true, false, null (3)

Reserved words (53):

Keywords (50) + constants (3) = Reserved Words (53)

Separators in java:

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

Java Comments:

- ✓ Comments are used to write the detailed description about application logics to understand the logics easily.
- ✓ The main objective of comments is application maintenance will become easy.
- ✓ Comments are non-executable code these are ignored during compilation.

There are 3 types of comments.

Single line Comments: Possible to write the description in single line.

Syntax: //description

Multi line Comments: To write the description in more than one line.

Starts with : /*
Ends with : */

Syntax: /* statement-1
statement-2
.....
statement-n
*/

Documentation Comments: Used to prepare API documents.

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

API(Application programming interface) :

- ✓ It contains detailed description about how to use java product.
- ✓ It is an interface between end-user & product.

Downloading Api document: it contains detailed description about how to use java product.

To download java api document use fallowing link : <http://docs.oracle.com/javase/8/docs>



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

Java SE Development Kit 8u65 Documentation

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

Accept License Agreement
 Decline License Agreement

Product / File Description	File Size	Download
Documentation	88.07 MB	jdk-8u65-docs-all.zip

Accept the license agreement and download the file.

Differences between C & CPP & JAVA:

C-language(1972)

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

Author: **Dennis Ritchie**

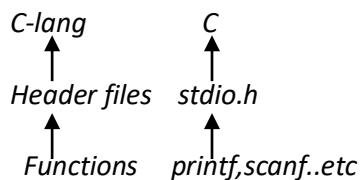
Implementation languages:
COBOL,FORTRAN,BCPL, B...

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

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

The header files contain predefined functions.

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



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

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

To print data use **printf()**

Cpp-language(1985)

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

Author : **Bjarne Stroustrup**

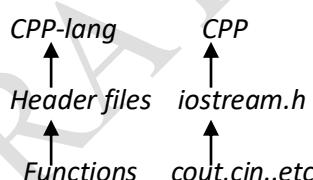
implementation languages:
c ,ada,ALGOL68.....

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

Ex:- **iostream.h**

The header files contains predefined functions.

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



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

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

To print data use **cout**

Java –language(1996)

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

Author : **James Gosling**

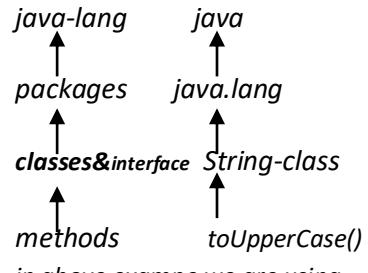
implementation languages
C,CPP,ObjectiveC...

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

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

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

Ex:- **String,System**



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

a) Import **java.lang.***; all lasses
b) Import **java.lang.System**; required
Import **java.lang.String**; classes
In above two approaches 2nd good

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

Packages contains:

- Classes*
- Interfaces*
- Enum*
- Annotations*
- Exceptions*
- Errors*

ASCII vs. Unicode:

ASCII	:	American Standard Code for Information Interchange
Unicode	:	Universal code

ASCII

Data representation in C-language
support only English
a-z A-Z 0-9 special symbols
char size : 1-byte
a=97 A=65

Unicode

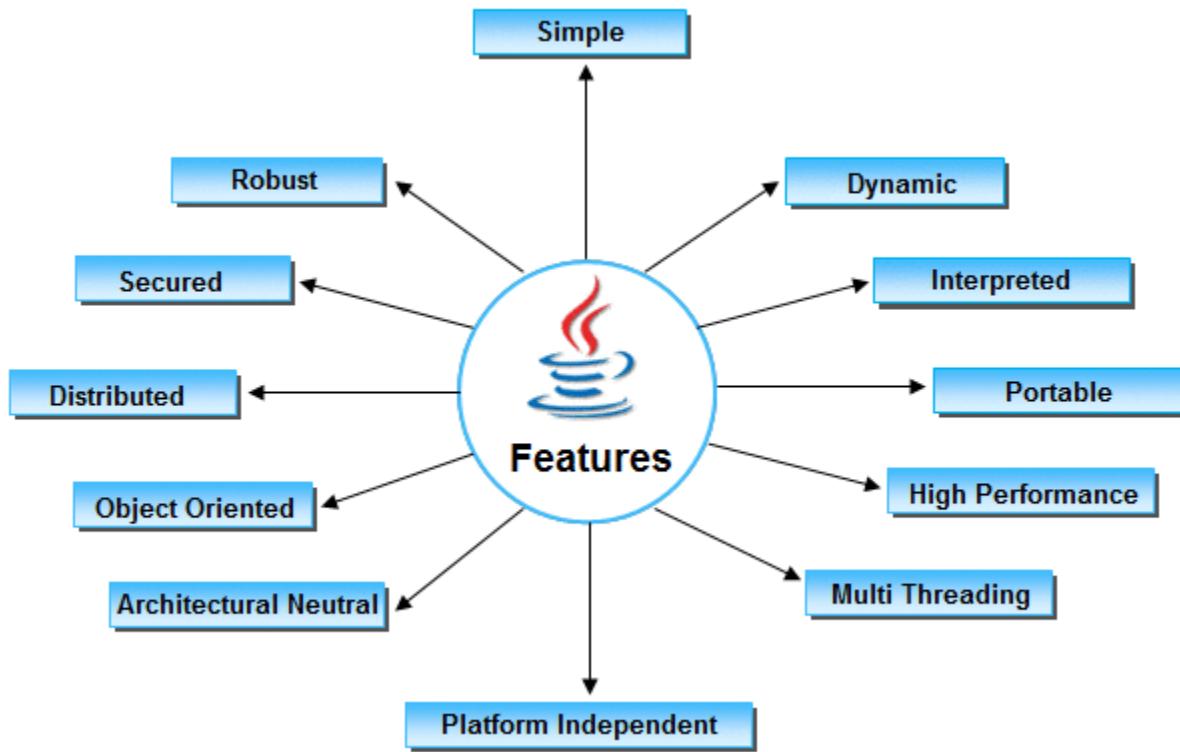
Data representation in java language.
Support all languages present in the world.
It support all languages characters.
char size 2-bytes
a=97 A=65

ASCII is the
American Standard Code
for Information Interchange

- Java uses two bytes to store characters as Unicode

Letter	ASCII Code	Letter	ASCII Code
A	01000001	N	01001110
B	01000010	O	01001111
C	01000011	P	01010000
D	01000100	Q	01010001
E	01000101	R	01010010
F	01000110	S	01010011
G	01000111	T	01010100
H	01001000	U	01010101
I	01001001	V	01010110
J	01001010	W	01010111
K	01001011	X	01011000
L	01001100	Y	01011001
M	01001101	Z	01011010

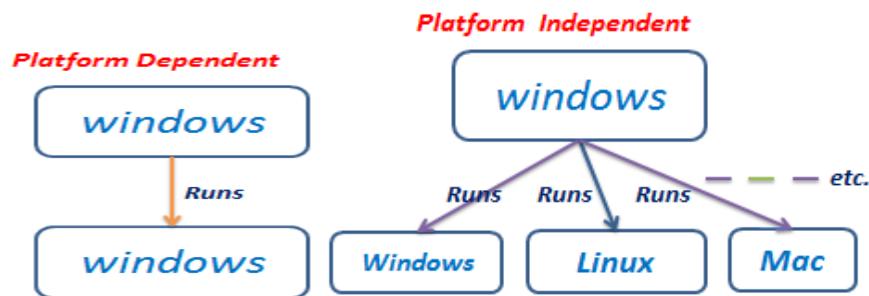
Character	ASCII code	Binary code
null character	0	00000000
a	97	1100001
b	98	1100010
c	99	1100011
A	65	1000001
B	66	1000010
C	67	1000011
%	37	0100101
+	43	0101011
0	48	0110000
1	49	0110001

JAVA Features : (Buzz words)**Simple**

- ✓ Java technology has eliminated all the difficult and confusion oriented concepts like pointers, multiple inheritance in the java language.
- ✓ Java uses c, cpp syntaxes mainly hence who knows C,CPP for that java is simple language.

Platform Independent

- ✓ Once we develop the application by using any one operating system(windows) that application runs only on same operating system is called platform dependency.
ex : C,CPP
- ✓ Once we develop the application by using any one operating system(windows) that application runs on in all operating system is called platform independency.
ex : java, python



Architectural Neutral

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

Portable

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

Object Oriented

Java is object oriented because it is representing total data of the class in the form of object.

Oops are methodologies to design the application, it's support all oops features like,

Class
object
Inheritance
Polymorphism,
Encapsulation
Abstraction

Robust

Robust simply means strong. Java is robust because:

In java dynamic memory management

Strong in exception handling , type checking mechanism.

Automatic memory management in java with the help of garbage collector.

The above all the points makes java robust.

Secure

To provide security Java provides one component inside JVM called Security Manager.

To provide security for the Java applications **java.security** package.

Dynamic : Java is dynamic language at runtime the memory is allocated.

Distributed : By using java it is possible to develop distributed applications by using RMI,EJB...etc

Interpreted : JVM mostly uses interpreter to convert byte code to machine dependent code.

High Performance

Java support features like Robust, Security, Platform Independent, Dynamic and so on then that technology is high performance.

Multithreaded

Executing more than one thread simultaneously is called multithreading.

Main advantage of multithreading is used to develop multimedia, gaming, web application.

Important overview of technologies & frameworks & servers & database:



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

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

2. Web applications:

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

3. Enterprise applications:-

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

4. Distributed applications:-

Software that executes on two or more computers in a network. In a client-server environment. Application logic is divided into components according to function.
ex : aircraft control systems, industrial control systems, network applications...etc

5. Mobile applications:-

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

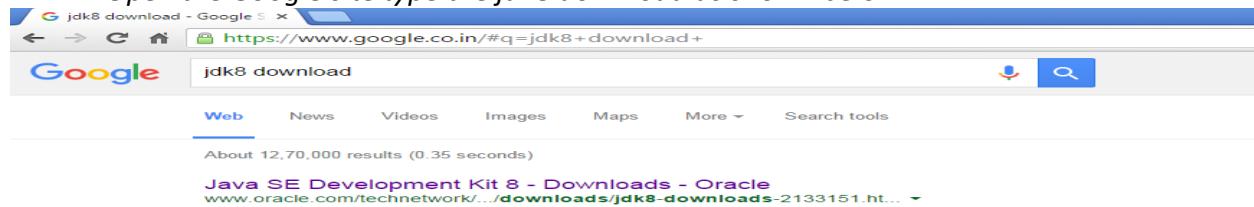
Install the software and set the path:

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

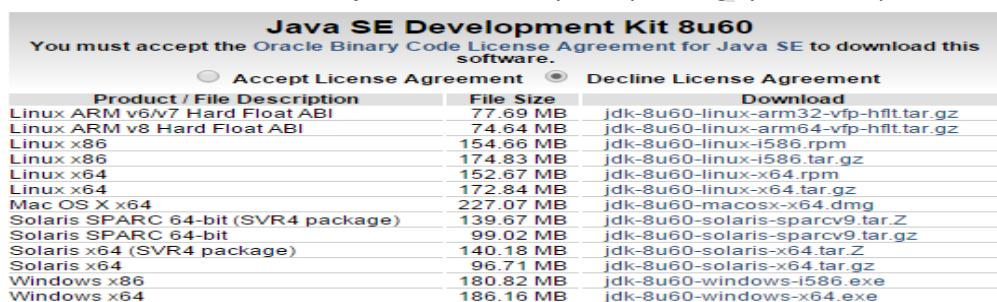
Step 1: Download the software:-

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

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



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



For 32-bit operating system please click on Windows x86

For 64-bit operating system please click on Windowsx64

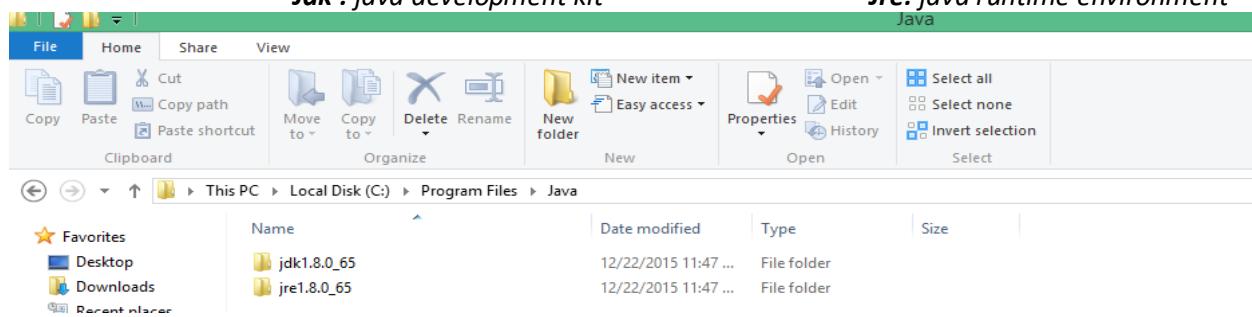
Step 2: Install the java software

- ✓ Install the java software just like media players in your machine. Next---next---next---finish
- ✓ After installing the software the installation java folder is available in the following location by default. (But it is possible to change the location at the time of installation).

Local Disk c: --->program Files--->java

Jdk : java development kit

Jre: java runtime environment



Step 3: set the path to run the application from any disk.

In C language we can run the application only form turbo-C folder.

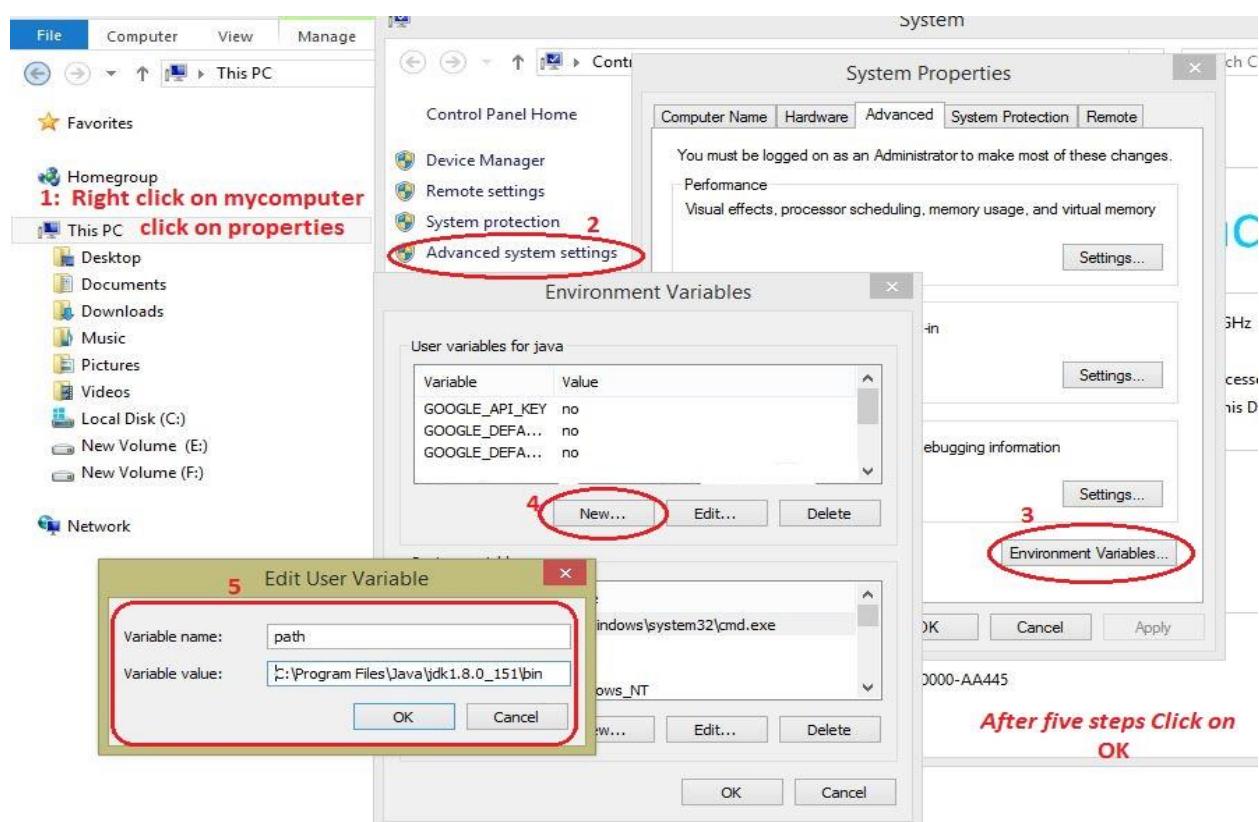
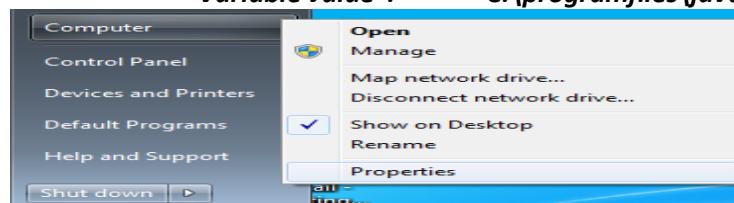
But in java it is possible to run the java files in any local disk (D: , E: , F: ...etc.) to achieve this when we have to set the path.

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

Variables --User variables--->new---->

variable name : path

Variable value : C:\programfiles\java\jdk1.8.0_11\bin;



After path setting open the command prompt type javac command then you will get list of commands then decide java is working properly your system.

```
C:\Users\welcome>javac
Usage: javac <options> <source files>
where possible options include:
  -g                         Generate all debugging info
  -g:none                     Generate no debugging info
  -g:{lines,vars,source}       Generate only some debugging info
  -nowarn                     Generate no warnings
```

Java coding conventions:

Camel Case

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

Classes	String	StringBuffer	InputStreamReaderetc
Interfaces	Serializable	Cloneable	RandomAccess ...etc	
Enum	Enum ,	ElementType ,	RetentionPolicy ...etc	
Annotation	Override ,	FunctionalInterface ...etc		
Exceptions	ArithmaticException			
Error	StackOverflowError			

Mixed case

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

Methods	post()	charAt()	toUpperCase()
Variables	out	in	pageContext ...etc

Lower Case : Every character should be lower case

Package	java.lang	java.util	java.io	...etc
Keywords	try	, if, break, continue	etc

Upper case : Every Character should be upper case

Constants: MAX_PRIORITY MIN_PRIORITY NORM_PRIORITY

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

Different types of languages:

- 1) Scripting languages : JS,php,python, perl ,vbscript
- 2) Procedural languages : BCPL , Pascal, Fortran , COBOL
- 3) Structured Programming languages : ALGOL, Pascal, Pl/I, C , Ada
- 4) Object-oriented programming : java
- 5) Object based language : Java Script
- 6) Functional programming language. : Scala ,python

Editors vs. IDE :

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

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

IDE: (Integrated development Environment)

IDE is providing very good environment to develop the application.

ex: Eclipse, MyEclipse, Netbeans, JDeveloper....etc

In IDE the code is automatically generated like,

- ✓ Automatic compilation.
- ✓ Automatic classes import.
- ✓ It shows all the predefined methods of classes.
- ✓ Automatically generate try catch blocks and throws.....

Note: Initial 20-days classes use the editor to develop the application, after that use the IDE to design to develop the application. (Remember projects we are using only IDE but not editor).

Editplus Download : <https://www.editplus.com/download.html>

The screenshot shows the 'Download' section of the Editplus website. It includes a note about the software being shareware, supported operating systems (Windows 7/8/8.1/10), and download links for the 'EditPlus 5.1 Evaluation Version' (1.97 MB and 64-bit versions). A watermark for 'DurgaSoft' is visible across the page.

IDE download :

<https://www.eclipse.org/downloads/packages/release/2018-12/r/eclipse-ide-java-developers>

The screenshot shows the 'Eclipse IDE for Java EE Developers' download page. It features a sidebar with 'RELEASES' and 'Neon Packages' highlighted. The main content area displays the 'Eclipse IDE for Java EE Developers' package description, which includes a brief overview, a list of included tools (Data Tools Platform, Eclipse Git Team Provider, Eclipse Java Development Tools, Eclipse Java EE Developer Tools), and download links for various platforms (Windows 32-bit, Windows 64-bit, Mac OS X (Cocoa) 64-bit, Linux 32-bit, Linux 64-bit). A note indicates 207,399 downloads.

Steps to Design a First Application:

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

Step 2: Write a program in edit plus.

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

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

In above example we are using two predefined classes(**String , System**) & these are present in **java.lang** package hence must import that package by using import statement.

There are two approaches to import the classes in java,

- 1) Import all classes of particular package.
 - a. Import **java.lang.*;**
- 2) Importing application required classes.
 - a. Import **java.lang.System;**
 - b. Import **java.lang.String;**

Step2: save the application.

- ✓ Save the java application by using (**.java**) extension.
- ✓ While saving the application must follow two rules
 - If the source file contains public class then must save the application by using **public class-name (publicClassName.java)**. Otherwise compiler generate error message.
 - if the source file does not contain public class then save the application by using any name(**anyName.java**) like **A.java , Ratan.java, Anu.java**etc.

Note: The source file is allows declaring only one public class, if we are trying to declare more than one public class then compiler generate error message.

Application location:

D:	(any disk)
-->java5pm	(any folder)
-->Ratan.java	(your file name)

Step-4: Compilation process.

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

Syntax: Javac filename
 Javac Test.java

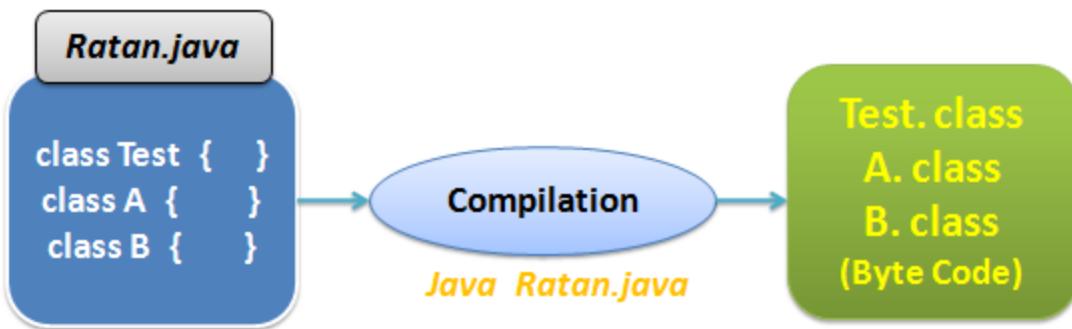
Open the command prompt then move to your application location:-

C:\Users\hp>	initial cursor location
C:\Users\hp>d:	move to local disk D
D:>cd java5pm	changing directory to ratan
D:\ratan>javac Ratan.java	compilation process

Compiler responsibilities:

- ✓ Compiler check the syntax errors, if the application contains syntax errors then compiler will generate error message in the form of compilation error.
- ✓ If the application does not contain syntax errors then compiler translate **.java** to **.class** file.

Note: In java compiler generate **.class** files based on number of classes present in source file.
If the source file contains 100 classes after compilation compiler generates 100 **.class** files.



*Step-4: Execution process. Run /execute the java application by using **java** command.*

Syntax: Java class-name
 Java Test

JVM responsibilities:

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

D:\java5pm>java Test

Ratan world

D:\java5pm>java A

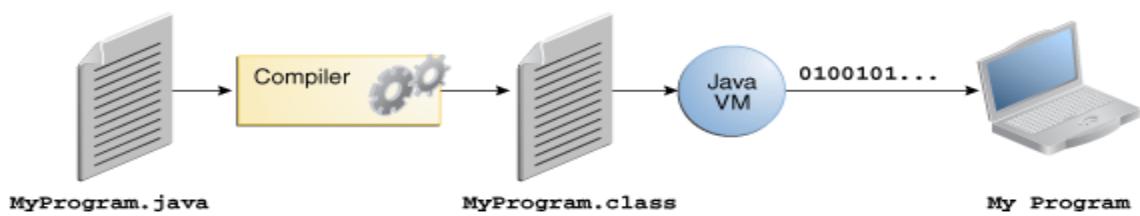
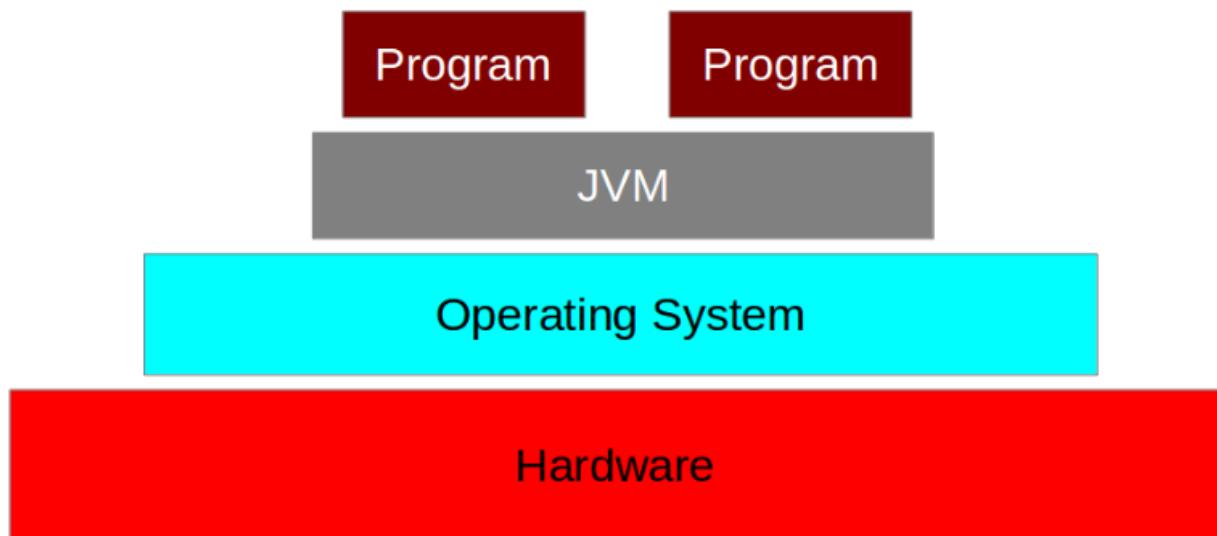
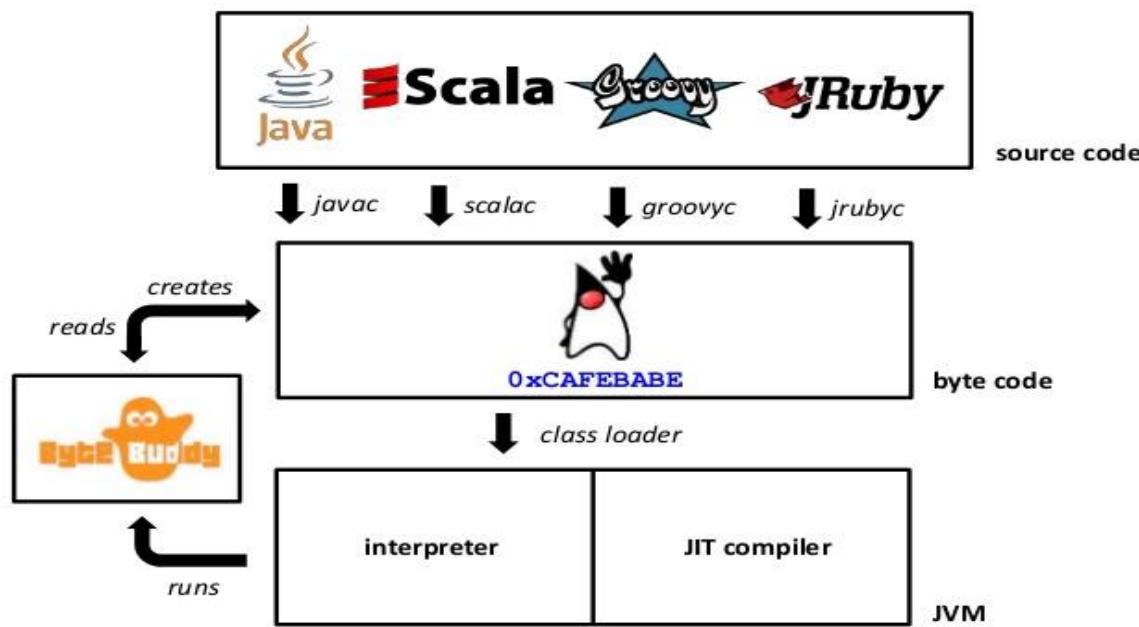
Error: Main method not found in class A, please define the main method as:

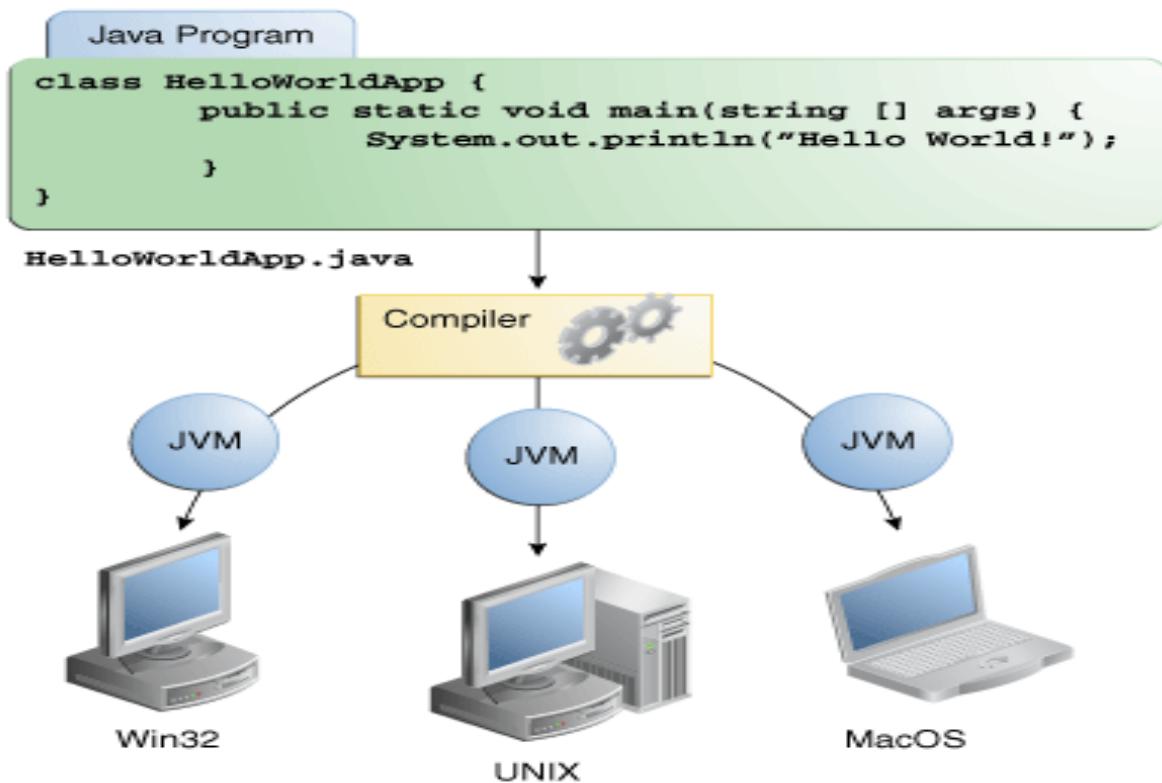
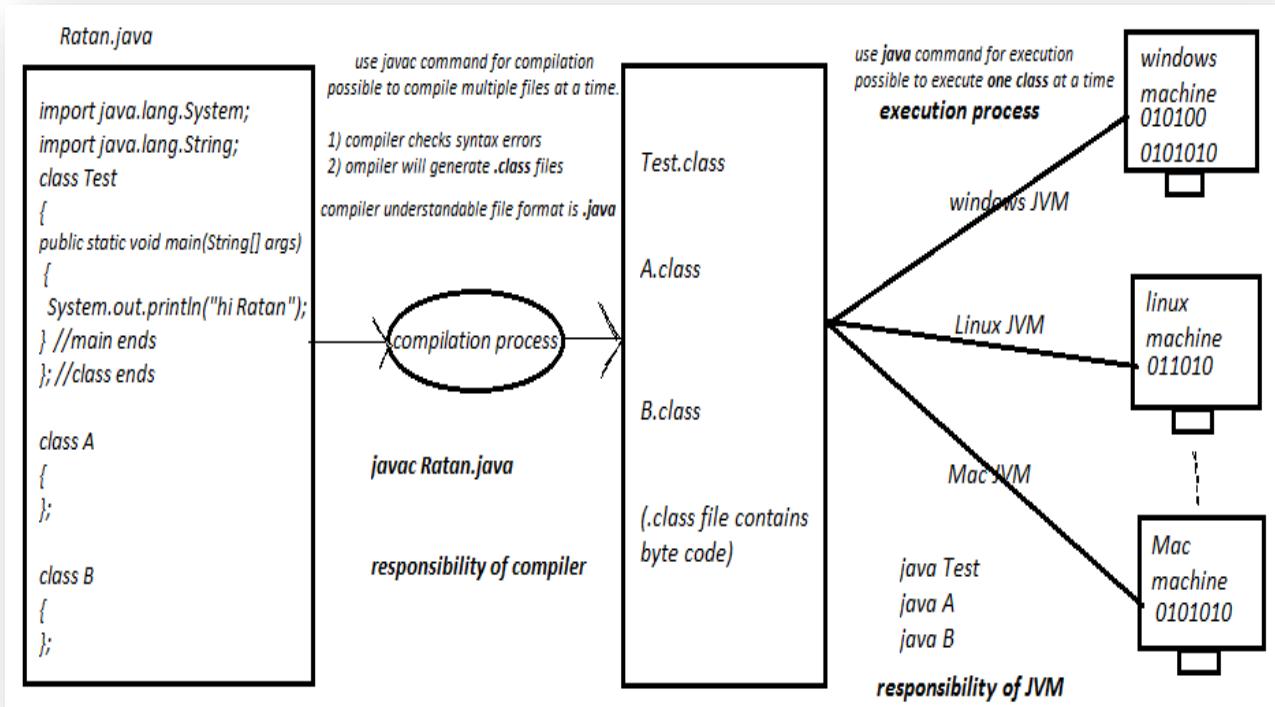
D:\java5pm>java B

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

D:\java5pm>java CCC

Error: Could not find or load main class CCC





First application conclusions:

Conclusion 1:

*Java contains 14 predefined packages but the default package in java is **java.lang***

Conclusion -2:

The source file is allows you to declare only one public class, if you are declaring more than one public class compiler generating error message.

Test.java

```
public class Test
{
}
public class A
{
}
```

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

Conclusion-3

- ✓ The coding conventions are mandatory for predefined library & optional for user defined but as a java developer must follow the coding conventions for user defined library also.
- ✓ The below example compiled & executed but it is not recommended because the class name starting with lower case letters.

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

Conclusion-4

D:

```
|-->java5pm
    |-->A.java
    |-->B.java
    |-->C.java
```

javac A.java	One file is compiled(A.java)
javac B.java C.java	Two files are compiled
javac *.java	All files are compiled
javac Emp*.java	files prefix with emp compiled (Empld.java EmpName.java...)
javac *Emp.java	files suffix with emp compiled (XEmp.java YEmp.java...)

Possible to compile more than one file at a time, but possible to execute only one class file(the class which contains main method).

Conclusion -5:

The class which contains main method is called **Mainclass** and java allows declaring multiple main class in a single source file.(but it is recommended to declare only one main class).

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

D:\java5pm>java Test
Test World!

D:\java5pm>java A
A World!

D:\java5pm>java B
B World!

Conclusion-6:

- ✓ compiler is a translator it is translating **.java** file to **.class**
Whereas JVM is also a translator it is translating **.class** file to **machine code**.
- ✓ Compiler understandable file format is **.java** file
But JVM understandable file format is **.class** file.
- ✓ The **.java** file contains high level language (English).
But **.class** file contains byte code instructions it is a platform independent code.
- ✓ Java is a platform independent language but JVM is platform dependent.
- ✓ source file allowed : only one public class
Source file allowed : multiple main classes

Print() vs Println ():

- Print():** This method used to print the data after printing control present in same line.
- Println():** This method used to print the data after printing control present in new line.

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

        System.out.println("durga");
        System.out.println("sravya");
    }
}
```

Output :

```
ratananudurga
sravya
```

Escape Sequences:

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

Escape Sequences

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

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

        System.out.println("hi\ttratan\ttsir");
        System.out.println("hi\nratan\nnsir");
        System.out.println("hi\bratan\bsir");
    }
}
```

We can apply escape sequence characters only on string data.

Java Tokens:

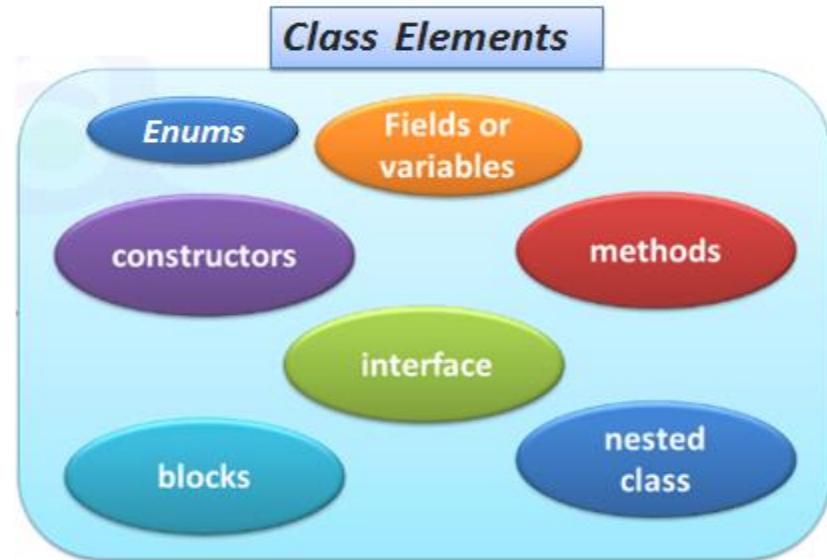
*Smallest individual part of a java program is called Token & It is possible to provide **n** number of spaces in between two tokens.*

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

Tokens are-----→class,test,{,".....etc
```

Class elements : The java class contains 7-elements

```
class Test
{    //basic elements
    variables
    methods
    constructors
    Blocks (instance , static)
    //advanced elements
    Nested classes
    Nested interfaces
    Nested Enums
}
```



Java identifiers:

Every name in java is called identifier such as, Class-name, Method-name, Variable-name...etc

Rules to declare identifier:

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

Identifier contains (a-z, A-Z, 0-9, _, \$) but not start with number & not allowed special characters.

<i>int abc123=10;</i>	<i>--->valid</i>
<i>int 123abc=40;</i>	<i>--->Invalid</i>
<i>int abc*123 =50;</i>	<i>--->Invalid</i>
<i>int abc_123=20;</i>	<i>---> Invalid</i>

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

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

3. The identifier must be unique.(duplicates are not allowed)

```
Class A{ }
Class A{ }
```

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

```
int if=10;
int try=20;
```

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

6. It is possible to use predefined class names & interfaces names as a identifier but it is not recommended.

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

        int Serializable = 20;
        System.out.println(Serializable);
    }
}
```

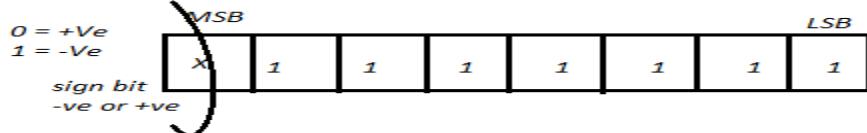
Java primitive Data Types :(8)

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

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

Byte data type information

Size : 1-byte
 MAX_VALUE : 127
 MIN_VALUE : -128
 Range : -128 to 127
 Formula : -2^n to 2^{n-1}



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

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

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

```

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

Printing variables:

```

int eid=111;
System.out.println(eid);      //valid
System.out.println("eid");    //invalid
System.out.println('eid');    //invalid

```

User provided values are printed :

```

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

```

Default values(assigned by JVM) :

```

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

```

String:

- ✓ *String is not a data type & it is a class present in java.lang package to represent group of characters or character array enclosed with in double quotes.*
- ✓ *The default value of the String is null*

Note : For any class type the default value is null.

```

String ename="ratan";
System.out.println(ename);    //ratan

```

```

String str;
System.out.println(str);     //null

```

float vs. double :

In java the decimal values are by default double values hence to represent float value use f constant or perform type casting.

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

ex-1:

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

        double d=20.5;
        System.out.println(d);
    }
}
```

ex-2: A double variable can provide precision up to 15 to 16 decimal points as compared to float precision of 6 to 7 decimal digits.

```
class Test
{
    public static void main(String[] args)
    {
        float f=10.123456789f;
        System.out.println(f);

        double d=10.123456789123456789;
        System.out.println(d);
    }
}
E:\>java Test
10.123457
10.123456789123457
```

Therefore a number like 3.14159265359 would most likely get rounded to 3.141593.

```
class Test
{
    public static void main(String[] args)
    {
        float f=3.14159265359f;
        System.out.println(f);

        double d=10.123456789123456789;
        System.out.println(d);
    }
}
E:\>java Test
3.141592
10.123456789123457
```

Number system in java :

- ✓ *Binary number system* : only two digits '0' and '1' so its base is 2
- ✓ *Octal number system* : It has eight digits (0, 1, 2, 3, 4, 5, 6, 7) so its base is 8.
- ✓ *Decimal number system* : It has ten digits (0,1,2,3,4,5,6,7,8,9) so its base is 10.
- ✓ *Hexadecimal number system* : It has the digits and letters [0-9], [a-f] so its base is 16.

Binary literals:

Java 7 allows you to express integral types (byte, short, int, and long) in binary number system.
To specify a binary literal, add the prefix `0b` or `0B` to the integral value.

```
public class Test {
    public static void main(String[] args) {
        byte b1 = 0b101;
        short s1 = 0b111;
        int i1 = 0b100;
        long l1 = 0b000001111100001;
        System.out.println("b1 = "+b1);
        System.out.println("s1 = "+s1);
        System.out.println("i1 = "+i1);
        System.out.println("l1 = "+l1);
    }
}

class Test
{
    public static void main(String[] args)
    {
        int x=0b101;
        int y=0b110;
        int z=x+y;
        System.out.println(x + "+" + y + "=" + z);

        //If you want to output in binary format, use Integer.toBinaryString()
        System.out.println(Integer.toBinaryString(x) + "+" + Integer.toBinaryString(y)
        + "=" + Integer.toBinaryString(z));
    }
}
```

Underscores in numeric literals:

With Java 7, you can include underscores in numeric literals to make them more readable. The underscore is only present in the representation of the literal in Java code, and will not show up when you print the value.

Rules to declare underscores:

You cannot use underscore at the beginning or end of a number.

```
int a = _10;
int a = 10_;
```

You cannot use underscore adjacent to a decimal point in a floating point literal.

```
float a = 10._0;
float a = 10_.0;
```

You cannot use underscore prior to an F or L suffix

```
long a = 10_100_00_L;
float a = 10_100_00_F;
```

You can't put an underscore before or after the binary or hexadecimal identifiers b and x.

```
byte size = 0_b111101;
byte me = 0b_111101;
int hexed = 0_x_BABE;
```

Could you quickly conclude how many ones and zeros in blow code?

Less readability:

```
int overflow = 0b1010101010101010101010101010101011;
long bow = 0b101010101010101010101010101010111L;
```

More readability:

```
int overflow = 0b1010_1010_1010_1010_1010_1010_1010_1011;
long bow = 0b1_01010101_01010101_01010101_01010111L;
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
```

```
class Test
{
    public static void main(String[] args)
    {
        int val1 = 10000000; //10 Million
        System.out.println("Amount is :" +val1);

        int val2 = 10_000_000; //10 Million : more readable format
        System.out.println("Amount is :" +val2);
    }
}
```

✓ Consecutive underscores is legal.

```
int n = 1000_____000;
```

✓ Underscores can be included in other numeric types as well.

```
double d = 1000_000.0;
int hex = 0xdead_c0de;
int bytes = 0x1000_0000;
```

✓ Underscore can be included after the decimal point.

```
double d = 1000_000.000_000d;
```

Java Introduction: interview questions

- 1) What is the importance of java?
- 2) What do you mean by API document?
- 3) What are the comments in java?
- 4) Why all the technologies & frame works depends on java?
- 5) What is the difference between open sources & licensed give some examples?
- 6) What do you mean by platform dependent & platform independent?
- 7) What are the predefined constants in java?
- 8) How many reserved words in java & how many keywords?
- 9) How many parts of java?
- 10) What is the difference between header files & packages?
- 11) Author of C, CPP, java?
- 12) What is the difference between #include & import statement?
- 13) What is difference between ASCII & UNICODE?
- 14) What is the difference between editor & IDE?
- 15) What is the difference between path & class-path?
- 16) Package contains what are the elements?
- 17) Give some points about jdk & jre?
- 18) What is the purpose of environmental variables setup?
- 19) Class contains how many elements what are those?
- 20) What are responsibilities of compiler?
- 21) What is the purpose of JVM?
- 22) JVM platform dependent or independent?
- 23) Can we have multiple public classes in single source file?
- 24) What is main class & is it possible to declare multiple main classes in single source file or not?
- 25) Possible to execute more than one class at time or not?
- 26) Define the token?
- 27) What is the default package in java?
- 28) What is the difference between print () vs. println ()?
- 29) Define escape sequence character?
- 30) What do you mean by identifier?
- 31) Give some information about _ (underscore in numeric literals)?
- 32) Give me some information about number system?
- 33) What are the commands required for compilation and execution?
- 34) The compiler understandable file format & JVM understandable file format?
- 35) What is the purpose of data types and how many data types are present in java?
- 36) Who is assigning default values to variables?
- 37) For the class types what are the default values?
- 38) Define the camel case?
- 39) How to calculate the ranges of the data types?
- 40) What is the difference between float & double?

*****Thank you*****

Java flow control Statements

There are three types of flow control statements in java

1. Selection/conditional Statements

if
if-else
else-if
switch

2. Iteration/ looping statements`

for
for-each
while
do-while

3. Transfer statements

goto
break
continue
try
return

1. if statement: to take only one option.

syntax:

```
if (cond)
{
    true body
}
```

case 1:

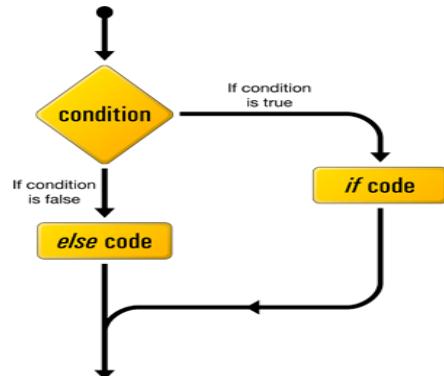
```
if (20 > 10) {
    System.out.println("20 is greater than 10");
}
```

case 2:

```
int x = 20, y = 10;
if (x > y) {
    System.out.println("x is greater than y");
}
```

If syntax:

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

**Case-1:**

```
class Test
{
    public static void main(String[] args)
    {
        int age = 20;
        if(age>22)
        {
            System.out.println("Eligible for marriage");
        }
        else
        {
            System.out.println("not Eligible for marriage ");
        }
    }
}
```

Case -2: To the if statement condition it is possible to provide Boolean constants directly.

```
if(true)
{
    System.out.println("true body");
}
else
{
    System.out.println("false body");
}
```

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

```
if(0)
{
    System.out.println("true body");
}
error: incompatible types: int cannot be converted to boolean
```

case-4 : comparision vs. assignment**valid: ==: this is comparision**

```
int x=20;
if(x==20)
{
    System.out.println("true body");
}
```

invalid: =: Assignment

```
int x=10;
if(x=20)
{
    System.out.println("true body");
}
```

case 5: The curly braces are optional but without curly braces it is possible to take only one statement that should not be a initialization.

```
if(true)
    System.out.println("true body");
else
    System.out.println("false body");
```

Observation: without curly braces it is possible to take only one statement that should not be a initialization.

```
if(true)
    int a=10;
error: variable declaration not allowed here
```

Assignments : if-else

Assignment 1: take the number from end user check it is even or odd?

Assignment 2: take the two numbers from end use print the bigger number?

Assignment 3: Take the number from end user check it is positive or Negative, and print it is even or odd. (use nested if-else condition).

ouptut :

```
enter a num : 2
Number is positive & even
```

ouptut :

```
enter a num : 3
Number is positive & odd
```

ouptut :

```
enter a num : -6
Number is Negative & even
```

ouptut :

```
enter a num : -7
Number is Negative & odd
```

Ans 1:

```
import java.util.Scanner;
class Test
{
    public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in);
        System.out.println("Enter a number:");
        int num = s.nextInt();
        if (num%2==0)
        {
            System.out.println("Even Number");
        }
        else
        {
            System.out.println("Odd number");
        }
        s.close();
    }
}
```

Ans2:

```
import java.util.Scanner;
class Test
{
    public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in);
        System.out.println("Enter first number:");
        int n1 = s.nextInt();
        System.out.println("Enter second number:");
        int n2 = s.nextInt();
        if (n1>n2)
        {
            System.out.println("First number is Bigger..... ");
        }
        else
        {
            System.out.println("Second number is Bigger..... ");
        }
        s.close();
    }
}
```

Ans-3:

```
import java.util.Scanner;
class Test
{
    public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in);
        System.out.println("Enter number:");
        int num= s.nextInt();
        if (num>0)
        {
            if (num%2==0)
            {
                System.out.println("Number is positive & even");
            }
            else
            {
                System.out.println("Number is positive & odd");
            }
        }
        else
        {
            if (num%2==0)
            {
                System.out.println("Number is Negative & even");
            }
            else
            {
                System.out.println("Number is Negative & odd");
            }
        }
        s.close();
    }
}
```

Switch statement:

- ✓ Switch statement is used to declare multiple options.
- ✓ Switch is taking the argument, the allowed arguments are
 - byte,short,int,char (primitive data types)**
 - Byte, Short, Integer, Character (wrapper classes)**
 - enum(1.5 v)**
 - String(1.7 v)**
- ✓ Inside the switch possible to declare more than one case but it is possible to declare only one default.
- ✓ Based on the provided argument the matched case will be executed if the cases are not matched default will be executed.
- ✓ **Float,double,long** is not allowed as a switch argument because these are having too large values.

Syntax:

```
switch(argument)
{
    case label-1      : statements;
                        break;
    case label-2      : statements;
                        break;
    /
    /
    case label-n      : statements;
                        break;
    default           : statements;
                        break;
}
```

Case 1:

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

Case 2 :

- ✓ Inside the switch statement break is optional.
- ✓ If we are not declaring break statement then from the matched case onwards up to the next break statement will be executed, if there is no break statement then end of the switch will be executed.
- ✓ The above situation is called as fall through inside the switch.

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

Case 3 : inside the switch the default is optional.

```
int a=10;
switch (a)
{
    case 10:System.out.println("ratan");
}
```

Case 4: Inside the switch cases are optional part.

```
int a=10;
switch (a)
{
    default: System.out.println("default");
}
```

Case 5: inside the switch both cases and default optional.

```
int a=10;
switch(a)
{
}
```

Case 6: **Invalid :** Inside the switch independent statements are not allowed. If we are declaring the statements that statement must be inside the case or default.

```
int x=10;
switch(x)
{
    System.out.println("Hello World");
}
```

Case 7 : In switch it is possible to declare the default at starting or middle or end of the switch.

```
int a=100;
switch (a)
{
    default: System.out.println("default");
    case 10:System.out.println("10");
}
```

Case 8: 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".

```
int a=10;
switch (a)
{
    case 10:System.out.println("ratan");
    case 10:System.out.println("anushka");
}
```

Case 9 : Inside the switch for the case labels & switch argument it is possible to provide expressions

```
int a=100;
switch (a+10)
{
    case 110      :System.out.println("ratan");
    case 10*5     :System.out.println("durga");
}
```

Case 10 : The advantage of fall through is used to execute common actions for multiple cases.

```
int a=2;
switch (a)
{
    case 1:
    case 2:
    case 3:System.out.println("Q-1");
    break;
    case 4:
    case 5:
    case 6:System.out.println("Q-2");
    break;
}
```

Case 11 :internal conversion for unicode. **Unicode values** a-97 A-65

Case 1 : int i=97;
 switch (i)
 { case 'a':System.out.println("ratan");
 }

Case 2 : char ch='d';
 switch (ch)
 { case 100:System.out.println("durga");
 }

Case 12: Invalid

- ✓ Inside the switch the case label must be constant values.
- ✓ If we are declaring variables as a case labels the compiler will generate error.

```
int a=10;
switch (a)
{
    case a:System.out.println("anushka");
}
```

Valid

- possible to declare final variables as a case label. Because the final variables are constants.
- The final variables are replaced with constants during compilation.

```
final int a=10;
switch (a)
{
    case a:System.out.println("anushka");
}
```

Case 13:

- ✓ Inside the switch the case label must match with provided argument data type otherwise compiler will raise compilation error "incompatible types".
- ✓ In below example we are passing String is a switch argument hence the case labels must be Strings constants.

```
String str="aaa";
Switch (str)
{
    case "aaa" :System.out.println("ratan");
    case 'a'   :System.out.println("durga");
}
```

Case 14 :

- ✓ 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".
- ✓ In below example we are passing byte as a switch argument hence the case labels must be within the range of byte.
- ✓ Case 128 : byte out of range so compiler generate error message.

```
byte b=127;
switch (b)
{
    case 127:System.out.println("ratan");
    case 128:System.out.println("anu");
}
```

case 15 : Unicode values : a=97 A=65

```
int a=97;
switch (a)
{
    case 99:System.out.println("ratan");
    case 'a':System.out.println("anu");
}
```

```
char ch='d';
switch (ch)
{
    case 100:System.out.println("ratan");
    case 'a':System.out.println("anu");
}
```

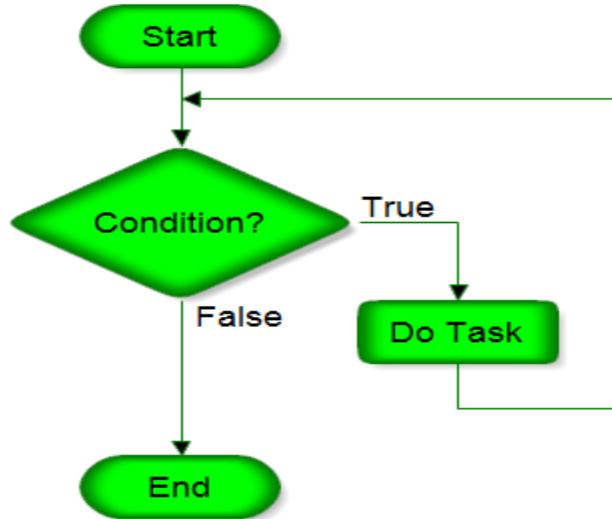
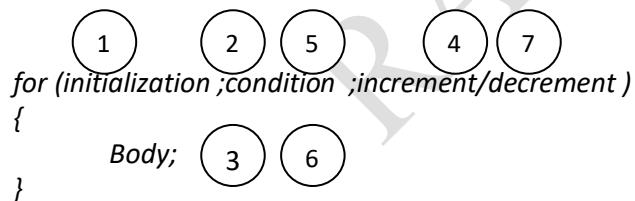
else-if:

```
class Test
{
    public static void main(String[] args)
    {
        int examscore=77;
        char grade;
        if(examscore>90)
        {
            grade='A';
        }
        else if(examscore>80)
        {
            grade='B';
        }
        else if(examscore>70)
        {
            grade='C';
        }
        else if(examscore>60)
        {
            grade='D';
        }
        else
        {
            grade='E';
        }
        System.out.println("The Grade is :" + grade);
    }
}
```

Iteration Statements:

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

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

**Flow of execution in for loop:****Case 1:****With out for loop**

```

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

By using for loop

```

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

Case 2: Inside the for loop initialization part is optional but semicolon is mandatory.

```
int i=0;
for (;i<10;i++)
{
    System.out.println("durga");
}
```

Case 3 : Initialization part it is possible to take n number of System.out.println("ratna") statements and each and every statement is separated by comma(,) .

```
int i=0;
for (System.out.println("Aruna"),System.out.println("Ratan");i<10;i++)
{
    System.out.println("anu");
}
```

Case 4 :

- ✓ Initialization part possible to declare only one initialization.
- ✓ Initialization part it is not possible to declare the data type two times whether it is same type or different types.
- ✓ But it is possible to declare the single data type with more than one variable.

Invalid

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

Valid

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

Conditional part :

Inside for loop conditional part is optional, if we are not providing condition at the time of compilation compiler will generate true constant.

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

increment/decrement :

case 1 : Inside the for loop increment/decrement part is optional.

```
for (int i=0;i<10;)
{
    System.out.println("durga");
    i++;
}
```

Case 2 : In the increment/decrement it is possible to take the n number of SOP() statements and each and every statement is separated by comma(,).

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

Case 1: error: Unreachable statement

- ✓ If the control unable to enter in particular area is called unreachable statement.
- ✓ We will get the unreachable code when we declare only Boolean constants(true,false).
- ✓ When we will give **true** condition the remaining code is unreachable
- ✓ when we will give **false** condition the body is unreachable.

Rest of the code unreachable

```
for (int i=1;true;i++)
{
    System.out.println("ratan");
}
System.out.println("rest of the code");
```

Loop body is unreachable

```
for (int i=1;false;i++)
{
    System.out.println("ratan");
}
System.out.println("rest of the code");
```

Case 2: Valid

- ✓ When you provide the condition even though that condition is represent infinite loop compiler is unable to find unreachable statements,because there may be chance of condition fail.
- ✓ When you provide Boolean constants as a condition then compiler is identifying unreachable statement because compiler knows that condition never change.

```
for (int i=1;i>0;i++)
{
    System.out.println("durga");
}
System.out.println("rest of the app");
```

Case 3: method calling return Boolean condition.

```
class Test
{
    static boolean m1()
    {
        return true;
    }
    public static void main(String[] args)
    {
        for (i=0 ;Test.m1();i++)
        {
            System.out.println("durga");
        }
    }
}
```

Case 4 :Inside the for loop each and every part is optional but semicolon is mandatory.

`for(; ;)` represent infinite loop because the condition is always true.

Case 5: error : unreachable statement

```
for ( ; ; )
{
    System.out.println("ratan");
}
System.out.println("anu"); //error: unreachable statement
```

For-each loop: (introduced in 1.5 version)

```
class Test
{
    public static void main(String[] args)
    {
        int[] a={10,20,30};
        System.out.println(a[0]);
        System.out.println(a[1]);
        System.out.println(a[2]);

        //printing data by using for-loop
        for (int i=0;i<a.length;i++)
        {
            System.out.println(a[i]);
        }

        //printing data by using for-each loop
        for (int aa: a)
        {
            System.out.println(aa);
        }
    }
}
```

for loop vs for-each loop :

- ✓ For loop is used to print the data & it is possible to apply conditions.
- ✓ For-each loop is used to print the data from starting element to ending element it is not possible to apply the conditions.

While loop: condition must be Boolean & mandatory.

```
while (Boolean-expression)
{
    body;
}
```

Case 1:

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

Case 3: error: unreachable statement

```
while (true)
{
    System.out.println("ratan");
}
System.out.println("anu");
```

Case 4: error: unreachable statement

```
while (flase)
{
    System.out.println("ratan");
}
System.out.println("anu");
```

Case 5 : Just to check the data & print the data use while loop.

```
while (itr.hasNext())
{
    System.out.println(itr.next());
}
```

for loop vs for-each vs while loop :

- ✓ To print the data from specific value to specific value with increment value use for loop.
- ✓ Just to check the data & print the data use while loop.
- ✓ For-each loop used to print the data from starting value to ending value without any condition.

Do-While:

- ✓ If we want to execute the loop body at least one time then we should go for do-while statement.
- ✓ In do-while first body will be executed then only condition will be checked.

Syntax:

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

Case 1:

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

Case 2:

```
int i=0;
do
{
    System.out.println("durga");
} while (true);
System.out.println("durgasoft");
```

Case 3:

```
int i=0;
do
{
    System.out.println("durga");
} while (false);
System.out.println("durgasoft");
```

Transfer statements: (jump statements)

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

- ✓ **Break**
- ✓ **continue**
- ✓ **return**
- ✓ **try**
- ✓ **goto**

break: Break is used to stop the execution & it is possible to use the break statement only two areas.

- ✓ Inside the switch statement.
- ✓ Inside the loops.

Case 1 : break is used to 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);
        }
    }
}
```

Case 2: if we are using break outside switch or loops the compiler will raise compilation error
"break outside switch or loop"

```
if (true)
{
    System.out.println("ratan");
    break;
    System.out.println("nandu");
}
```

Continue: it is used skip the current iteration and it is continuing 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);
        }
    }
}
```

Flow control statement interview Questions

- 1) How many types 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) Is it possible to use variables as case labels or not?
- 8) While declaring if , if-else , switch curly braces are optional or mandatory?
- 9) Switch is allowed String argument or not?
- 10) Switch allows float,double,long arguments or not?
- 11) Inside switch how many cases possible & how many default declarations are possible?
- 12) What are the advantages of fall through inside the switch?
- 13) What is difference between if & if-else & switch?
- 14) What is the default condition of for loop & who will provide the default condition?
- 15) What is the difference between for loop & for-each loop?
- 16) When we will get compilation error like “incompatible types”?
- 17) for (;;) representing?
- 18) When we will get compilation error like “unreachable statement ”?
- 19) What is the difference between for loop & while loop?
- 20) Is it possible to declare while without condition?
- 21) What is the difference between while and do-while?
- 22) What do you mean by transfer statements and what are transfer statements in java?
- 23) We are able to use break statements how many places and what are the places?
- 24) What is the difference between break& continue?
- 25) Is it possible to use break inside the if statement?

***** **Thank you** *****

Java class concept

The class contains 6-elements,

Class Test

{

Variables

Methods

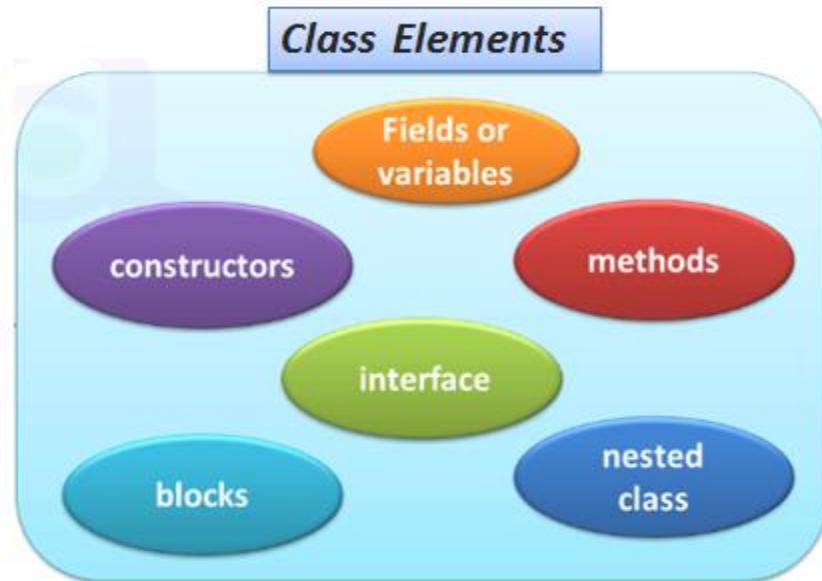
Constructors

Blocks (Instance blocks, Static blocks , synchronized blocks)

Nested Classes

Interfaces

}



Java Variables:

- ✓ *Variables are used to store the constant values by using these values we are achieving project requirements.*
- ✓ *Generally project level variables are :*

```
Employee project : int eid; String ename; double esal;
Student project   : int sid; String sname; double smarks;
```
- ✓ *Variables are also known as **fields** of a class or **properties** of a class.*
- ✓ *There are three types of variables in java*
 1. Local variables.
 2. Instance variables.
 3. Static variables.

✓ All variables must have some type. That type it may be,

- Primitive type (int, float,...)

```
int a=10;
```

- Array type

```
int[ ] a;
```

- class type

```
Test t;
```

```
Employee emp;
```

- Enum type

```
Week k;
```

✓ Variable declaration is composed of four components in order,

- Zero or more modifiers.

```
public int a=10;
```

- The variable type.

-->modifier (specify permission)

- The variable name.

-->data type (type of the variable)

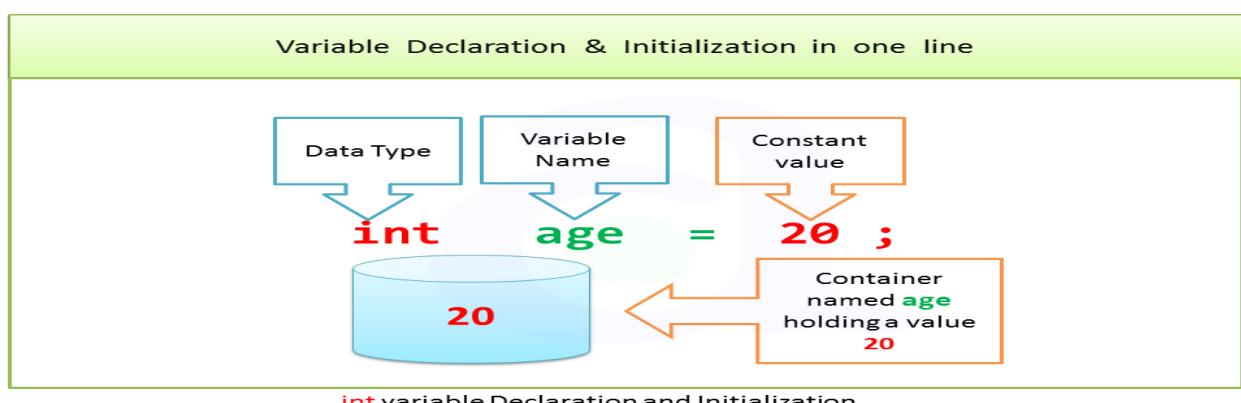
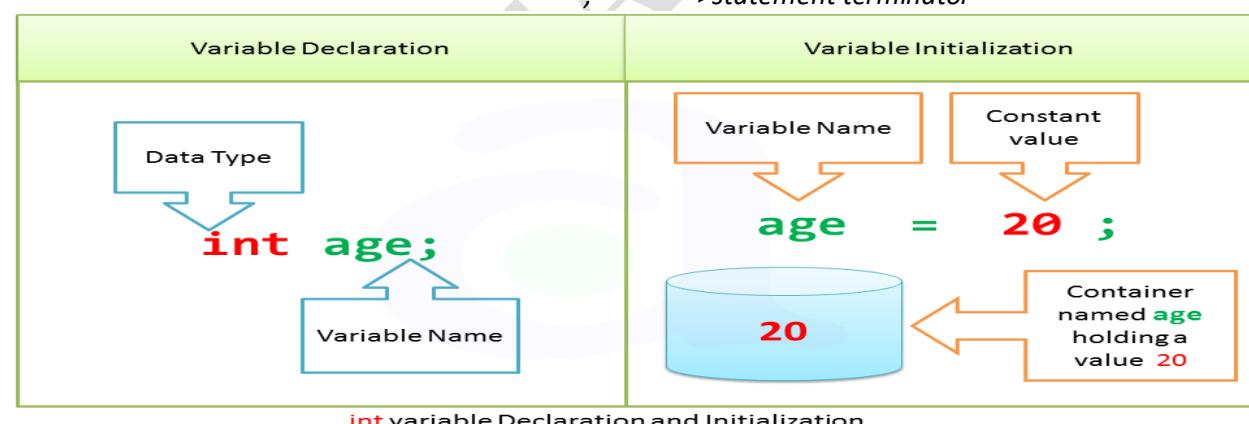
- Constant value/ literal

-->variable name

```
public final int x=100;
```

-->constant value or literal;

-->statement terminator



Local variables:

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

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

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

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

- ❖ Local variables memory allocated when method starts & memory released when method completed.
- ❖ The local variables are stored in stack memory.

There are five types of memory areas in java:

1. Heap memory
2. Stack memory
3. Method area
4. SCP(String constant pool) memory
5. Native method stacks

Areas of java: There are two types' areas in java.

Instance Area:

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

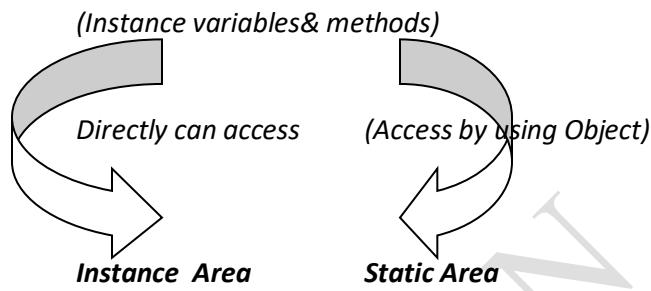
Static Area:

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

Instance variables (non-static variables):

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

Instance variable accessing:



ex : File Name : Ratan.java

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

- ✓ In java program execution always starts from main method called by JVM.
- ✓ The JVM is responsible to execute only main method so must call the user defined method (m1) inside the main method then only user method will be executed.

ex: Observation

```

class A
{
    int num=10;                      //instance variable
}
class Test
{
    void m1()                         //instance method of different class
    {
        A a = new A();
        System.out.println(a.num);
    }
}

```

Instance to instance : direct Access : only within the same class
Instance to instance : different classes : object required

Note: Always access the instance members (variables & method) by using object.

Operator overloading:

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

```

class Test
{
    public static void main(String[] args)
    {
        System.out.println(10+20);                //addition
        System.out.println("ratan"+"anu");          //concatenation

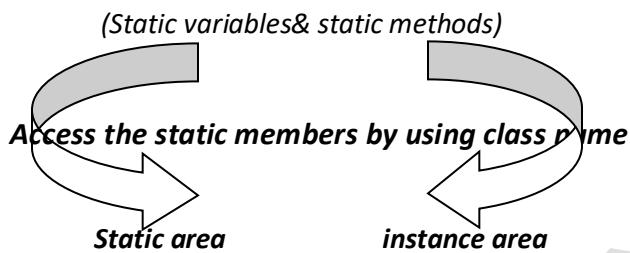
        int a=10;
        int b=20;
        int c=30;
        System.out.println(a);
        System.out.println(a+"---");
        System.out.println(a+"---"+b);
        System.out.println(a+"---"+b+"----");
        System.out.println(a+"---"+b+"----"+c);
    }
}

```

Static variables (class variables):

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

Static variables & methods accessing:



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

        Test t = new Test();
        t.m1();           //instance method calling
    }
    //instance method
    void m1()
    {
        System.out.println(Test.a);
        System.out.println(Test.b);
    }
}
```

Note-1: Always Access the instance members by using object.

Note-2: Always Access the static members by using class-name.

Static variables calling: There are three ways to access the static members

1. By using class-name. (Project level always use this approach)
2. Direct accessing
3. By using reference variable.

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

        System.out.println(a);                       //2-way(directly possible)

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

Example:

- ✓ It is possible to create the objects inside the main method & inside the user defined methods also.
- ✓ When we create the object inside the method, the scope of the object is only within the method.
- ✓ When we create object inside method that object is destroyed when method completed, if any other method required object then create the object inside that method.

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

Variables vs. default values:

ex 1: JVM will assign default values only for the instance & static variables but not for local variable .

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

ex 2:

- ✓ JVM does not assign default values for local variables.
- ✓ In java before using local variables must initialize some values to the variables otherwise compiler generates compilationerror “variable a might not have been initialized”.

Case 1: Invalid : Local variable not initialized so JVM will generate error message.

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

error: variable a might not have been initialized

case 2: Valid : Before using local variable must initialize the values.

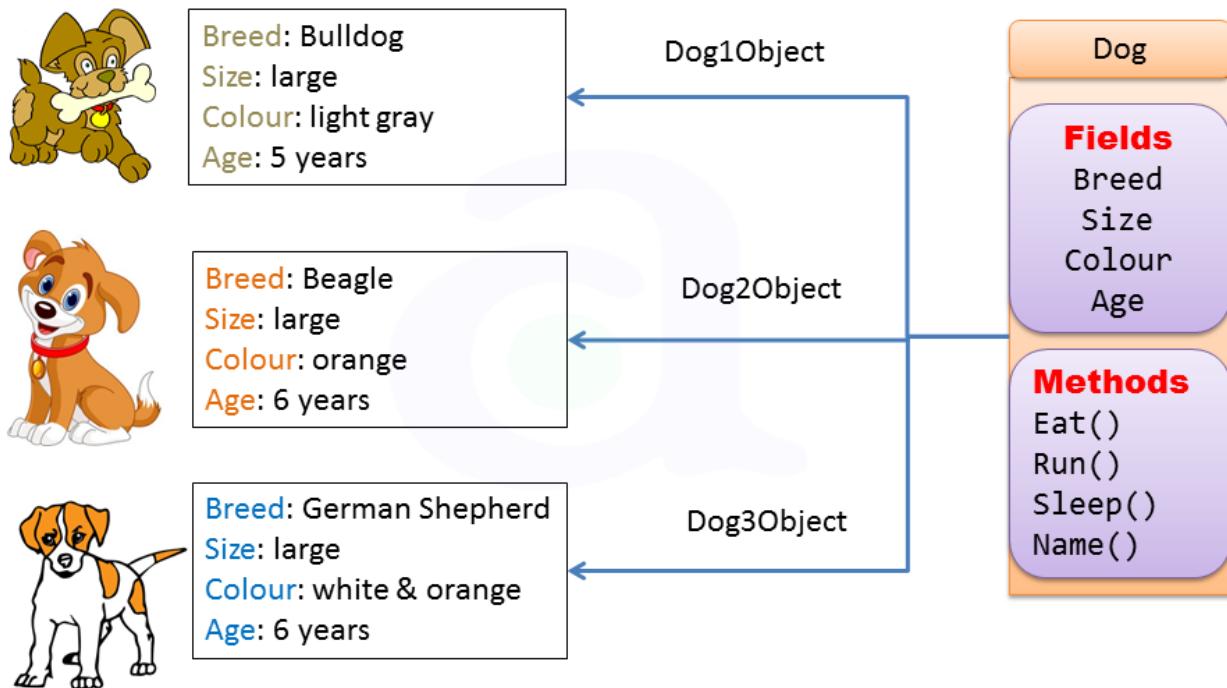
```
class Test
{
    public static void main(String[] args)
    {
        double d;
        :::::
        d=10.5;
        System.out.println(d);
    }
}
```

Case 3: invalid: initialized after using the variables.

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

Class vs. Object:

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



Dog is a single class but here we created three objects means three different memories are created.

Instance vs. Static variables:

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

```

System.out.println(Test.b);

Test t1 = new Test();
System.out.println(t1.a);
System.out.println(Test.b);
Test.b=333;

Test t2 = new Test();
System.out.println(t2.a);
System.out.println(Test.b);
}

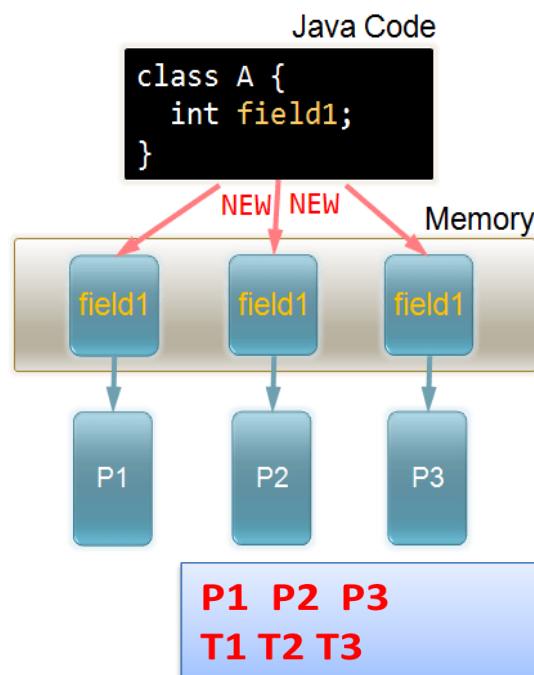
}

```

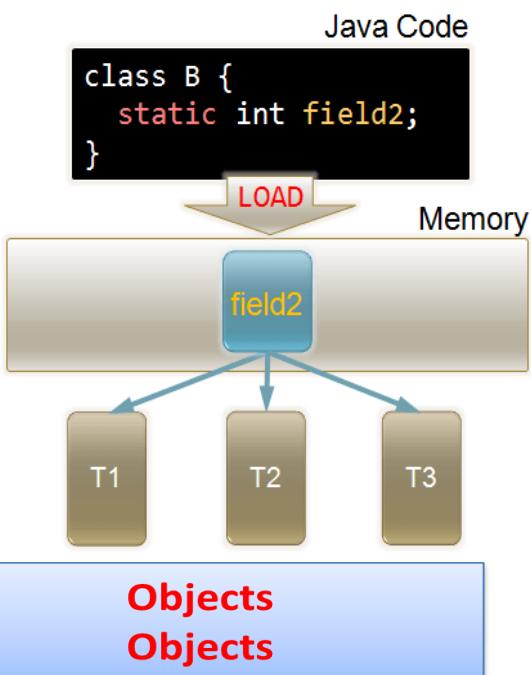
- ✓ In case of instance variables the JVM will create separate copy of memory for each and every object.
- ✓ Always access the instance by using object.
- ✓ When we perform modifications on instance variables that will be reflected on same object only.(only one object)

- ✓ In case of static variables irrespective of object creation per class single memory is allocated.
- ✓ Always access the static by using clas-name.
- ✓ When we performed modifications on static variable that will be reflected on multiple objects.

● Instance Fields



● Static Fields

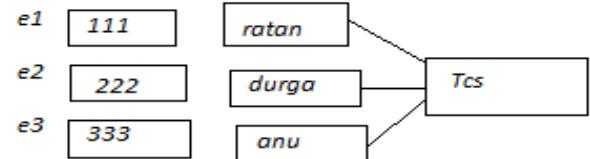


Instance variable vs. static variable:

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



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

**Different ways to initialize the variables:**

```
int p=10,q=20,r=30;           //10 20 30
int i=10,j=20,k;             //10 20 0
int x=10,y,z;                //10 0 0
static int d=10,e,f=30;       //10 0 30
static int a,b,c;             //0 0 0
```

***** Thank You : Completed *****
Java Methods

- ✓ Inside the classes it is not possible to write the business logics directly, so inside the class declares the method to write the logics of the application.
- ✓ Methods are used to write the business **logics** of the application.

Coding convention:

Method name starts with lower case letter and every inner word starts with uppercase letter
ex: post() , charAt() , toUpperCase() , compareToIgnoreCase().....etc

There are two types of methods in java,

1. Instance method
2. Static method

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

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

Instance method:

```
Void add()
{ body : logics here : instance area
```

}

Instance member's memory is allocated during object creation hence accesses the instance members by using object-name (reference-variable).

Objectname.instancemethod();

Test testobj = new Test();

testobj.add();

Static method:

static void mul()

{ *Body: logics here: static area*
}

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

Classname.staticmethod();

Test.mul();

Note: Always access the instance methods using object-name & static methods using class-name.

Every method contains three parts.

```
Static void mul()
{
    Body (Business logic);
}
Test.m1();
```

-----> **method declaration**
 -----> **method implementation**
 -----> **method calling**

Method Syntax:

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

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

Public void m1()

Private int m2(int a,int b)

Private String m2(char ch) throws Exception

Example -1 :

```

class Test
{
    void sayHello()
    {
        System.out.println("Good Morning");
    }
    static void add(int num1,int num2)           //local variables
    {
        System.out.println(num1+num2);
    }
    void mul(int val1,int val2)
    {
        System.out.println(val1*val2);
    }
    void login(String username,String password)
    {
        if(username.equals("ratan"))
        {
            System.out.println("Login Success");
        }
        else
        {
            System.out.println("Login Fail");
        }
    }
    static void display(int a,boolean b,char ch)
    {
        System.out.println(a+" "+b+" "+ch);
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.sayHello();

        Test.add(4,5);

        int x=10,y=20;
        t.mul(x,y);

        t.login("ratan","durga");

        Test.display(10,false,'a');
    }
}

```

- ✓ If the method is taking parameters (inputs to functionality), while calling that method must pass the values to parameters.
- ✓ If the method expecting three arguments must pass three arguments, not less than three not more than three.
- ✓ Method arguments are always local variables.
- ✓ While calling the method we can pass constants as a argument values **Test.add(10,20)**
Or we can pass variables as a argument values **Test.add(x,y)**

Example -2:

Case 1: java methods return type is mandatory, otherwise the compilation will generate error.

```
class Test
{
    add()
    {
    }
}

error: invalid method declaration; return type required
```

Case 2:

Method Signature: Method-name & parameters list is called method signature.
Syntax: Method-name(parameter-list)
 m1(int a)
 m2(int a,int b)

Inside the class it is not possible to declare more than one method with same signature(duplicate methods) , if we are trying to declare with same signature compiler generates error.

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

error: m1() is already defined in Test
```

Case 3:

- ✓ Declaring the class inside another class is called nested classes, java supports nested classes.
- ✓ Declaring the methods inside other methods is called inner methods but java not supporting inner methods concept.

```
class Test
{
    void m1()
    {
        void m2() //inner method : error: illegal start of expression
        {
        }
    }
}
```

ex -4 :

If the application contains both instance & local variables with same name, in this case to represent instance variables we have two approaches.

1. Access by using object.
2. Access by using this keyword

'This' keyword is used to represent current class object.

Case 1:

```
class Test
{
    int a=100,b=200;
    void add(int a,int b)
    {
        System.out.println(a+b);

        Test t = new Test();
        System.out.println(t.a+t.b);           //approach-1

        System.out.println(this.a+this.b);     //approach-2
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.add(10,20);
    }
}
```

Case 2: Invalid : inside the static area this keyword not allowed

```
Int a=100,b=200;
static void add(int a,int b)
{
    System.out.println(this.a+this.b);
}
```

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

Case 3: In almost all cases to represent instance variables we are using this keyword but inside the static area this keyword is not allowed hence use object creation approach.

```
int a=100,b=200;
static void add(int a,int b)
{
    Test t = new Test();
    System.out.println(t.a+t.b);
}
```

ex-5 : methods vs. All data- types

- ✓ In java numeric values are by default integer values,
 - To represent long value use small "l" or capital "L" constant.
 - To represent byte, short perform typecasting.
- ✓ In java decimal values are by default double values ,
 - to represent float value use "f" constant.

```

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

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

```

ex-6 : java method calling: one method is able to call more than one method

```

class Test
{
    void m1()
    {
        m2(100);
        System.out.println("m1....");
        m2(20);
    }
    void m2(int a)
    {
        System.out.println("m2...." + a);
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
    }
}

```

ex- 7 : methods vs. return type.

- ✓ Java methods return type is mandatory & void represents no return value. And methods can have any return type like primitive type such as byte, short, int, long, float & Arrays type , Class type , Interface type ,Enum type....etc
- ✓ If the method is having return type other than void then must return the value by using **return** keyword otherwise compiler will generate error message "**missing return statement**".
- ✓ Inside the method it is possible to declare only one return statement, that statement must be last statement of the method otherwise compiler will generate error message.
- ✓ Method is able to returns the value, it is recommended to hold the return value check the status of the method but it is optional.

```

class Test
{
    int add(int a,int b)
    {
        int c;
        c = a+b;
        return c;
    }
    static String login(String uname,String upwd)
    {
        if(uname.equals("ratan"))
        {
            return "success";
        }
        else
        {
            return "fail";
        }
    }
    boolean print(int num)
    {
        if(num==10)
            return true;
        else
            return false;
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        int addition = t.add(10,20);
        System.out.println("Return value="+addition);

        String status = Test.login("ratan", "anu");
        System.out.println("Logins status="+status);

        boolean b = t.print(10);
        System.out.println("Return value="+b);
    }
}

```

Ex-8 : method vs. return variables

Case 1: if the application contains both instance & local variables the return value will be local.

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

Case 2: No local variables in application the return value will be instance value.

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

Case 3: If the application contains both local & instance variables with same name then first priority goes to local variables but to return instance value use **this** keyword.

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

ex 9: Observations

Case 1: It is possible to print return value of the method in two ways,

1. Hold the return value & print that value.
2. Directly print the value by calling method using `System.out.println()`

```
class Test
{
    int m1()
    {
        System.out.println("m1 method");
        return 10;
    }
    public static void main(String[] args)
    {
        Test t =new Test();
        int x = t.m1();
        System.out.println("return value="+x);           //Approach-1

        System.out.println("return value="+t.m1());       //Approach-2
    }
}
```

Case 2: If the method is having return type is `void` but if we are trying to call method by using `System.out.println()` then compiler will generate error message.

```
Static void m2()
{
    System.out.println("m2 method");
}
System.out.println(Test.m2());
error:'void' type not allowed here
```

ex-10:

- ✓ To take the input from end user use scanner class.
- ✓ Scanner class present in **java.util** package and it is introduced in 1.5 versions.
- ✓ To create an object of Scanner class, we usually pass the predefined object **System.in**, which represents the standard input stream.

To get int value	----> s.nextInt()
To get float value	----> s.nextFloat()
To get String value	----> s.next()
To get single line	----> s.nextLine()
To read a single character	----> next().charAt(0)
To close the input stream	----> s.close()

```

import java.util.Scanner;
class Test
{
    public static void main(String[] args)
    {
        Scanner s=new Scanner(System.in);

        System.out.println("enter emp hobbies");
        String ehobbies = s.nextLine();

        System.out.println("enter emp no");
        int eno=s.nextInt();

        System.out.println("enter emp name");
        String ename=s.next();

        System.out.println("enter emp salary");
        float esal=s.nextFloat();

        System.out.println("enter emp gender");
        char gender=s.next().charAt(0);

        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);
        System.out.println("emp Gender----->" + gender);
        s.close();
    }
}

```

ex-11:

```

import java.util.Scanner;
class Test
{
    public static void main(String[] args)
    {
        int count=0,sum=0;
        Scanner s = new Scanner(System.in);
        System.out.println("enter number to perfrom addition");
        while (s.hasNextInt())
        {
            sum = sum+s.nextInt();
            count++;
        }
        System.out.println("Total number:"+count+" addition is:"+sum);
        s.close();
    }
}

```

ex-12:

```

import java.util.Scanner;
class Test
{
    public static void main(String[] args)
    {
        String s = "hi, This is Ratanit. ";

        //Create scanner Object and pass string value
        Scanner scan = new Scanner(s);

        //Check if the scanner has a tokens or not
        System.out.println("Boolean Result: " + scan.hasNext());

        //Print the complete string
        System.out.println("String: " +scan.nextLine());
        scan.close();
    }
}

```

ex 13:

```

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

```

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

```

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

```

if the instance variable names and local variable names are different we can assign directly without using this keyword.

```

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

```

Ex-15: conversion of local values to instance**Case 1: Inside the method taking local value then converting local to instance**

```
import java.util.*;
class Test
{
    int sid;
    void details()
    {
        Scanner s = new Scanner(System.in);
        System.out.println("enter student id");
        int sid = s.nextInt();
        this.sid=sid;
    }
    void disp()
    {
        System.out.println("student is="+sid);
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.details();
        t.disp();
    }
}
```

Case 2: Directly assigning values to instance variables.

```
import java.util.*;
class Test
{
    int sid;
    void details()
    {
        Scanner s = new Scanner(System.in);
        System.out.println("enter student id");
        sid = s.nextInt();
    }
    void disp()
    {
        System.out.println("student is="+sid);
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.details();
        t.disp();
    }
}
```

ex 16 : Method recursion method is calling itself during execution is called recursion.

case 1: (normal output)

```
class RecursiveMethod
{
    static void withoutrecursive(int a)
    {
        System.out.println("number is : "+a);
    }
    static void withrecursive(int a)
    {
        System.out.println("number is : "+a);
        if (a==5)
            return;
        withrecursive(++a);
    }
    public static void main(String[] args)
    {
        RecursiveMethod.withoutrecursive(1);
        RecursiveMethod.withoutrecursive(2);
        RecursiveMethod.withoutrecursive(3);
        RecursiveMethod.withoutrecursive(4);
        RecursiveMethod.withoutrecursive(5);

        RecursiveMethod.withrecursive(1);
    }
}
```

case 2:- (StackOverflowError)

```
class RecursiveMethod
{
    static void recursive(int a)
    {
        System.out.println("number is :- "+a);
        if (a==0)
            return;
        recursive(++a);
    }
    public static void main(String[] args)
    {
        RecursiveMethod.recursive(10);
    }
}
```

ex 17 : Stack Memory

- ✓ The JVM will create empty stack memory just before calling main method & JVM will destroyed empty stack memory after completion of main method.
- ✓ When JVM calls particular method that method entry and local variables stored in stack memory When the method completed, that particular method, local variables destroyed from stack memory That released memory used by some other methods.

```
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);
    }
}
```

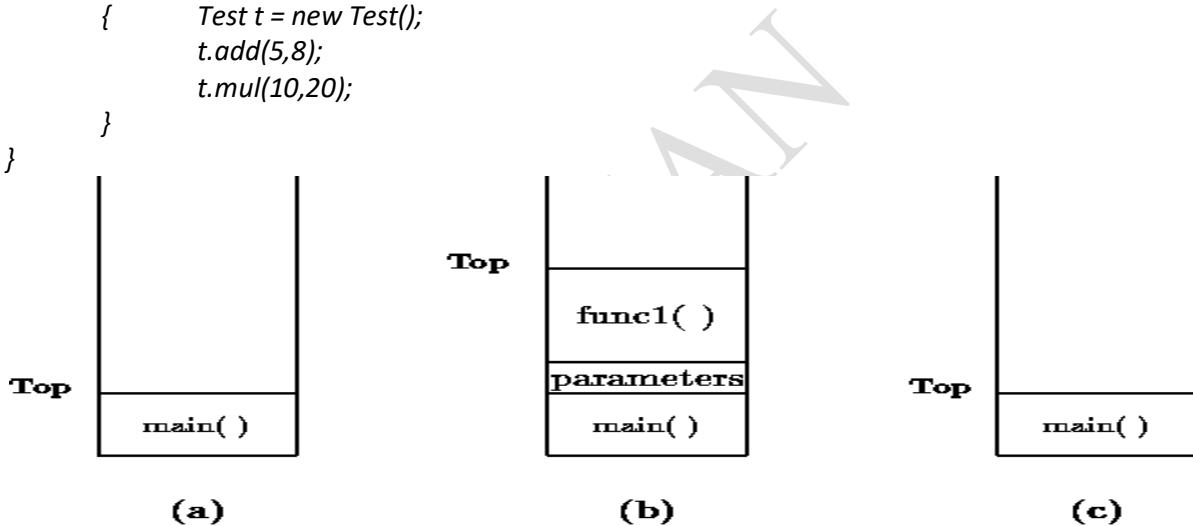


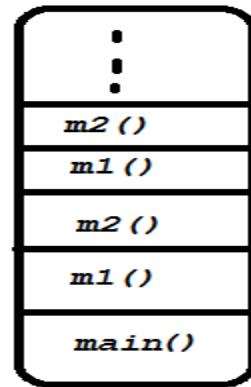
Figure 14.13: Organization of the Stack



ex 18: when we call methods recursively then JVM will generate StackOverflowError.

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

StackOverflowError



*******Methods Completed*******

CONSTRUCTORS

Class vs. Object:

- ✓ Class is a logical entity it contains logics whereas object is physical entity it is representing memory.
- ✓ Class is blue print it decides object creation without class we are unable to create object.
- ✓ Based on single class it is possible to create more than one object but every object occupies memory.
- ✓ Declare the class by using class keyword & create the object by using new keyword.

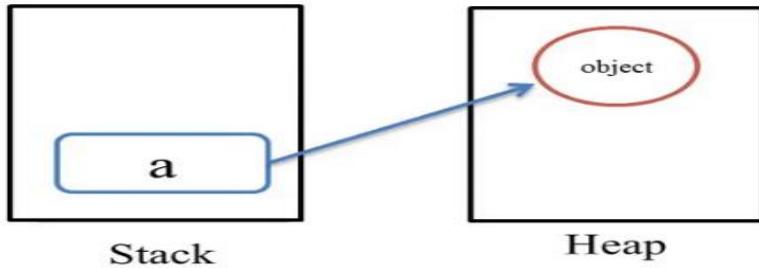
Object creation syntax:

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

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

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

```
Test a = new Test();
```

**Different ways to create object:**

- ✓ By using new operator
- ✓ By using clone() method
- ✓ By using new Instance() method
- ✓ By using instance factory method.
- ✓ By using static factory method
- ✓ By using pattern factory method
- ✓ By using deserialization process

new keyword:

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

Rules to declare constructor:

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

0-arg constructor declaration

```
Test()
{
    //logics here
}
```

1-arg constructor declaration

```
Test(int a)
{
    //logics here
}
```

Types of constructors in java:

There are two types of constructors,

- | | |
|-----------------------------|-------------------------------------------------------------|
| 1) Default Constructor | : provided by compiler : 0-arg constructor with empty impl. |
| 2) User defined Constructor | : declared by user : it may be 0-arg & parameterized. |

Default Constructor:

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

Application before compilation

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

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

Application after compilation

```

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

```

User defined constructor:

```

class Test
{
    void m1()
    {
        System.out.println("m1 method");
    }
    Test()
    {
        System.out.println("0-arg constructor");
    }
    Test(int i)
    {
        System.out.println("1-arg constructor");
    }
    public static void main(String[] args)
    {
        Test t1=new Test();
        Test t2=new Test(10);
        t1.m1();
        t2.m1();
    }
}

```

Note 1:

Methods are used to write logics of the application these are executed when we call methods.

Constructor is a special method to write the logics, these logics are automatically executed during object creation process.

Note 2:

Only the compiler generated 0-argument is called default constructor.

The user defined 0-argument constructor is not a default constructor.

Note 3:

Inside the class if we are not declaring any constructor (either 0-arg or parameterized) then only compiler generate 0-arg constructor with empty implementation is called default constructor.

Inside the class if we are declaring at least one constructor (either 0-arg or parameterized) then compiler not generating default constructor.

Observation: The below code is not compiled.

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

If we are trying to compile above application the compiler will generate error message "Cannot find symbol" because compiler is unable to generate default constructor.

Constructor calling: 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

case-1 : Call the methods by using method name but call the contructor using this keyword.

```
class Test
{
    Test()
    {
        this(100);
        System.out.println("0-arg constructor logics");
    }
    Test(int a)
    {
        this('g',10);
        System.out.println("1-arg constructor logics");
    }
    Test(char ch,int a)
    {
        System.out.println("2-arg constructor logics");
    }
    public static void main(String[] args)
    {
        Test t = new Test();
    }
}
```

Case -2 : Inside the constructor this keyword must be first statement.

```
Test()
{
    System.out.println("0 arg");
    this(10);
}
error : call to this must be first statement in constructor.
```

Case 3: One method is able to call more than one method at a time but
One constructor is able to call only one constructor at time.

```
Test()
{
    this(10);
    this(10,20);
    System.out.println("0-arg cons");
}
error: call to this must be first statement in constructor
```

Object creation formats: 2-formats of object creation.

- | | | |
|--------------------|------------------------------|-----------------------------|
| 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
        Test t = new Test();
        t.m1();

        //Nameless object
        new Test().m1();
    }
}
```

Case 1: Method calling: **Named object is recommended Approach.**

```
void m1(){}
void m2(){}
void m3(){}
```

Named Object Approach: Here one object created

```
Test t = new Test();
t.m1();
t.m2();
t.m3();
```

Name less object Approach : Here three objects are created

```
new Test().m1();
new Test().m2();
new Test().m3();
```

Assignment: Check this Assignment after compilation of 3-cases. (This assignment present in next page)
Write the application with named & nameless object Approaches.

```
class Test
{
    Take 1-ins method (Dog d, Puppy p, Animal a) ---> void return type
    Take 1-static method () ---> Student object return type
    public static void main(String[] args)
    {
        Call 2-methods by passing arguments & hold the return value & print it
    }
}
```

Case 2: Method arguments: name less object Approach is recommended.

```
class Emp{
}
class Student{
}
class Test
{
    void m1(Emp e,Student s,int a)
    {
        System.out.println("m1 method");
        System.out.println(e+" "+s+" "+a);
    }
    public static void main(String[] args)
    {
        //Named object: passing arguments
        Test t1 = new Test();
        Emp e = new Emp();
        Student s = new Student();
        t1.m1(e,s,10);

        //Nameless object : passing arguments
        Test t2 = new Test();
        t2.m1(new Emp(),new Student(),10);
    }
}
```

Case 3: Method Return Type: Name less object Approach is recommended

```
class Emp{
}
class Test
{
    static Emp m1()
    {
        System.out.println("m1 method");
        Emp e = new Emp();
        return e;
    }
    static Emp m2()
    {
        System.out.println("m2 method");
        return new Emp();
```

```

        }
    public static void main(String[] args)
    {
        Emp e1 = Test.m1();
        System.out.println("m1 method return value="+e1);

        Emp e2 = Test.m2();
        System.out.println("m2 method return value="+e2);
    }
}

```

Application With named object:

```

class Dog{
}
class Puppy{
}
class Animal{
}
class Student{
}
class Test
{
    void m1(Dog d,Puppy p,Animal a)
    {
        System.out.println("m1 method");
        System.out.println(d+" "+p+" "+a);
    }
    static Student m2()
    {
        System.out.println("m2 method");
        Student s = new Student();
        return s;
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        Dog d = new Dog();
        Puppy p = new Puppy();
        Animal a = new Animal();
        t.m1(d,p,a);

        Student s = Test.m2();
        System.out.println("Return value="+s);
    }
}

```

Application with nameless object Approach:

```

class Dog{
}
class Puppy{
}
class Animal{
}
class Student{
}
class Test
{
    void m1(Dog d,Puppy p,Animal a)
    {
        System.out.println("m1 method");
        System.out.println(d+" "+p+" "+a);
    }
    static Student m2()
    {
        System.out.println("m2 method");
        return new Student();
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1(new Dog(),new Puppy(),new Animal());

        Student s = Test.m2();
        System.out.println("Return value="+s);
    }
}

```

Creation of object with eager & lazy instantiation

1) Eager object creation : creation of object assigning to reference variables.

2) Lazy object creation : first declaration later instantiation.

```

class Test
{
    void wish()
    {
        System.out.println("Good Morning");
    }
    public static void main(String[] args)
    {
        //Eager object creation approach
        Test t = new Test();
        t.wish();
        //Lazy object creation approach
        Test t1;
        ::::::::::::::::::::
        t1=new Test();
        t1.wish();
    }
}

```

problem : In below example in every method we are creating object

```
class A
{
    void disp() { }
}

class Test
{
    void m1()
    {
        A a = new A();
        a.disp();
    }

    void m2()
    {
        A a = new A();
        a.disp();
    }

    void m3()
    {
        A a = new A();
        a.disp();
    }
}
```

Solution: only one time we are creating object at instance level using multiple times.

Instance data solution

```
class A{
    void disp(){}
}

class Test
{
    A a=null;
    void m1()
    {
        a = new A();
        a.disp();
    }

    void m2()
    {
        a.disp();
    }

    void m3()
    {
        a.disp();
    }
}
```

Static Data solution

```
class A{
    void disp(){}
}

class Test
{
    static A a=null;
    static void m1()
    {
        a = new A();
        a.disp();
    }

    static void m2()
    {
        a.disp();
    }

    static void m3()
    {
        a.disp();
    }
}
```

Advantages of constructors

- ✓ Constructors are used to write the logics these logics are executed during object creation.
- ✓ Constructors are used to initialize the values to instance variables during object creation (Constructor is a special method used to initialize the object).

Case 1: Problem: in below example we are passing duplicate parameter. It is not recommended.

```
class RectangleEx
{
    void area(int l,int b)
    {
        System.out.println(l*b);
    }
    void volume(int l,int b,int h)
    {
        System.out.println(l*b*h);
    }
    void perimeter(int l,int b)
    {
        System.out.println(2*(l+b));
    }
    public static void main(String[] args)
    {
        RectangleEx r = new RectangleEx();
        r.area(3,4);
        r.volume(3,4,5);
        r.perimeter(3,4);
    }
}
```

Case 2: To overcome above limitation to initialize the values during object creation use constructor.

```
class RectangleEx
{
    int l,b,h;
    RectangleEx() //cons executed during object creation to initialize the data
    {
        l=2;
        h=3;
        b=4;
    }
    void area()
    {
        System.out.println(l*b);
    }
    void volume() { System.out.println(l*b*h); }
    void perimeter() { System.out.println(2*(l+b)); }
    public static void main(String[] args)
    {
        RectangleEx r = new RectangleEx();
        r.area();
        r.volume();
        r.perimeter();
    }
}
```

When we create only one object the above example is good...

Case 3: when we create multiple objects for every object same constructor(0-argument constructor) executed initialize the same values.

```
public static void main(String[] args)
{
    RectangleEx r1 = new RectangleEx();
    r1.area();
    r1.volume();
    r1.perimeter();

    RectangleEx r2 = new RectangleEx();
    r2.area();
    r2.volume();
    r2.perimeter();
}
```

Case 4: use parameterized constructor to initialize different values to different objects.
If it is a parameterized constructor: every object can pass different values.

```
class RectangleEx
{
    int l,b,h;
    //constructor executed during object creation to initialize the data
    RectangleEx(int l,int b,int h)    //local variables
    {
        //conversion of local to instance variable
        this.l=l;
        this.b=b;
        this.h=h;
    }
    void area()
    {
        System.out.println(l*b);
    }
    void volume()
    {
        System.out.println(l*b*h);
    }
    void perimeter()
    {
        System.out.println(2*(l+b));
    }
    public static void main(String[] args)
    {
        RectangleEx r1 = new RectangleEx(1,2,3);
        r1.area();
        r1.volume();
        r1.perimeter();

        RectangleEx r2 = new RectangleEx(4,5,6);
        r2.area();
        r2.volume();
        r2.perimeter();
    }
}
```

Case 5: Assignment

```

class Student
{
    during object creation initialize the student values : id name marks
    void printGrade()
    {
        marks>70 : A grade
        marks<70 : B grade
    }
    void printDetails()
    {
        print all student details : id name marks
    }
    public static void main(String[] args)
    {
        create the two students object by passing parameters.
        call printGradce() method
        call print Details() method
    }
}

```

Object creation parts: Every object creation having three parts.

1) Declaration

```

Test t;
Student s;
Emp e;

```

2) Instantiation: (object creation)

```

new Test();
new Student();
new Emp();

```

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

```

new Test(10,20);
new Student("ratan",111);
new Emp(111,"ratan",1000);

```

Primitive variable vs. reference variable:

```

int a=10;           //a is primitive variable
Test t = new Test(); //t is reference variable
➤ a : is variable of primitive type such as int,char,double,boolean...etc
➤ t : is reference variable & it is the memory address of object.

```

Copy the values from one object to another object:-

There are three ways to copy the values from one object to another object.

- 1) Assigning values of one object to another object without using constructor.
- 2) Copy the values of one object to another object by using constructor.
- 3) By using clone () method of Object class.

ex-1: Assigning values of one object to another object without using constructor.

```
class Emp
{
    int eid;
    String ename;
    Emp(int eid, String ename)
    {
        this.eid=eid;
        this.ename=ename;
    }
    Emp(){}
    void disp()
    {
        System.out.println("Emp id="+eid);
        System.out.println("Emp name="+ename);
    }
    public static void main(String[] args)
    {
        Emp e1 = new Emp(111, "ratan");
        Emp e2 = new Emp();
        e2.eid=e1.eid;
        e2.ename=e1.ename;
        e1.disp();
        e2.disp();
    }
}
```

Example 2: Assigning values of one object to another object by using constructor.

```
class Emp
{
    int eid;
    String ename;
    Emp(int eid, String ename)
    {
        this.eid=eid;
        this.ename=ename;
    }
    Emp(Emp e)
    {
        eid=e.eid;
        ename=e.ename;
    }
    void disp()
    {
        System.out.println("Emp id="+eid);
        System.out.println("Emp name="+ename);
    }
    public static void main(String[] args)
    {
        Emp e1 = new Emp(111, "ratan");
        Emp e2 = new Emp(e1);
        e1.disp();
        e2.disp();
    }
}
```

Methods Vs. constructors: Main focus on concept but not differences.

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

Instance Blocks:-

- Instance blocks are used to write the logics of projects & these logics are executed during object creation just before constructor execution.

Syntax:

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

- Instance blocks execution depends on object creation it means if we are creating 10 objects 10 times instance blocks are executed.
- Inside the class it is possible to declare the more than one instance block the execution order is top to bottom.

Example:

- ✓ During compilation the compiler will copy instance blocks code in every constructor.
- ✓ The constructor logics are specific to object but instance block logics are common for all objects.

//Application before compilation (.java)

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

//Application after compilation (.class)

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

Example:-

- ✓ Instance block execution depends on object creation but not constructor exaction.
- ✓ In below example two constructors are executing but only one object is creating hence only one time instance block is executed.
- ✓ Inside the constructor when we declare this or super keyword in this instance block is not copied in constructor.

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

Example:-

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

```
class Emp
{
    int eid;
    {
        eid=111;           //instance block

        Emp() { eid=222; } //constructor

        void disp()
        {
            System.out.println("emp id="+eid);
        }
    public static void main(String[] args)
    {
        new Emp().disp();
    }
}
```

Case 1:- When we declare instance block & instance variable the execution order is top to bottom.

In below example instance block is declared first so instance block is executed first.

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

    int a=m1();                                         //instance variables

    int m1()
    {
        System.out.println("m1() method called by variable");
        return 100;
    }
    public static void main(String[] args)
    {
        new Test();
    }
}

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

case 2:- When we declare instance block & instance variable the execution order is top to bottom.

In below example instance variable is declared first so instance block is executed first.

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

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

Static block:

- ✓ Static blocks are used to write the logics these logics are executed during .class file loading time.
- ✓ In java .class file is loaded only one time hence static blocks are executed only once per class.

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

- ✓ In class it is possible to write more than one static blocks but the execution order is top to bottom.

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

Ex :

```
class Test
{
    static { System.out.println("static block-1"); }
    static { System.out.println("static block-2"); }
    public static void main(String[] args)
    {
    }
}
```

Example :-

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

Example :-

up to 1.6 version **Valid** : it is possible to execute static blocks without using main method.

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

From 1.7 version **Invalid** : it is not possible to execute static blocks without using main method.

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

Example:-The .class file loaded into memory in three different ways

1. When we execute the class by using java command the static blocks are executed but in this case main method is mandatory.
2. When we create the object the class loaded , the class static blocks are executed but in this case main method is mandatory.
3. When we load the file by using forName() method, the static blocks are executed but in this case main method is mandatory.

File 1:Demo.java

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

File 2: Test.java

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

Example:- Static blocks are used to initialize static variables.

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

Java class concept interview questions

1. What is the purpose of the variables?
2. How many types of variables in java?
3. What are the memory areas in java?
4. What is the scope of local variables & instance & static?
5. Instance variables when memory allocated & when destroyed?
6. What are the areas in java?
7. What is the purpose of instance & static variables?
8. How to access the instance members & static members?
9. What do you mean by operator overloading & java supporting or not?
10. Is it possible to declare the instance variables inside the instance method or not?
11. Is it possible to declare the static variables inside the static method or not?
12. How many types of methods in java?
13. Define method signature?
14. For java methods return type mandatory or optional?
15. How to take the input from the keyboard?
16. What do you mean by method recursion & java supporting or not?
17. Java support inner methods or not?
18. When we will get the error message : StackOverflowError
19. When we will get error message : missing return statement?
20. When we will get the error message : method already defined in class?
21. Who creates stack memory & who destroyed stack memory?
22. What are the rules to declare the constructor?
23. How many types of constructors in java?
24. What are the advantages of constructors?
25. What is the difference between method & constructor?
26. What do you mean by default constructor & who generate default constructor?
27. Inside the class I declared 1-arg constructor then default constructor generated or not?
28. What is the difference between named object & name less object?
29. How to call the current class constructor?
30. When we will get error message : this keyword must be first statement in constructor?
31. One constructor is able to call more than one constructor or not?
32. Define the words : declaration & instantiation & initialization?
33. User defined 0-arg constructor is default constructor or not?
34. What is the purpose of the instance block?
35. Executed 4-constructors by calling but I created only object how many times ins blocks executed?
36. What is the difference between instance block & constructor?
37. What is he purpose the static block?
38. What is the difference between instance block & static block?
39. How many ways are there to load the .class into memory?
40. To execute the static block inside the class main method mandatory or not?
41. Once I created 5 objects how many times instance blocks & static blocks executed?

***** Thank you *****

Operators in java

Operator is a symbol used to perform the operation; there is different type of operators in java ,

- ✓ *Unary Operator*
- ✓ *Arithmetic Operator*
- ✓ *Relational Operator*
- ✓ *Bitwise Operator*
- ✓ *Logical Operator //conditional operator*
- ✓ *Ternary Operator //conditional operator*
- ✓ *Assignment Operator.*

Operator Type	Category	Precedence
Unary	postfix	<i>expr++ expr--</i>
	prefix	<i>++expr --expr +expr -expr ~ !</i>
Arithmetic	multiplicative	<i>* / %</i>
	additive	<i>+ -</i>
Shift	shift	<i><< >> >>></i>
Relational	comparison	<i>< > <= >= instanceof</i>
	equality	<i>== !=</i>
Bitwise	bitwise AND	<i>&</i>
	bitwise exclusive OR	<i>^</i>
	bitwise inclusive OR	<i> </i>
Logical	logical AND	<i>&&</i>
	logical OR	<i> </i>
Ternary	ternary	<i>? :</i>
Assignment	assignment	<i>= += -= *= /= %= &= ^= = <<= >>= >>>=</i>

Unary operator:

Operator	Description
+	Unary plus operator; indicates positive value (numbers are positive without this, however)
-	Unary minus operator; negates an expression
++	Increment operator; increments a value by 1
--	Decrement operator; decrements a value by 1
!	Logical complement operator; inverts the value of a boolean

a++ : post increment : print the value then increment
++a : pre increment : increment the value then print
a-- : post decrement : print the value then decrease
-a : pre decrement : decrease the value then print

ex: class Test

```
{
    public static void main(String[] args)
    {
        int a=10;
        System.out.println(a++);
        System.out.println(++a);
        System.out.println(a--);
        System.out.println(--a);
    }
}
```

ex: class Test

```
{
    public static void main(String[] args)
    {
        int a=10;
        System.out.println(a++ + ++a );
        System.out.println(a++ - ++a );
        System.out.println(a-- + --a );
        System.out.println(a-- - --a );

        System.out.println(a++ + ++a + --a + a--);
        System.out.println(a++ - ++a - --a - a--);
    }
}
```

ex: class Test

```
{
    public static void main(String[] args)
    {
        byte b1=-10;
        byte b2=+10;
        System.out.println(b1);
        System.out.println(b2);
        boolean b=true;
        System.out.println(!b);
    }
}
```

Arithmetic Operators:

Operator	Description
+	Additive operator (also used for String concatenation)
-	Subtraction operator
*	Multiplication operator
/	Division operator
%	Remainder operator

Division (/) : Divides the first operand by the second.

Modulus (%) : Returns the remainder when first operand is divided by the second.

*Operator precedence : Multiplicative : * / % Addition : + -*

ex-1: class Test

```
{     public static void main(String[] args)
{         int a=10,b=20;
        System.out.println(a+b);
        System.out.println(a-b);
        System.out.println(a*b);
        System.out.println(a/b);
        System.out.println(a%b);
    }
```

ex-2: *class Test*

```
{     public static void main(String[] args)
{         System.out.println(10*10/5+3-1*4/2);
        System.out.println(10+10/5+3*2*4/2-20);
        System.out.println(10-10*5/2+3-10*4/2);
    }
```

ex 3: By using + operator we can perform concatenation also. In java string with concatenation with any data, the return value will be String.

dirty data, class Test

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("ratan"+"anu");
        System.out.println(10+20+"ratan"+"durga");
        System.out.println(10+"ratan"+20+"durga");
        System.out.println(10+"ratan"+3*10/2+"durga");
    }
}
```

10+"ratan"+3*10/2-1+"duraq" ; tell me the output: if you said correct output your perfect in java.

Relational operator:**Java Equality and Relational Operators**

Operator	Description	Example
<code>==</code>	equal to	<code>5 == 3</code> is evaluated to <code>false</code>
<code>!=</code>	not equal to	<code>5 != 3</code> is evaluated to <code>true</code>
<code>></code>	greater than	<code>5 > 3</code> is evaluated to <code>true</code>
<code><</code>	less than	<code>5 < 3</code> is evaluated to <code>false</code>
<code>>=</code>	greater than or equal to	<code>5 >= 5</code> is evaluated to <code>true</code>
<code><=</code>	less than or equal to	<code>5 <= 5</code> is evaluated to <code>true</code>

Observation : not possible to use relational operators on **String** type.

```
System.out.println(10>20);           Valid
System.out.println(10<20);           Valid
System.out.println("ratan"<"anu");   error: bad operand types for binary operator '<'
```

<i>op</i>	<i>meaning</i>	<i>true</i>	<i>false</i>
<code>==</code>	<i>equal</i>	<code>2 == 2</code>	<code>2 == 3</code>
<code>!=</code>	<i>not equal</i>	<code>3 != 2</code>	<code>2 != 2</code>
<code><</code>	<i>less than</i>	<code>2 < 13</code>	<code>2 < 2</code>
<code><=</code>	<i>less than or equal</i>	<code>2 <= 2</code>	<code>3 <= 2</code>
<code>></code>	<i>greater than</i>	<code>13 > 2</code>	<code>2 > 13</code>
<code>>=</code>	<i>greater than or equal</i>	<code>3 >= 2</code>	<code>2 >= 3</code>

Bitwise operators:

Operator	Description
<code>~</code>	Bitwise Complement
<code><<</code>	Left Shift
<code>>></code>	Right Shift
<code>>>></code>	Unsigned Right Shift
<code>&</code>	Bitwise AND
<code>^</code>	Bitwise exclusive OR
<code> </code>	Bitwise inclusive OR

A	B	A B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

ex: `class Test`

```

{   public static void main(String[] args)
    {
        int a=10;
        int b=31;
        System.out.println(a&b);
        System.out.println(a/b);
        System.out.println(a^b);
    }
}

```

0000 1010 : 10

0001 1111 : 31

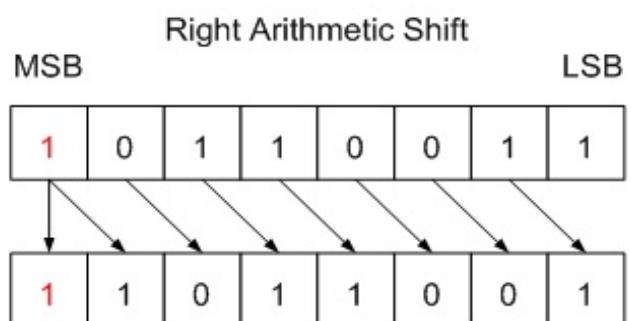
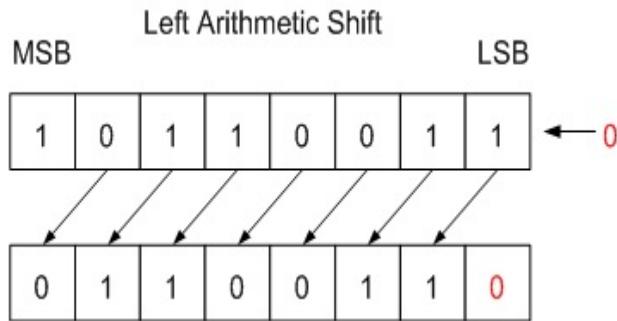
0000 1010 : & : 10

0001 1111 : / : 31

0001 0110 : ^ : 21

Write the output for below template:

```
int a = 3; // 0 + 2 + 1 or 0011 in binary
int b = 6; // 4 + 2 + 0 or 0110 in binary
int c = a | b;
int d = a & b;
int e = a ^ b;
int f = (~a & b) | (a & ~b);
```



ex: class Test

```
{
    public static void main(String[] args)
    {
        System.out.println(31<<1);
        System.out.println(31<<2);

        System.out.println(31>>1);
        System.out.println(31>>2);
    }
}
```

31<<1	0001 1111	:	31
0011 1110	:	62	
31<<2	0001 1111	:	31
0111 1100	:	124	
31>>1	0001 1111	:	31
0000 0111	:	15	
31>>2	0001 1111	:	31
0000 0111	:	7	

System.out.println(40 >> 1) == 2 == 2^1 == 40 / 2 == 20

System.out.println(40 >> 2) == 2 * 2 == 2^2 == 40 / 4 == 10

System.out.println(40 >> 3) == 2 * 2 * 2 == 2^3 == 40 / 8 == 5

Logical operators:

The bitwise & always checking both conditions then only decide the result.

In logical operator second condition checking completely depends on first condition result.

The logical **&&** operator does not check second condition if the first condition is false. It checks the second condition if the first condition is true.

The logical **||** operator does not check the second condition if the first condition true. It will check the second condition if the first condition false.

Note : The **&&** and **||** operators perform Conditional-AND and Conditional-OR operations on two Boolean expressions. These operators exhibit "short-circuiting" behavior, which means that the second operand is evaluated only if needed.

```
ex:    int a=10,b=20,c=30;
        System.out.println(a>b && a<c);      //false
        System.out.println(a>b & a<c);       //false
```

ex-1 : class Test

```
{     public static void main(String[] args)
{     int a=10,b=20,c=30;
    System.out.println(a>b && a++<c);
    System.out.println(a);
    System.out.println(a>b & a++<c);
    System.out.println(a);
}
```

ex-2: class Test

```
{     public static void main(String[] args)
{     int a=10,b=6,c=30;
    System.out.println(a > b || a < c);      //true || true = true
    System.out.println(a > b | a < c);       //true | true = true
    System.out.println(a > b || a++ < c);    //true || true = true
    System.out.println(a);                  //10 because second condition is not checked
    System.out.println(a > b | a ++ < c);   //true | true = true
    System.out.println(a);                  //11 because second condition is checked
}
```

ex-3: class Test

```
{     public static void main(String[] args)
{     int a=10,b=20,c=30;
    System.out.println(a<b || a++<c);
    System.out.println(a);
    System.out.println(a<b | a++<c);
    System.out.println(a);
}
```

Ternary operator : (?)

Java Ternary operator is used as one line replacement for if-then-else statement.

variable = Expression ? result1 : result2
If the Expression is true, result1 is assigned to variable.
If the Expression is false, result2 is assigned to variable.

ex-1 :

```
class Test{
    public static void main(String args[])
    {
        int a=16;
        int b=5;
        int min=(a < b) ? a : b;
        System.out.println(min);
    }
}
```

ex-2 : class ConditionalDemo2 {

```
public static void main(String[] args){
    int value1 = 1;
    int value2 = 2;
    boolean result = value1>value2? true : false;
    System.out.println(result);
}
```

ex-3 : class Test

```
{    public static void main(String[] args)
{        String uname = "ratan";
        String upwd = "anu";
String status = uname.equals("ratan")&&upwd.equals("anu")?"login success":"login fail";
        System.out.println(status);
    }
}
```

ex 4: class Test

```
{    public static void main(String[] args)
{        String str = "Australia";
        String data = str.contains("A") ? "Str contains 'A'" : "Str doesn't contain 'A'";
        System.out.println(data);
    }
}
```

Assignment operator :

<u>Operator</u>	<u>Example</u>	<u>Equivalent To</u>
<code>+=</code>	<code>x += y</code>	<code>x = x + y</code>
<code>-=</code>	<code>x -= y</code>	<code>x = x - y</code>
<code>*=</code>	<code>x *= y</code>	<code>x = x * y</code>
<code>/=</code>	<code>x /= y</code>	<code>x = x / y</code>
<code>%=</code>	<code>x %= y</code>	<code>x = x % y</code>

ex: `class Test`
`{ public static void main(String[] args)`
`{ int a=10,b=20;`
`a+=5;`
`b-=5;`
`System.out.println(a+" "+b);`
`}`

ex: `class Test`
`{ public static void main(String[] args)`
`{ int a=10;`
`a+=1;`
`System.out.println(a);`
`a-=2;`
`System.out.println(a);`
`a*=3;`
`System.out.println(a);`
`a/=3;`
`System.out.println(a);`
`}`

Observation :**Invalid : error**

```
byte a=10;
byte b=20;
a=a+b;
System.out.println(b);
```

Valid : after type conversion

```
byte a=10;
long b=20;
b=(byte)(a+b);
System.out.println(b);
```

Observation :

```
byte + byte = int
short + short = int
byte + short = int
int + int = int
int + long = long
long + byte = long
long+long = long
```

```
float +float = float
float + int = float
float + byte = float
float + double = double
double + double = double
double + int = int
***** Operators Completed *****
```

[OOPs] Object-Oriented Programming System

- ✓ The first object oriented programming is : **Simula, smalltalk**
- ✓ Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.
- ✓ In object oriented programming languages everything represented in the form of object.
- ✓ Object is real world entity that has state & behavior.
ex: such as pen, chair, table, house....etc.

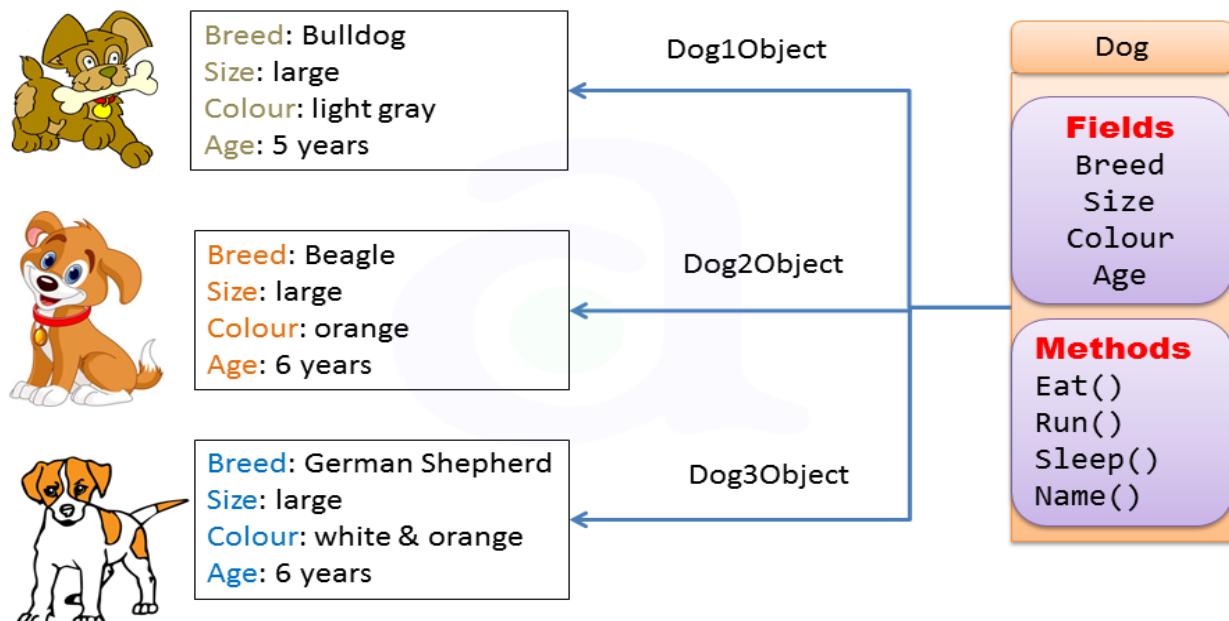
Every object contains three characteristics,

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

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

Class vs. Object:

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



Dog is a single class but here we created three objects means three different memories are created.

Inheritance:

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

Application code without inheritance

```

class A
{
    void m1(){}
    void m2(){}
}

class B
{
    void m1(){}
    void m2(){}
    void m3(){}
    void m4(){}
}

class C
{
    void m1(){}
    void m2(){}
    void m3(){}
    void m4(){}
    void m5(){}
    void m6(){}
}

```

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

Application code with inheritance

```

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

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

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

```

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

Note: Child support all features of parent class & support extra features.

Object creation of parent & child classes:

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

- ✓ When we create object of parent class it is possible to access only parent specific methods.

A a=new A();

a.m1(); a.m2();

- ✓ When we create object of child class it is possible to access both parent & child specific methods.

B b=new B();

b.m1();b.m2(); b.m3();b.m4();

C c=new C();

c.m1(); c.m2(); c.m3(); c.m4(); c.m5(); c.m6();

Note: it is always recommended to create the object of child classes.

Important observations: (interview questions)

```
class A extends Object{
}
class B extends A{
}
class C extends B{
}
```

- ✓ The default super class in java is : **Object class**
- ✓ The root class of all java classes is : **Object class**
- ✓ Every java class contains parent class except : **Object class**
- ✓ In java every class is child class of object either directly (A) or indirectly (B,C).
- ✓ Object class present in **java.lang** package and it contains 11 methods & all java classes able to use these 11 methods because Object class is root class of all java classes.
- ✓ To check the predefined support use javap command.

Javap full-class-name
Javap java.lang.Object

E:\>**javap java.lang.Object**

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

Preventing inheritance: it means preventing child class.

You can prevent sub class creation by using **final** modifier.

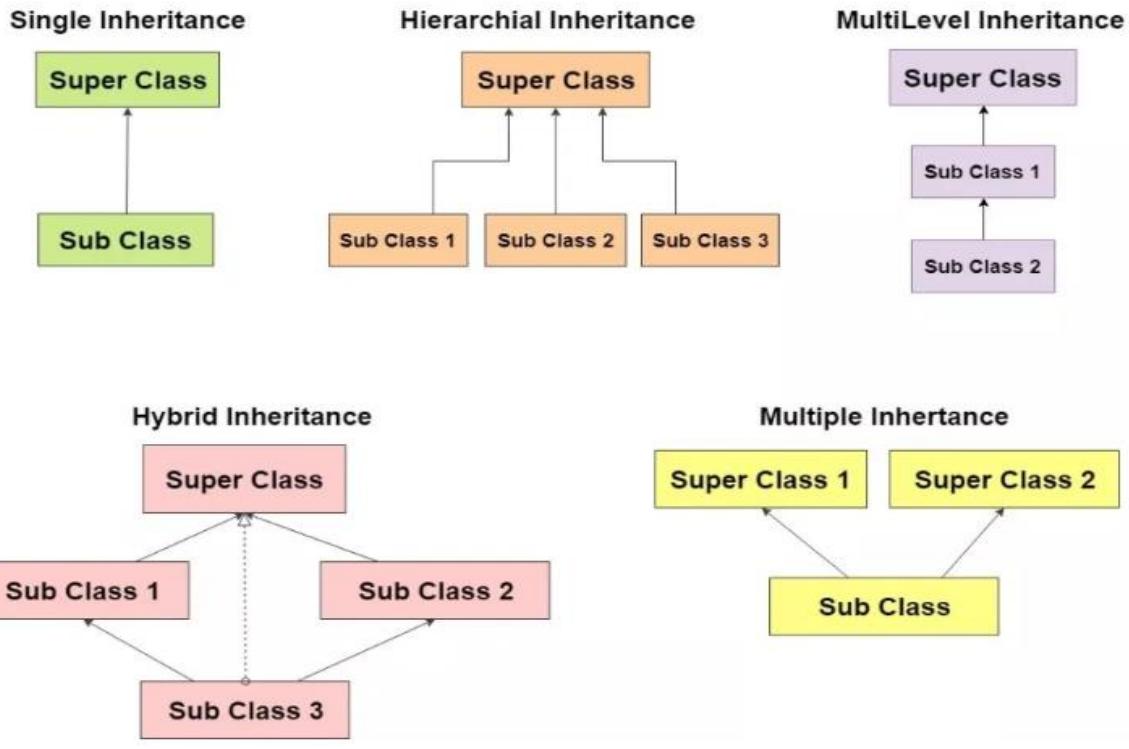
If a class declared as **final** we can't create sub class.

```
final class Parent{
}
class Child extends Parent{
}
compilation error: cannot inherit from final Parent
```

Types of inheritance:

Single Inheritance		public class A {} public class B extends A {}
Multi Level Inheritance		public class A { } public class B extends A { } public class C extends B { }
Hierarchical Inheritance		public class A { } public class B extends A { } public class C extends A { }
Multiple Inheritance		public class A { } public class B { } public class C extends A,B { } } // Java does not support multiple inheritance

Hybrid is combination of multiple & hierarchical, java not supporting hybrid.



ex: Single inheritance

```

class Animal
{
    void eat()
    {
        System.out.println("eating... ");
    }
}
class Dog extends Animal
{
    void bark()
    {
        System.out.println("barking... ");
    }
}
class TestInheritance
{
    public static void main(String args[])
    {
        Dog d=new Dog();
        d.bark();
        d.eat();
    }
}

```

ex: Multilevel inheritance

```

class Animal {
    void eat() {
        System.out.println("eating... ");
    }
}
class Dog extends Animal {
    void bark() {
        System.out.println("barking... ");
    }
}
class BabyDog extends Dog {
    void weep() {
        System.out.println("weeping... ");
    }
}
class TestInheritance {
    public static void main(String args[]) {
        BabyDog d = new BabyDog();
        d.weep();
        d.bark();
        d.eat();
    }
}

```

ex : Hierarchical inheritance.

```
class Animal {
    void eat() {
        System.out.println("eating... ");
    }
}
class Dog extends Animal {
    void bark() {
        System.out.println("barking... ");
    }
}
class Cat extends Animal {
    void sleep() {
        System.out.println("Sleeping... ");
    }
}
class TestInheritance {
    public static void main(String args[]) {
        Cat c = new Cat();
        c.eat();
        c.sleep();

        Dog d = new Dog();
        d.eat();
        d.bark();
    }
}
```

ex : multiple inheritance not supported by java

```
class A{
    void msg(){System.out.println("Hello");}
}
class B{
    void msg(){System.out.println("Welcome");}
}
class C extends A,B
{
    public static void main(String args[])
    {
        C obj=new C();
        obj.msg(); //Now which msg() method would be invoked? : Error
    }
}
```

Instanceof operator:

- ✓ It is used check the type of the object at runtime & it returns Boolean value as a return value.

Syntax: reference-variable instanceof class-name

- ✓ To use the instanceof operator the class name & reference variable must have some relationship either parent to child or child to parent otherwise compiler will generate error message.
- ✓ If the relationship is,
 - child to parent it returns **true**
 - Parent to child it return **false**.
 - No relation compiler generates error message incompatible types.

ex 1:

```
class Animal { }
class Dog extends Animal{ }
class Test
{
    public static void main(String[] args)
    {
        Animal a = new Animal();
        Dog d = new Dog();
        Object o = new Object();

        System.out.println(d instanceof Animal);           //true
        System.out.println(a instanceof Object);           //true
        System.out.println(a instanceof Dog);              //false
        System.out.println(o instanceof Animal);           //false
    }
}
```

ex 2:

```
class Parent{ }
class Child1 extends Parent{}
class Child2 extends Parent{}
class Test
{
    public static void main(String[] args)
    {
        Parent p =new Parent();
        Child1 c1 = new Child1();
        Child2 c2 = new Child2();

        System.out.println(c1 instanceof Parent);          //true
        System.out.println(c2 instanceof Parent);          //true
        System.out.println(p instanceof Child1);           //false
        System.out.println(p instanceof Child2);           //false

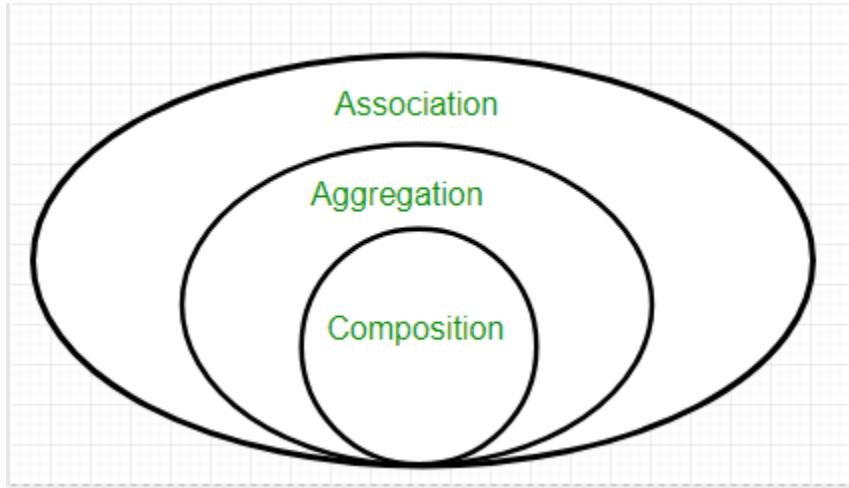
        p = c1;
        System.out.println(p instanceof Child1);           //true
        System.out.println(p instanceof Child2);           //false
        p = c2;
        System.out.println(p instanceof Child1);           //false
        System.out.println(p instanceof Child2);           //true
    }
}
```

Association:

The name of an association specifies the nature of the relationship between objects.
Association can be one-to-one, one-to-many, many-to-one, many-to-many.

There are two forms of association

1. Composition
2. Aggregation



Aggregation:

- Class A has instance of class B is called aggregation.
- Aggregation is an association that represents part of a whole relationship where a part can exist without a whole. It has a weaker relationship.
- Take the relationship between teacher and department. A teacher may belong to multiple departments hence teacher is a part of multiple departments but if we delete department object teacher object will not destroy.

Example-1:

File-1 : Address.java

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

File-2 : Heroin.java

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

        // new Heroin("anushka",30,new Address("india","banglore",111)).display();

        Address a2 = new Address("US","california",333);
        Heroin h2 = new Heroin("AJ",40,a2);
        h2.display();

        // new Heroin("AJ",40,new Address("US","california",333)).display();
    }
}
```

Example-2:**File 1: Test1.java**

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

File 2: Test2.java

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

File 3 : Test3.java

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

File -4 : MainTest.java

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

Composition:

- Strong aggregation is called composition.
- Composition is an association represents a part of a whole relationship where a part cannot exist without a whole. If a whole is deleted then all parts are deleted. It has a stronger relationship.
- For example, if order HAS-A line-items, then an order is a whole and line items are parts. If an order is deleted then all corresponding line items for that order should be deleted..
- 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.

ex:

Marks.java

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

student.java

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

        Marks m2 = new Marks(100,200,300);
        Student s2 = new Student(m2,"anu",222);
        s2.display();
        new Student(new Marks(100,200,300),"anu",222).display();
    }
}
```

Example-2 :

File 1: Parents.java

```
class Parents
{
    Declare fname, mname
    Declare the constructor to initialize the values during object creation.
}
```

File 2: Marks.java

```
class Marks
{
    Decalre m1 m2 m3 of int type
    Declare the constructor to initialize the values during object creation.
}
```

File 3 : Address.java

```
class Address
{
    Declare dno street
    Declare the constructor to initialize the values during object creation.
}
```

File 4 : Student.java

```
class Student
{
    int sid;
    String sname;
    Parents p;
    Marks m;
    Address addr;
```

Declare the constructor to initialize the values during object creation.

Decalre the display() method to print the data

Declare the main method to create the object of Student class & Access the display() method

```
}
```

Super keyword:

- ✓ “**this**” keyword is used to represent current class object, “**super**” represent super class object.
- ✓ Inside the static area both **this,super** keywords are not allowed.
 1. Super class variables.
 2. Super class methods.
 3. Super class constructors.
 4. Super class instance blocks.
 5. Super class static blocks.

super class variables:

case 1: variable names are different hence **this,super** keywords not required.

```
class Parent
{
    int a=10,b=20;
}
class Child extends Parent
{
    int i=100,j=200;
    void add(int x,int y)
    {
        System.out.println(x+y);
        System.out.println(i+j);
        System.out.println(a+b);
    }
    public static void main(String[] args)
    {
        new Child().add(1000,2000);
    }
}
```

case 2: variable names are same hence **this,super** keywords required.

```
class Parent
{
    int a=10,b=20;
}
class Child extends Parent
{
    int a=100,b=200;
    void add(int a,int b)
    {
        System.out.println(a+b);
        System.out.println(this.a+this.b);
        System.out.println(super.a+super.b);
    }
    public static void main(String[] args)
    {
        new Child().add(1000,2000);
    }
}
```

super class methods calling:

```

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

class Child extends Parent
{
    void m1()
    {
        System.out.println("child class m1() method");
    }

    void m2()
    {
        this.m1();           //this keyword is optional
        super.m1();
    }

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

```

super class constructors calling:

case 1: To call the current class constructors use **this** keyword

To call super class constructor use **super** keyword.

this()	----> current class 0-arg constructor calling
this(10)	----> current class 10-arg constructor calling
super()	----> super class 0-arg constructor calling
super(10)	----> super class 1-arg constructor calling

```

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

class Child extends Parent
{
    Child()
    {
        this(10);
        System.out.println("Child 0-arg constructor");
    }

    Child(int a)
    {
        super();
        System.out.println("child 1-arg constructor-->" + a);
    }

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

```

Case 2: Inside the constructor super keyword must be first statement otherwise compiler generates error message “**call to super must be first line in constructor**”.

```
Child(int a)
{
    System.out.println("child 1-arg constructor-->" + a);
    super();
}
```

Case 3: Inside the constructor it is possible to use either this keyword or super keyword but,

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

Invalid

```
Child()
{
    super(10);
    super();
}
```

Invalid

```
Child()
{
    this(10);
    this();
}
```

Invalid

```
Child()
{
    this(10);
    super();
}
```

Case 4: In below example parent class default constructor is executed that is provided by compiler.

```
class Parent
{
    // default constructor
}

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

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

Case 5: compiler generate default constructor with super keyword.

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

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

Case 6:

- ✓ Inside the constructor if we are not declaring this or super keyword then compiler generate super keyword at first line of the constructor.
- ✓ The compiler generated super keyword is always 0-arg constructor calling.

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

Observation -1: what is the output? If you say correct : you are perfect in java

```
class Parent
{
    Parent(int a){ System.out.println("parent 1-arg cons"); }
}
class Child extends Parent
{
    Child()
    {
        System.out.println("Child 0-arg constructor");
    }
    Child(int a)
    {
        super(10);
        System.out.println("child 1-arg cons");
    }
    public static void main(String[] args)
    {
        new Child();
        new Child(10);
    }
}
```

Super class instance blocks & static blocks :

Instance blocks are executed during object creation: In parent and child relationship first parent class instance blocks are executed then child class instance blocks are executed.

Static blocks are executed only once during class loading: In parent and child relationship first parent class static blocks are executed then child class static blocks are executed.

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

***** **Inheritance completed: Thank you** *****

Polymorphism:

- ✓ One functionality with different behaviors is called polymorphism.
- ✓ The ability to appear in more forms is called polymorphism.
- ✓ Polymorphism is the ability of an object to take on many forms

Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.



Same method different behaviors:

Same sleep method with two different behaviors

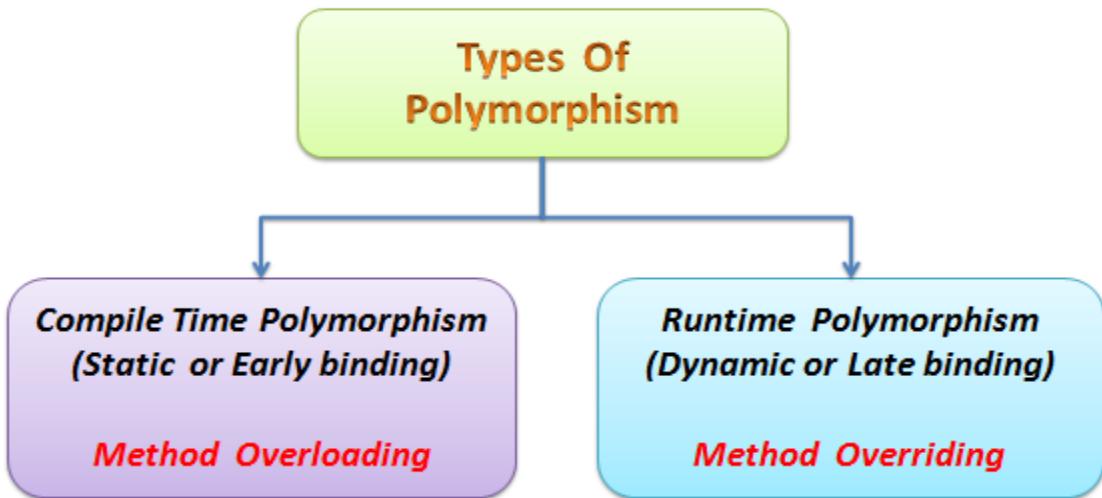
```
public static native void sleep(long)      // 1-arg
public static native void sleep(long, int)  // 2-args
```

Same wait method with three different behaviors

```
public final void wait()
public final native void wait(long)
public final void wait(long, int)
```

There are two types of polymorphism in java,

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



Compile time polymorphism [Method Overloading]:

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

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

```
void m1(int a){}
void m1(int a,int b){}
```

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

```
void m1(int a){}
void m1(char ch){}
```

To achieve overloading concept one java class sufficient. In single class It is possible to overload any number of methods in single java class.

ex: class Test

```
{      overloaded methods : same method name but different no of arguments
      void m1(int a){ }
      void m1(int a,int b){ }
```

overloaded methods : same method name & same number of arguments but diff types

```
void m1(int a){ }
void m1(char ch){ }
```

```
}
```

ex-1:

```
class Test
{      //overloaded methods
      void sum(int a)
      {
          System.out.println(a+a);
      }
      void sum(int a,int b)
      {
          System.out.println(a+b);
      }
      void sum(double d1,double d2)
      {
          System.out.println(d1+d2);
      }
      public static void main(String[] args)
      {
          Test t = new Test();
          t.sum(10);
          t.sum(10,20);
          t.sum(10.5,20.5);
      }
}
```

t.sum(10); In above example during compilation compiler is checking the method with 1-argument is available or not ,this mapping is done during compilation is called static binding.

ex-2:

- ✓ Overloaded methods can have different return types.
- ✓ While overloading the methods check the signature(methodname+parameters) but not return type.

```
class Test
{
    double m1(int a,int b)
    {
        return a+b+10.5 ;
    }
    int m1(int a)
    {
        return a+a;
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        double d = t.m1(10,20);
        System.out.println("return Value="+d);

        int x = t.m1(10);
        System.out.println("return value="+x);
    }
}
```

Observations:**case 1: valid**

```
void m1(int a)
void m1(int b,int a)
```

case 2: valid

```
void m1(int a)
void m1(char ch)
```

case 3: Invalid

```
void m1(int a)
void m1(int b)
```

case 4: valid

```
void m1(int a,char ch)
void m1(char ch,int a)
```

Without Method Overloading

```
int add2(int x, int y)
{
    return(x+y);
}
int add3(int x, int y,int z)
{
    return(x+y+z);
}
int add4(int w, int x,int y, int z)
{
    return(w+x+y+z);
}
```

With Method Overloading

```
int add(int x, int y)
{
    return(x+y);
}
int add(int x, int y,int z)
{
    return(x+y+z);
}
int add(int w, int x,int y, int z)
{
    return(w+x+y+z);
}
```

Method Overloading in Java

*Without overloading must define multiple method names: add2 add3 add4**With method overloading same name we can use with different behaviors.*

ex : It is possible to overload the methods in parent & child classes.

```
class Parent
{
    void m1(int a)
    {
        System.out.println("m1 method 1-arg");
    }
}

class Child extends Parent
{
    void m1(char ch)
    {
        System.out.println("m1 method 1-arg char");
    }

    void m1(int a,int b)
    {
        System.out.println("m1 method 2-arg");
    }

    public static void main(String[] args)
    {
        Child c = new Child();
        c.m1(10);
        c.m1(10,20);
        c.m1('a');
    }
}
```

ex: Method overloading with type promotion:

byte → short

char → Int → long → float → double

Type promotion means implicit type casting it perform from left to right

```
class Test
{
    void m1(int a,long b)
    {
        System.out.println("int,long arguments method");
    }

    void m1(float f)
    {
        System.out.println("float argument");
    }

    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1(10,20l);
        t.m1(10,20);
        t.m1((byte)10,(short)20);
        t.m1('a','b');                                // here the Unicode values are assigned
        t.m1(10.5f);
        t.m1(10);
        t.m1('a');
        t.m1((byte)10);
    }
}
```

✓ In java the numeric values are by default int values,

- To represent byte, short perform type casting
- to represent long value use L constant (small or capital L)

✓ In java the decimal values are by default double values but to represent float value use f constant.

Ex:

```

class Test
{
    void m1(int a) { System.out.println("int arguments method"); }
    void m1(float f) { System.out.println("float argument"); }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1(10);
        t.m1(10.5f);
        t.m1('a');
    }
}

```

In above example `t.m1('a');` this matches both int & float arguments but the type promotion done to immediate level.

Types of overloading:

There are three types of overloading in java,

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

Constructor Overloading:

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

```

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

```

Operator overloading: One operator with different behaviors is called Operator overloading .

Java is not supporting operator overloading but only one overloaded in java language is '+'.

- If both operands are integer then "+" performs addition.
- If at least one operand is String then "+" perform concatenation.

```

class Test
{
    public static void main(String[] args)
    {
        System.out.println(10+20);           //30      [addition]
        System.out.println("ratan"+ "anu");  //ratananu [concatenation]
    }
}

```

Runtime polymorphism [Method Overriding]:

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

ex:

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

If we are not overriding marry method then parent class marry executed the output is :

```
E:\>java Child
money+land+house+gold..... 
black girl
```

If we are overriding marry method then our class marry method executed output is :

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

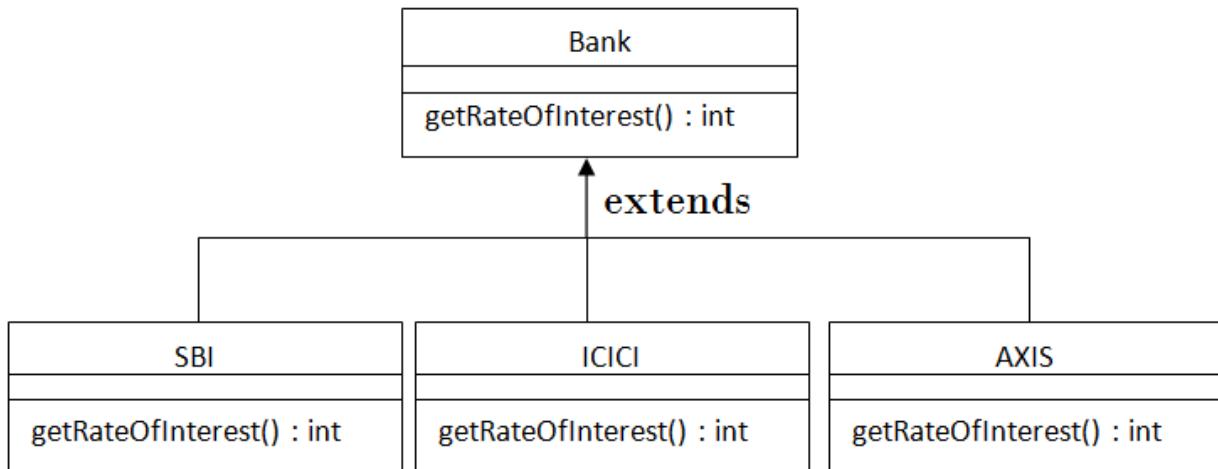
ex-2:

```

class Bank{
    int getRateOfInterest(){return 0;}
}
class SBI extends Bank{
    int getRateOfInterest(){return 7;}
}
class ICICI extends Bank{
    int getRateOfInterest(){return 8;}
}
class AXIS extends Bank{
    int getRateOfInterest(){return 9;}
}

class Test{
    public static void main(String args[]){
        SBI s=new SBI();
        ICICI i=new ICICI();
        AXIS a=new AXIS();

        System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
        System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
        System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
    }
}
E:\>java Test
SBI Rate of Interest: 7
ICICI Rate of Interest: 8
AXIS Rate of Interest: 9
  
```



While overriding methods must follow these rules:

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

Method overriding rule 1:

While overriding the overridden method signature and overriding method signature must be same. (Method signature nothing but method-name & parameters list.)

Method overriding rule-2:

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

```
class Parent
{
    void mrg(){System.out.println("very balck girl");}
} // overridden method

class Child extends Parent
{
    int mrg(){System.out.println("red girl");}
} //overriding method

error: mrg() in Child cannot override mrg() in Parent
return type int is not compatible with void
```

Method overriding rule-3:

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

```
class Parent
{
    final void mrg(){System.out.println("very balck girl");}
} // overridden method

class Child extends Parent
{
    void mrg(){System.out.println("Red Girl");}
} //overriding method

error: mrg() in Child cannot override mrg() in Parent
overridden method is final
```

Method overriding rule-4:

- ✓ While overriding methods it is possible to change the return type of overridden method & overriding methods at class level by using co-variant return type concept.
- ✓ **Co-variant return type:** The return type of overriding method is must be sub-type of overridden method return type this is called covariant return types.
- ✓ **Co-variant return type:** overridden method return type is parent type & overriding method return type is child type.

Case 1:

```

class Animal{ }
class Dog extends Animal{}
class Parent
{
    Animal mrg()
    {
        return new Animal();
    }
}
class Child extends Parent
{
    Dog mrg()
    {
        return new Dog();
    }
    public static void main(String[] args)
    {
        Child c = new Child();
        Dog d = c.mrg();
    }
}

```

Case 2: valid

```

class Parent
{
    Object mrg(){ return new Animal(); }
}
class Child extends Parent
{
    Dog mrg() { return new Dog(); }
}

```

Case 3: Valid

```

class Parent
{
    Object mrg(){ return new Animal(); }
}
class Child extends Parent
{
    Animal mrg() { return new Dog(); }
}

```

Case 4 : invalid

```

class Parent
{
    Dog mrg(){ return new Animal(); }
}
class Child extends Parent
{
    Animal mrg() { return new Dog(); }
}

```

Method overriding rule-5:

- ✓ Instance methods are bounded with object it is possible to override instance methods in java.
- ✓ Static methods are bounded with class it is not possible to override static methods in java.

```
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();
    }
}
```

- ✓ The above example seems to be overriding but it is method hiding concept.
- ✓ If a subclass defines a static method with the same signature as a static method in the super class, then the method in the subclass hides the one in the super class.
- ✓ In method overriding method execution depends on object creation but in method hiding method execution depends on class-name.

Case 1 : Both methods are instance : method Overriding : execution decided by object creation

```
class Parent
{
    void m1(){      System.out.println("parent m1()");      }
}
class Child extends Parent
{
    void m1(){      System.out.println("child m1()");      }
}
```

Case 2: Both methods are static : method hiding : execution decided by class-name

```
class Parent
{
    static void m1(){      System.out.println("parent m1()");      }
}
class Child extends Parent
{
    static void m1(){      System.out.println("child m1()");      }
}
```

Case 3: Invalid

```
class Parent
{
    static void m1(){      System.out.println("parent m1()");      }
}
class Child extends Parent
{
    void m1(){      System.out.println("child m1()");      }
}      error: m1() in Child cannot override m1() in Parent: overridden method is static
```

case 4: Invalid

```
class Parent
{
    void m1(){      System.out.println("parent m1()");      }
}
class Child extends Parent
{
    static void m1(){      System.out.println("child m1()");      }
}      error: m1() in Child cannot override m1() in Parent : overriding method is static
```

Method overriding rule : 6

In java not possible to override private methods because these methods are specific to class.

```
class Base
{
    private void fun(){}
}
class Derived extends Base
{
    private void fun(){}
}
```

Method overriding rule-7:

- ✓ In java while overriding it is possible to maintain same level(**public-public**) permission or increasing order(**default-public**) but it is not possible to decrease(**public-default**) the permission.
- ✓ In java if we are trying to decrease the permission compiler will generate error message "attempting to assign weaker access privileges"

Case 1: same level: Valid [public-public]

```
class Parent
{
    public void m1(){System.out.println("m1 method");}
}
class Child extends Parent
{
    public void m1(){System.out.println("m1 method");}
}
```

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

```
class Parent
{
    protected void m1(){System.out.println("m1 method");}
}
class Child extends Parent
{
    public void m1(){System.out.println("m1 method");}
}
```

Case3 : decreasing permission Invalid [public-protected]

```
class Parent
{
    public void m1(){System.out.println("m1 method");}
}
class Child extends Parent
{
    protected void m1(){System.out.println("m1 method");}
}
```

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

Method overriding -8:

If the super class overridden method does not declare an exception, subclass overriding method cannot declare the checked exception but it can declare unchecked exception.

Case 1: Invalid

```
import java.io.*;
class Parent
{
    void m1()
    {
    }
}
class Child extends Parent
{
    void m1()throws IOException
    {
    }
}
```

case 2: Valid

```
class Parent
{
    void m1()
    {
    }
}
class Child extends Parent
{
    void m1()throws ArithmeticException
    {
    }
}
```

Method overriding -9

If the super class method declares an exception, subclass overridden method can declare same exception, subclass exception or no exception but cannot declare parent exception.

Case 1: same type exception: Valid

```
class Parent
{
    void m1()throws ArithmeticException
    {
    }
}
class Child extends Parent
{
    void m1()throws ArithmeticException
    {
    }
}
```

Case 2: no exception : Valid

```
class Parent
{
    void m1()throws ArithmeticException
    {
    }
}
class Child extends Parent
{
    void m1()
    {
    }
}
```

Case 3: sub class exception: valid

```
class Parent
{
    void m1()throws Exception
    {
    }
}
class Child extends Parent
{
    void m1()throws ArithmeticException
    {
    }
}
```

Case 4: not parent exception : Invalid

```
class Parent
{
    void m1()throws ArithmeticException
    {
    }
}
class Child extends Parent
{
    void m1()throws Exception
    {
    }
}
```

error: m1() in Child cannot override m1() in Parent
overridden method does not throw Exception.

Final modifier:

- ✓ If a class declared as final we cannot create sub class.
- ✓ If a method declared as final we can't override that method in child class.
- ✓ If a variable declared as final we can't modify that value.

Final modifier vs. local variables:

- ✓ When we declare variable as a final it is not possible to change the value of final variable. If we are trying to change final variable compiler will generate error message.
- ✓ In java for the local variables only one modifier is applicable that is **final**.

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

error: cannot assign a value to final variable a

Final modifier vs. static variables:

In java it is not possible to make the default values as a final constants, hence must initialize the static variables only by using static blocks.

Case 1: invalid

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

error: variable a not initialized in the default constructor

Case 2: Valid : initializing static variable by using static blocks

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

Note: Final class variables are not a final but final class methods are by default final.

Final class methods are by default final because for the final class not possible to create sub-classes hence it is not possible to override that method.

Final modifier vs. instance variables:

- ✓ In java it is not possible to make the default values as a final constants, so must initialize instance variables either by using instance blocks or by using constructors but not both.

Case 1: invalid

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

error: variable a not initialized in the default constructor

Case 2: Valid: initializing instance variable by using constructor

```
class Test
{
    final int a;
    Test()
    {
        a=200;
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        System.out.println(t.a);
    }
}
```

Case 3: Valid: initializing instance variable by using instance block.

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

Case 4: Invalid: initializing instance variables by using both instance blocks & constructor.

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

error: variable a might already have been assigned

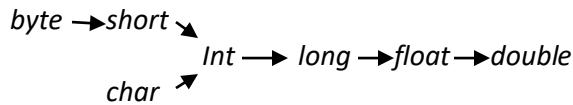
Type-conversion in java :

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

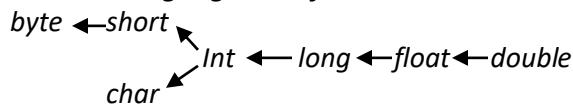
There are two types of type casting

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

Up-casting: left to right



Down-casting: right to left

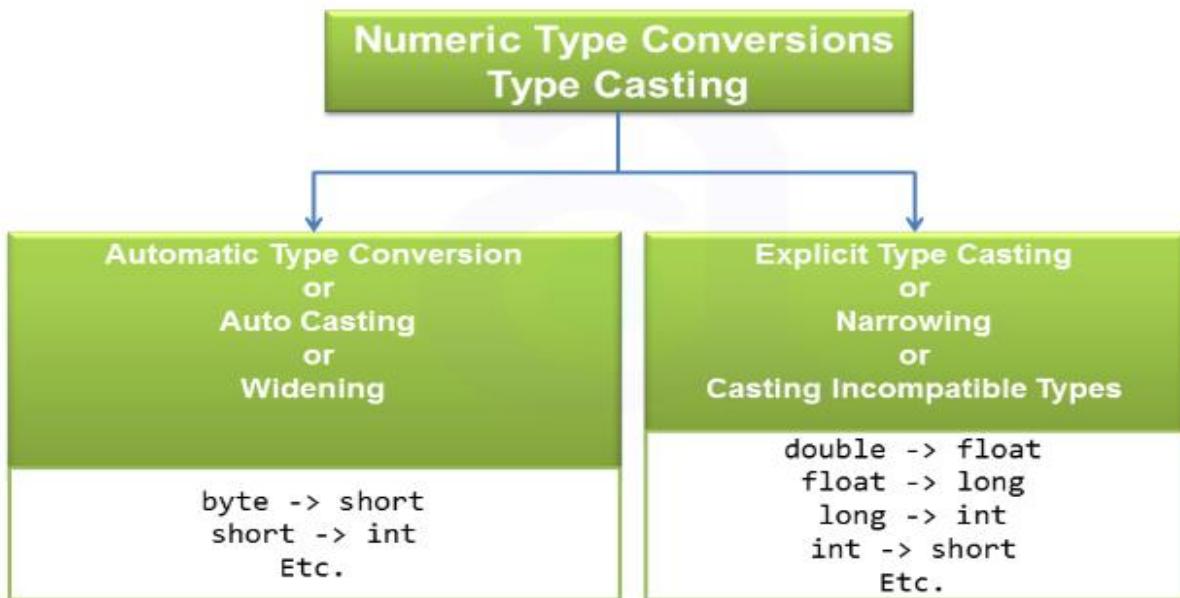


Implicit-typecasting: (widening) or (up casting)

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 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.



ex:

```
class Test
{
    public static void main(String[] args)
    {
        //implicit typecasting (up casting)
        byte b=120;
        int i=b;
        System.out.println(b);

        char ch='a';
        float a=ch;      //[automatic conversion of char to float]
        System.out.println(a);
        //explicit-typecasting (down-casting)
        int a1=130;
        byte b1 =(byte)a1;
        System.out.println(b1);
    }
}
```

Class level type casting:-

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 Parent();
Child c = new Child();
Parent p = new Child(); //valid
Child c = new Parent(); //invalid
```

Example-1:

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

class Child extends Parent
{
    void m1(){System.out.println("child m1 method");}
    void m2(){System.out.println("child m2 method");}
    public static void main(String[] args)
    {
        Parent p1 = new Child();
        p1.m1();
        //p1.m2(); Compilation error we are unable to call m2() method

        Child c1 =(Child)p1;
        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.
- ✓ By using parent reference variables it is possible to access only overriding methods but not direct methods so to access the direct methods perform type casting.

Example -2:

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

        Test t = (Test)o;
        t.m1();
    }
}
```

Example -3:

```
class A
{
    void m1(){}
    void m2(){}
}

class B extends A
{
    void m1(){System.out.println("m1 method");} //overriding method
    void m2(){System.out.println("m2 method");} //overriding method
    void xxx(){System.out.println("xxx method");} //direct method

    public static void main(String[] args)
    {
        A a = new B();
        a.m1();
        a.m2();
        // a.xxx(); compilation error

        B b = (B)a;
        b.xxx();
    }
}
```

Example-4: - overriding vs. hierarchical inheritance

```
class Bike
{
    void run()
    {
        System.out.println("Running");
    }
}

class Splender extends Bike
{
    void run()
```

```

    {
        System.out.println("Splender Running with speed 70KM");
    }
}

class Unicorn extends Bike
{
    void run()
    {
        System.out.println("Unicorn Running with speed 150KM");
    }
}

class MainTest
{
    public static void main(String[] args)
    {
        Bike b1,b2;
        b1 = new Splender();
        b1.run();
        b2= new Unicorn();
        b2.run();
    }
}

```

Example-5 :- Overriding vs multilevel inheritance

```

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

class Ratan extends Person
{
    void eat()
    {
        System.out.println("10-idly");
    }
};

class RatanKid extends Ratan
{
    void eat()
    {
        System.out.println("2-idly");
    }
};

class Test
{
    public static void main(String[] args)
    {
        Person p = new Ratan();
        p.eat();           //completetime: Person runtime:Ratan
        Ratan r = new RatanKid();
        r.eat();           //completetime: Ratan runtime:RatanKid
        Person p1 = new RatanKid();
        p1.eat();          //completetime: Person runtime:RatanKid
    }
};

```

Example-6:- 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);
    }
}

```

Public static void main (String[] args)

Public To provide access permission to the JVM.

Static To provide direct access permission to the JVM (without object creation).

Void Don't return any values to the JVM.

String[] args Used to take command line arguments.

String It is possible to take any type of argument.

[] used to take any number of arguments.

args It is variable of String[] type .

Modifications on main():

✓ Modifiers order is not important it means it is possible to declare **public static** or **static public** .

public static void main(String[] args)

static public void main(String[] args)

✓ The following declarations are valid

string[] args

String []args

String args[]

static public void main(String[] args)

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

- ✓ instead of args it is possible to take any variable name (a,b,c,... etc)
`static public void main(String[] ratan)`
- ✓ from 1.5 version onwards instead of String[] args it is possible to take String... args (var-arg)
`static public void main(String... ratan)`
- ✓ 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 are optional.

Which of the following declarations are valid:

- | | |
|------------------------------------------------------------------------------------|---------------------|
| 1. <code>public static void main(String... a)</code> | ---> valid |
| 2. <code>final strictfp static void mian(String[] Sravya)</code> | ---> invalid |
| 3. <code>static public void mian(String a[])</code> | ---> valid |
| 4. <code>final strictfp synchronized public static void main(String... tcs)</code> | ---> valid |

Strictfp modifier:

- a. It is applicable for classes, methods.
- b. In java the floating point calculations are varied from operating system to operating system & processor to processor hence it will generate platform dependent output.
- c. To overcome above problem to get platform independent results use strictfp modifier.
- d. 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.
- e. If a class is declared as strictfp all methods inside the class will follow IEEE754 standard so we will get platform independent results.

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".

<code>public final int getPriority();</code>	normal method
<code>public static native java.lang.Thread currentThread();</code>	native method

Synchronized modifier:

It is applicable for only methods.

There are two types of methods in java

- a. Synchronized methods
- b. Non-synchronized methods.

Synchronized methods:

Only one thread is able to access the synchronized methods and these methods are thread-safe methods but the performance of the application will be reduced.

If the application requirement is thread-safe then use synchronized methods.

Non-Synchronized methods:

More than one thread is able to access non-synchronized methods but these methods are not thread-safe methods but the performance of the application will be increased.

If the application requirement is performance use non synchronized method.

ex-1:

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

ex-2: main method VS inheritance

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

ex-3 : main method VS overloading

In java it is possible to overload main method but JVM is always calling String[] main method.

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("String[] main method");
        main(100);
    }
}
```

```

        }
    public static void main(int a)
    {
        main('r');
        System.out.println("int main method");
    }
    public static void main(char ch)
    {
        System.out.println("char main method");
    }
}
E:\>java Test
String[] main method
char main method
int main method

```

- ex-4:** It is not possible to override main method because main is a static method.
In java it is not possible to override static methods.

Command Line Arguments:

- ✓ The arguments which are passed from command prompt to application at runtime are called command line arguments.
- ✓ The command line argument separator is space.
- ✓ All command line arguments are stored in `String[]` in the form of String.

```

class Test
{
    public static void main(String[] ratan)
    {
        System.out.println(ratan.length);
        System.out.println(ratan[0]);
        System.out.println(ratan[1]);
        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 45 56
4
100
200
100200
300.0

```

ex-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[] args)
    {
        System.out.println(args[0]);
        System.out.println(args[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
```

Variable -argument method: [var-arg] (1.5 version)

It allows the methods to take any number of arguments.

ex-1 : In below example the m1() method will be executed 4 times.

```
class Test
{
    void m1(int... a)
    {
        System.out.println("m1 Method");
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1();
        t.m1(10);
        t.m1(10,20);
        t.m1(10,20,30);
    }
}
```

ex-2: java methods it is possible to declare normal arguments along with variable arguments.

```
class Test
{
    void m1(char ch,int... a)
    {
        System.out.println(ch);
        for (int a1:a)
        {
            System.out.println(a1);
        }
    }
}
```

```

public static void main(String[] args)
{
    Test t=new Test();
    t.m1('a');
    t.m1('b',10);
    t.m1('c',10,20);
    t.m1('d',10,20,30,40);
}
}

```

ex-3 : 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

ex-4: variable-argument vs. Normal-arguments

If the method contains both variable-argument method & normal argument method then it prints normal argument value.

```

class Test
{
    void m1(int... a)
    {
        System.out.println("variable argument="+a);
    }
    void m1(int a)
    {
        System.out.println("normal argument="+a);
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1(10);
    }
}

```

E:\>java Test
normal argument=10

ex-5 : variable-argument 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);
        t.m1("ratan","Sravya");
    }
}

```

ex 6 : variable -argument method vs. ambiguity

```

class Test
{
    void m1(int... a)      { ----- }
    void m1(String... str) {----- }
    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1(10,20,30);
        t.m1("ratan","Sravya");
        t.m1(); // [compilation error ambiguous]
    }
}

```

*error: reference to m1 is ambiguous,
both method m1(int...) in Test and method m1(String...) in Test match*

Arrays vs. Variable-argument:

Arrays : Here must create the String[] object then pass the data.

Declaration: function(String[] args)
Accessing: function(new String[] {"ratan","anu","durga"})

Var-arg : Here without object directly we can pass the data

Declaration: function(String... args) we can access this declaration in two ways
Accessing: function("ratan","anu","durga")
function(new String[] {"ratan","anu","durga"})

ex:

```

class Test
{
    public static void main(String[] args){
        Test.callMe1(new String[] {"a", "b", "c"});
        Test.callMe2("a", "b", "c");
        Test.callMe2(new String[] {"a", "b", "c"}); // You can also do this
    }
    public static void callMe1(String[] args){
        System.out.println(args.getClass() == String[].class);
        for (String s : args) {

```

```
        System.out.println(s);
    }
}
public static void callMe2(String... args) {
    System.out.println(args.getClass() == String[].class);
    for (String s : args) {
        System.out.println(s);
    }
}
```

***** *Thank you: main method completed* *****

RATAN

Abstraction: Abstract classes

There are two types of methods in java

- a. Normal methods
- b. Abstract methods

Normal methods: (component method/concrete method)

Normal method contains method declaration & method implementation.

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

Abstract methods:

- ✓ The abstract method contains only method declaration but not implementation.
- ✓ Every abstract method must ends with semicolon.
- ✓ To represent method is abstract use abstract modifier.
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: The class which contains only normal methods that class is said to be normal class.

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

Abstract class:-

- ✓ The abstract class may contains abstract methods or may not contains abstract methods but for the abstract classes object creation not allowed.
- ✓ To represent particular class is abstract class use abstract modifier.

Case 1: the class contains at least one abstract method is called abstract class.

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

Case 2: Abstract class does not contain abstract methods.

```
abstract class Test
{
    void m1(){body}
    void m2(){body}
    void m3(){body}
}
```

ex-1:-

- ✓ If the abstract class contains abstract methods write the implementations in child classes.
- ✓ For the abstract classes object creation not possible :
- ✓ The abstract is able to hold Child class object.

Abstract-class reference-variable = new child-class();

```

abstract class Test
{
    abstract void m1();
    abstract void m2();
    abstract void m3();
    void m4() { System.out.println("m4 method"); }
}

class Test1 extends Test
{
    void m1(){System.out.println("m1 method");}
    void m2(){System.out.println("m2 method");}
    void m3(){System.out.println("m3 method");}
}

public static void main(String[] args)
{
    //Test t = new Test(); error: Test is abstract; cannot be instantiated

    Test1 t = new Test1();
    t.m1();
    t.m2();
    t.m3();
    t.m4();

    Test t1 = new Test1(); //abstract class reference variable Child class object
    t1.m1();
    t1.m2();
    t1.m3();
    t1.m4();
}
}

```

ex-2: Inside the abstract class it is possible to declare main method.

```

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

```

ex-3:

- ✓ If the abstract class contains abstract methods write the implementation in child classes.
- ✓ If the child class is unable to provide implementation of all abstract class methods then declare the child class with abstract modifier, and complete the remaining method implementations in next created child classes.
- ✓ It is possible to declare multiple child classes but at final complete the implementation of all abstract methods.
- ✓ If we are completing implementations of all methods then only it is possible to create the object of normal class and possible to access the methods.

```

abstract class Test
{
    abstract void m1();
    abstract void m2();
    abstract void m3();
    void m4(){System.out.println("m4 method");}
}
abstract class Test1 extends Test
{
    void m1(){System.out.println("m1 method");}
}
abstract class Test2 extends Test1
{
    void m2(){System.out.println("m2 method");}
}
class Test3 extends Test2
{
    void m3(){System.out.println("m3 method");}
    public static void main(String[] args)
    {
        Test3 t = new Test3();
        t.m1(); t.m2();
        t.m3(); t.m4();
    }
}

```

ex-4: Inside the abstract class it is possible to declare the constructor.

In below example abstract class constructor is executed but object is not created.

```

abstract class Test
{
    Test()
    {
        System.out.println("abstract calss con");
    }
};
class Test1 extends Test
{
    Test1()
    {
        super();
        System.out.println("normal class con");
    }
}
public static void main(String[] args)
{
    new Test1();
}

```

Example-5:-Abstract methods it is possible to declare any return type& argument

```

class Emp{ };
abstract class Test1
{
    abstract int m1(char ch);
    abstract Emp m3();
}
abstract class Test2 extends Test1
{
    int m1(char ch)
    {
        System.out.println("char value is:-"+ch);
        return 100;
    }
};
class Test3 extends Test2
{
    Emp m3()
    {
        System.out.println("m3 method");
        return new Emp();
    }
    public static void main(String[] args)
    {
        Test3 t=new Test3();
        int a=t.m1('a');
        System.out.println("m1() return value is:-"+a);

        Emp e = t.m3();
        System.out.println("m3() return value is:-"+e);
    }
};

```

Example-6 :-

```

abstract class Test
{
    abstract void m1(int a);
    abstract void m1(char ch);
    abstract void m1(int a,int b);
}
class Test1 extends Test
{
    //overloaded methods
    void m1(int a) {System.out.println("m1 method int arg method");}
    void m1(char ch){System.out.println("m1 method char arg method");}
    void m1(int a,int b){System.out.println("m1 method int,int arg method");}
    public static void main(String[] args)
    {
        Test1 t = new Test1();
        t.m1(10);
        t.m1('a');
        t.m1(10,20);
    }
}

```

Example-7: In child classes it is possible to override abstract methods & possible to declare user defined normal methods also.

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

abstract class Test1 extends Test
{
    void m1(){System.out.println("m1 method");}           //overriding abstract method
    void xxx(){System.out.println("xxx method implementation");} // user specific normal method
}

class Test2 extends Test
{
    void m2(){System.out.println("m2 method");}
    void yyy(){System.out.println("yyy method implementation");}
}
```

Example-8: In child class it is possible to override abstract method & possible to declare our own abstract methods.

```
abstract class Test
{
    abstract void m1();
}

abstract class Test1 extends Test
{
    void m1(){System.out.println("m1 method");}// overriding abstract method
    abstract void xxx();                         // user specific abstract method
}

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

Example-9:-

case 1:- [it is possible to override abstract method to normal method]

```
abstract class Test
{
    abstract void m1();
};

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

case 2:-[it is possible to override normal method to abstract method]

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

abstract class Test1 extends Test
{
    abstract void m1();
};
```

Example 10:-

```
abstract class Bank
{
    abstract int rateOfInterest();
}

class Sbi extends Bank
{
    int rateOfInterest()
    {
        return 8;
    }
}

class Axis extends Bank
{
    int rateOfInterest()
    {
        return 9;
    }
}

class Test
{
    public static void main(String[] args)
    {
        Sbi s1 = new Sbi();
        Axis s2 = new Axis();
        System.out.println(s1.rateOfInterest());
        System.out.println(s2.rateOfInterest());

        Bank b1,b2;
        b1 = new Sbi();
        b2 = new Axis();
        System.out.println(b1.rateOfInterest());
        System.out.println(b2.rateOfInterest());

        Bank b;
        b = new Sbi();
        System.out.println(b.rateOfInterest());
        b = new Axis();
        System.out.println(b.rateOfInterest());
    }
}
```

Example 11:-Inside the abstract class it is possible to declare the variables.

```
abstract class Test
{
    int a,b;
    Test(int a,int b)
    {
        this.a=a;
        This.b=b;
    }
}
class Test1 extends Test
{
    Test1(int a,int b)
    {
        super(a,b);
    }
    void m1()
    {
        System.out.println(a+"----"+b);
    }
    public static void main(String[] args)
    {
        new Test1(10,20).m1();
    }
}
```

Example 12 :-Inside the abstract class it is possible to declare instance blocks & static blocks.

```
abstract class Test
{
    {
        System.out.println("abstract class instance block");
    }
    Static {System.out.println("abstract class static block");}
}
class Test1 extends Test
{
    {System.out.println("normal class instance block");}
    static {System.out.println("normal class static block");}
    public static void main(String[] args)
    {
        new Test1();
    }
}
```

Abstraction definition :-

- ✓ The process highlighting the set of services and hiding the internal implementation is called abstraction.
- ✓ We are achieving abstraction concept by using Abstract classes & Interfaces.
- ✓ 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 classrooms.

Encapsulation:-

- ✓ The process of binding the data(variables) and code(methods) as a single unit is called encapsulation.
- ✓ Group mechanism is called encapsulation.
- ✓ The process of hiding the implementation details to user is called encapsulation. And we are achieving this concept by declaring variables as a private modifier because it is possible to access private members within the class only.

Data hiding :

- ✓ The main objective of data hiding is security and it is possible to hide the data by using private modifier.
- ✓ If a variable declared as a private it is possible to access those variables only inside the class is called data hiding.

Fully or Tightly encapsulated class:-

The class contains only private properties that class is said to be tightly encapsulated class.

```
class Emp
{
    private int eid;
    private String ename;
    private double esal;
}
```

Example:- java bean class or VO(value object) class or BO(business object) class

If the variables are declared as a private it is possible to access those variables only within the class but it is possible to set(update) the data by using setter methods and it is possible to get(read) the data by using getter methods.

The data of the private field can be accessed only by using public setter & getter method

In this way we are hiding implementation to other classes. The setter and getter methods are userdefined methods.

Syntax:-
 setXXX() where xxx=property name
 getXXX() where xxx=property name

The setter method return type is always void & getter method return type is always property return type.

Java Bean rules :-

- ✓ Bean class is reusable software component anyone can set the data & get the data.
- ✓ The bean class contains private properties & public setter & getter methods.
- ✓ The bean class must be public class.
- ✓ In bean class declare 0-arg cons.
- ✓ The bean class must implement serializable interface to support serialization concept.

Core Java	:	bean class
Struts	:	form bean
Spring	:	bean component
Hibernate	:	pojo class
JSF	:	managed bean
EJB	:	Enterprise bean

File-1 EmpBean.java

```
Public class EmpBean
{
    private int sid;
    private int sname;
    public void setSid(int sid)
    {
        this.sid=sid;
    }
    public void setSname(String sname)
    {
        this.sname=sname;
    }
    public int getSid()
    {
        return sid;
    }
    public String getSname()
    {
        return sname;
    }
};
```

Accessing encapsulated use following code: Test.java

```
class Test
{
    public static void main(String[] args)
    {
        EmpBean e=new EmpBean();
        e.setSid(100);
        e.setSname("ratan");

        System.out.println(e.getSid());
        System.out.println(e.getSname());
    }
}
```

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 supported by java?
- 5) How to prevent inheritance concept?
- 6) What is the default super class in java?
- 7) One class able to extends how many classes at a time?
- 8) What is the purpose of extends keyword?
- 9) What do you mean by cyclic inheritance java supporting or not?
- 10) Which approach is recommended to create object either parent class object or child class object?
- 11) Every class contains parent class in java except one class what is that class?
- 12) What is the purpose of instanceof keyword in java?
- 13) What is the root class for all java classes?
- 14) How to call super class constructors?
- 15) Is it possible to use both super and this keyword inside the method?
- 16) Is it possible to use both super and this keyword inside the constructor?
- 17) Inside the constructor if we are not providing this() and super() keyword the compiler generated which type of super keyword?
- 18) What is the execution process of constructors if two classes are there in inheritance relationship?
- 19) What is the execution process of instance blocks if two classes are there in inheritance relationship?
- 20) What is the execution process of static blocks if two classes are there in inheritance relationship?
- 21) What is the purpose of instanceof operator in java & what is the return-type?
- 22) If we are using instanceof both reference-variable & class-name must have some relationship otherwise compiler generated error message is what?
- 23) If the child class and parent class contains same variable name that situation how to call parent class variable in child class?
- 24) What do you mean by aggregation and what is the difference between aggregation and inheritance?
- 25) What do you mean by aggregation and composition and Association?
- 26) Aggregation is also known as?
- 27) What do you mean by polymorphism?
- 28) How many types of polymorphism in java?
- 29) What do you mean by method overloading and method overriding?
- 30) How many types of overloading in java?
- 31) Java supports operator overloading or not?
- 32) Is it possible to overload the constructors are not?
- 33) What do you mean by constructor overloading?
- 34) What are the implicit overloaded operators in java explain it?
- 35) What do you mean by overriding? What are the rules?
- 36) What do you mean by overridden method & overriding method?
- 37) To achieve overriding how many java classes are required?

- 38) Is it possible to override variable in java?
- 39) When we will get compilation error like “overridden method is final”?
- 40) What is the purpose of final modifier java?
- 41) Is it possible to override static methods yes- → how no- → why?
- 42) Parent class reference variable is able to hold child class object or not?
- 43) What do you mean by dynamic method dispatch?
- 44) The applicable modifiers only on local variables?
- 45) What do you mean by type casting & how many types?
- 46) 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?
- 47) When we will get compilation error like “con not inherit from final parent”?
- 48) What do you mean by co-varient return types?
- 49) What do u mean by method hiding and how to prevent method hiding concept?
- 50) What do you mean by abstraction?
- 51) What is the difference between normal method and abstract method?
- 52) What is the difference between normal class and abstract class?
- 53) Is it possible to create a object for abstract class?
- 54) Is it possible to override non-abstract method as a abstract method?
- 55) Is it possible to declare main method inside the abstract class or not?
- 56) What is the purpose of abstract modifier in java?
- 57) How to prevent object creation in java?
- 58) What is the definition of abstract class?
- 59) In java is it abstract class reference variable is able to hold child class object or not?
- 60) What do you mean by encapsulation & what is the examples encapsulation?
- 61) What do you mean by tightly encapsulated class?
- 62) What do you mean accessor method and mutator method ?
- 63) How many ways area there to set some values to class properties (variables)?
- 64) What do you mean by javaBean class?
- 65) In java program execution starts from which method & who is calling that method?
- 66) Can we inherit main method in child class?
- 67) The applicable modifiers on main method?
- 68) What is the return type of main method?
- 69) What are the mandatory modifiers for main method and optional modifiers of main method?
- 70) What do you by command line arguments ?
- 71) Is it possible to pass command line arguments with space symbol no → why yes → how ?
- 72) What is the purpose of strictfp modifier?
- 73) What is the purpose of native modifier?
- 74) What do you mean by native method and it also known as?
- 75) Is it possible to overload the main method or not yes=how no=why?
- 76) Is it possible to override the main method or not yes=how no=why?
- 77) What is the purpose of variable argument method & what is the syntax?
- 78) If the application contains both normal argument & var-arg then which one executed first?
- 79) Is it possible to overload the variable argument methods are not?
- 80) What is the difference between method overloading & variable argument method.

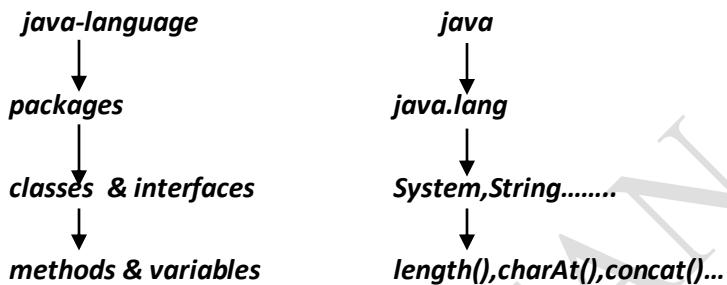
***** Thank you *****

Packages

In java the predefined support maintained in the form of packages and these packages contains,

- Classes
- Interfaces
- Enum
- Annotations
- Exceptions
- Errors

The above all contains predefined methods, variables, constants & constructors.



- java is open source technology and it is possible to check source code of the java.
- The source code location **C:\Program Files\Java\jdk1.7.0_75\src(zip file)** extract the zip file.
- Java contains 14 predefined packages but the default package in java if **java.lang**. package.

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

Note : package is nothing but **physical folder structure**.

Types of packages:

There are two types of packages in java

- 1) Predefined packages.
- 2) User defined packages.

Predefined packages:

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

ex: **java.lang, java.io ,java.util.....etc**

User defined packages:

The packages which are defined by user, and these packages contains user defined classes and interfaces.

- ✓ Declare the package by using **package** keyword.

syntax : **package package-name;**

ex : package declaration

```
package com.tcs;
class A
{
}
```

Folder Structure :
 com
 |--->tcs
 |--->A.class

- ✓ Inside the source file it is possible to declare only one package statement and that statement must be first statement of the source file, otherwise compiler generate error message.

case-1: valid

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

case-2: Invalid

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

case-3: Invalid

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

case-4: Invalid

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

Package name coding conventions :- (not mandatory but we have to follow)

- 1) The package name must reflect with organization domain name(reverse of domain name).

Domain name: www.tcs.com

Package name: package com.tcs;

- 2) Package name must reflect with project name.

Project name : bank

package : package com.tcs.bank;

- 3) The project name must reflect with project module name.

Domain name: www.tcs.com

Project name: bank

Module name: deposit

package name: package com.tcs.bank.deposit;

Advantages of packages:

company name : tcs
project name : bank

module-1 deposit

```
com
|-->tcs
  |-->bank
    |-->deposit
      |--->.class files
```

module-2 withdraw

```
com
|-->tcs
  |-->bank
    |-->withdraw
      |--->.class files
```

Module-3 loans

```
com
|-->tcs
  |-->bank
    |-->loans
      |--->.class files
```

module-4 account

```
com
|-->tcs
  |-->bank
    |-->account
      |--->.class files
```

- 1) It improves parallel development of the project.
- 2) Project maintenance will become easy.
- 3) It improves readability.
- 4) It improves reusing of properties.

Note : In real time the project is divided into number of modules that each and every module is nothing but package statement.

Example-1:**Step-1: write the application with package statement.**

```

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

```

Step-2: compilation process : use below command.

Javac -d . Test.java

javac	--->	java compiler
-d	--->	creates folder structure
.	--->	Current working directory
Test.java	--->	source file name

Step-3: folder Structure.

```

com
|-->tcs
  |-->java
    |-->corejava
      |-->Test.class
      |-->A.class
      |-->It.class

```

Step-4:-execution process.

Execute .class file by using fully qualified name(class name with complete package structure).

E:\>java com.tcs.java.corejava

Example-2 :

```

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

```

G:\>javac -d F:\ratan Test.java

The folder structure is stored in local disk F in ratan folder.

Example-3:**error-1 :**

Whenever we are using other packages classes then import that package by using import statement.

There are two ways to import the classes,

- **Importing all classes.**
Import java.lang.*;
- **Importing application required classes**
Import java.lang.System;
Import java.lang.String;

error-2:

Whenever we are using other package classes, that classes must be public classes otherwise compiler generate error message.

Default modifier:

- It is applicable for variables, methods, classes, constructors.
- The default modifier in java is default.
- It is possible to access default members only within the package but not outside package.
- Default access is also known as package level access.

Public modifier:

- ✓ Public modifier is applicable for variables, methods, classes, constructors.
- ✓ All packages are able to access public members.

error-3:

Whenever we are using other package class member that members also must be public.

Note: if a class declare class as public the corresponding members are not public, if we want access public class members that members also must be public.

File-1: Durga.java

```
package com.sravya.states.info;
public class Durga
{
    public void ts() {System.out.println("jai telengana");}
    public void ap(){System.out.println("jai andhra");}
    public void others(){System.out.println("jai others");}
}
```

File-2: Tcs.java

```
package com.tcs.states.requiredinfo;
import com.sravya.states.info.Durga;
class Tcs
{
    public static void main(String[] args)
    {
        Durga d = new Durga();
        d.ts(); d.ap(); d.others();
    }
}
```

E:\>javac -d . Durga.java

E:\>javac -d . Tcs.java

compilation of Sravya

compilation of Tcs

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

execution of Tcs

Example-6:

It is possible to use other package classes in two ways.

- 1) Application with import statement. (Access the class directly without full-name)
- 2) Application without import statement. (Access the class by using full-name)

Test.java

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

MainTest.java : Application with import statement : (Access the class directly without full-name)

```
package com.dss.client;
import com.dss.Test;
class MainTest
{
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
    }
}
```

MainTest.java : Application without import statement. (Access the class by using full-name)

```
package com.dss.client;
class MainTest
{
    public static void main(String[] args)
    {
        com.dss.Test t = new com.dss.Test();
        t.m1();
    }
}
```

Example - 7 :**A.java :**

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

A.java: Different module

```
package app2;
public class A
{
    public void m1()
    {
        System.out.println("app2 m1() method");
    }
}
```

*In above two A.java files**First write the first file then compile the file then it will generate folder structure.**Don't create another file, do the modification on existing file compile it then folder are generated.***MainTest.java:**

```
package com.dss.client;
class MainTest
{
    public static void main(String[] args)
    {
        app1.A a = new app1.A();
        a.m1();

        app2.A a1 = new app2.A();
        a1.m1();
    }
}
```

```
G:\>javac -d . A.java
G:\>javac -d . A.java
G:\>javac -d . MainTest.java
G:\>java com.dss.client.MainTest
app1 m1() method
app2 m1() method
```

<code>app1 -->A.class</code>	module-1
<code>app2 -->A.class</code>	module-2

<code>com -->dss</code>	module-3
---------------------------------	-----------------

|-->client |-->MainTest.class

Note : If two different modules contains same class name(A) if third module required those two classes then use the fully qualified name to access those classes . It is not required to import.

Example -8:

Test.java:

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

A-Java

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

B. *jaya*:

```
package com.dss.advjava;
public class B
{
    public void m1()
    {
        System.out.println("B class m1");
    }
}
```

MainTest.java:

```
package com.dss.client;
import com.dss.Test;
import com.dss.corejava.A;
import com.dss.advjava.B;
class MainTest
{
    public static void main(String[] args)
    {
        new A().m1();
        new B().m1();
        new Test().m1();
    }
}
```

Compilation & execution:

```
G:\>javac -d . Test.java  
G:\>javac -d . A.java  
G:\>javac -d . B.java  
G:\>javac -d . MainTest.java  
G:\>java com.dss.client.MainTest  
A class m1()  
B class m1  
Test class m1()
```

Folder Structure:

```
com
|-->dss
    |-->Test.class
    |-->corejava
        |         |-->A.class
    |-->advjava
        |         |-->B.class
    |-->client
        |-->MainTest.class
```

Note : when we import the main package with * then it is possible to access only main package classes but not sub package classes, if we want to access the sub package classes must import sub packages also.

ex-4 : Private modifier

- ✓ private modifier applicable for methods and variables, constructors.
- ✓ It is possible to access private members only within the class, it is not possible to access even in child classes.

```
class Parent
{
    private int a=10;
}
class Child extends Parent
{
    void m1()
    {
        System.out.println(a);
    }
    public static void main(String[] args)
    {
        Child c = new Child();
        c.m1();
    }
}
error: a has private access in Parent
```

ex-5 : Protected modifier

- ✓ Protected modifier is applicable for variables, methods, constructors.
- ✓ We are able access protected members within the package and it is possible to access outside packages also but only direct child classes & it is not possible to call even in indirect child classes.
- ✓ But in outside package we can access protected members only by using child reference. If we try to use parent reference we will get compile time error.

pkg-1

```
class A
{
    protected int a=10;
}
class B
{
    System.out.println(a); //possible
}
class C extends A
{
    System.out.println(a); //possible
}
```

pkg-2

```
class D extends A      //Direct sub class
{
    System.out.println(a); //possible
}
class E
{
    System.out.println(a); //not possible
}
```

```
class F extends D      //indirect sub cals
{
    System.out.println(a); //not possible
}
```

Modifiers Summary-1:

	Private	default	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

Modifiers Summary-2:

	classes	methods	variables	constructors
Public	yes	yes	yes	yes
Private	no	yes	yes	yes
Protected	no	yes	yes	yes
Default	yes	yes	yes	yes

Example-12: Static import (1.5 version)

There are two types of imports in java

1. Normal import
2. Static import

By using static import it is possible to access only static members directly into application without using class-name.

Durga.java

```
package com.dss;
public class Durga
{
    public static int a=100;
    public static void m1(){System.out.println("m1 method");}
}
```

Tcs.java : Application with normal import : access the static members by using class-name

```
package com.tcs;
import com.dss.Durga;
class Tcs
{
    public static void main(String[] args)
    {
        System.out.println(Durga.a);
        Durga.m1();
    }
}
```

Tcs.java : Application with static import : Access the static members directly without class-name

```
package com.tcs;
import static com.dss.Durga.*;
class Tcs
{
    public static void main(String[] args)
    {
        System.out.println(a);
```

```

        m1();
    }
}

```

ex-14 :

There are two types of imports in java

1. *normal import*
2. *static import*

Normal Import : *By using normal import it is possible to access both instance & static members but access the static members by using class-name.*

Import java.lang.System;

Static import :- *By using static import It is possible to access only static members of particular class directly into application without using class-name.*

Import static java.lang.System.*;

Durga.java:

```

package com.dss;
public class Durga
{
    public static int fee=1000;
    public void course()
    {
        System.out.println("core java");
    }
}

```

Tcs.java:

```

package com.tcs;
import static com.dss.Test.*; // static import
import com.dss.Test; // normal import
class Tcs
{
    public static void main(String[] args)
    {
        System.out.println(fee);

        Test t = new Test();
        t.course();
    }
}

```

Question : *Already normal import is available to access both static and non-static members of a class then what is the use of the static import.*

Answer :

By using normal import we can access both static & instance members but access the static members by using class-name

By using static import we can access the only static members directly into application without using class-name

ex-10 : System.out.println

- System:** *System is a class present in java.lang package.*
- Out:** *out is a static variable of system class of type PrintStream.*
`public final static PrintStream out = null;`
- println:** *it is a method of printStream class used to print data in output console.*
The PrintStream class present in java.io package.

The predefined implementation is like this

```
public void println(String x) {
    synchronized (this) {
        print(x);
        newLine();
    }
}
```

Example-13:

With normal import

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

With static import

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

Example -9: In java it is not possible to use predefined package names as a user defined packages. If we are trying to use predefined package names as a user defined packages at runtime JVM will generate securityException.

```
package java.lang;
class Test
{
    public static void main(String[] args)
    {
        System.out.println("Ratan World!");
    }
}
D:\DP>javac -d . Test.java
D:\DP>java java.lang.Test
Exception in thread "main" java.lang.SecurityException: Prohibited package name: java.lang
```

Example-11 : The source file is allows to declare only one public class but if you want more public classes in single module use fallowing structure.

In below four files save separately when we compiled all the file single folder structure is generated in that folder all the four classes are stored.

A.java:

```
package com.dss;
public class A
{}
```

B.java:

```
package com.dss;
public class B
{}
```

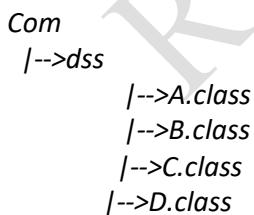
C.java:

```
package com.dss;
public class C
{}
```

D.java:

```
package com.dss;
public class D
{}
```

[in single module all public classes are stored]



Applicable modifiers on constructors:-

- 1) Public
- 2) Private
- 3) Protected
- 4) Default

example : Default constructors we can access only within the package

A.java:

```
package com.dss;
public class A
{   A()
    { System.out.println("A class cons");
    }
}
```

B.java:

```
package com.tcs;
import com.dss.A;
public class B
{   public static void main(String[] args)
    { new A();
    }
}
```

G:\>javac -d . A.java

G:\>javac -d . B.java
B.java:5: error: A() is not public in A; cannot be accessed from outside package

Public constructor : all packages are able to access the public constructors.

Private constructor:

- ✓ It is possible to access the private constructor only within the class. Hence in another classes it is not possible to create the object.
- ✓ If you want to prevent the object creation outside of the class then declare the private constructor inside the class.

Test.java:

```
public class A
{   private A()
    { System.out.println("A class cons");
    }
}
class B
{   public static void main(String[] args)
    { A a = new A();
    }
}
```

G:\>javac Test.java

A.java:9: error: A() has private access in A

Example :Durga.java

```
class Parent
{   private Parent(){}
}
```

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

G:\>javac Durga.java

Durga.java:6: error: Parent() has private access in Parent

In above example in child class default constructor is generated in that default constructor super() keyword is generated but Parent class constructor is private hence it will generate compilation error.

Packages

1. What do you mean by package and what it contains?
2. How many pre-defined packages in java?
3. What is the default package in java?
4. Is it possible to declare package statement any statement of the source file?
5. What is the difference between user's defined package and predefined package?
6. What are coding conventions must follow while declaring user defined package names?
7. Is it possible to declare multiple packages in single source file?
8. What do you mean by import?
9. What is the location of predefined packages in our system?
10. How many types of imports present in java explain it?
11. How to import individual class and all classes of packages and which one is recommended?
12. What do you mean by static import?
13. What is the difference between normal and static import?
14. I am importing two packages, both packages contains one class with same name at that situation how to create object of two package classes?
15. If we are importing root package at that situation is it possible to use sub package classes in our applications?
16. What is difference between main package and sub package?
17. If source file contains package statement then by using which command we are compiling that source file?
18. What do you mean by fully qualified name of class?
19. What is the default modifier in java?
20. What is the public access and default access?
21. The public class members(variables,methods,constructors) are by default public or not?
22. What is private access and protected access?
23. What is most restricted modifier in java?
24. What is most accessible modifier in java?
25. Is it possible to declare pre-defined package names as a user defined package names or not?
26. What are the applicable modifiers for constructors?
27. Is it possible to override private methods or not yes=how no=why?
28. When we will get compilation error "attempting to assign weaker access privileges" how to rectify?

*******Thank you*******

Interfaces

- ✓ The interfaces are extension of abstract classes
 - The abstract class may contains abstract methods or may not contains abstract methods but for the abstract classes object creation not possible.
 - The interface allows to declare only abstract methods(up to 1.7 version)
- ✓ Interfaces are used to declare the functionality declarations but not definition.
- ✓ Interfaces are used to achieve full abstraction.
- ✓ Interface is similar to class it is a collection of abstract methods.
- ✓ For the interfaces compiler will generates .class files after compilation.

Declare the interface by using interface keyword.

Syntax: Interface interface-name
 { abstract methods here;
 }

Code Before compilation

```
interface It1
{
    void m1();
    void m2();
    void m3();
}
```

Code After compilation

```
abstract interface It1
{
    public abstract void m1();
    public abstract void m2();
    public abstract void m3();
}
```

The interfaces are by default abstract so interfaces object creation not allowed.

Interface methods are by default **public & abstract**.

Interface variables are by default **public static final**.

Why we use Interface?

1. It is used to achieve fully abstraction.
2. It can be used to achieve loose coupling.

How interface different from class:

- ✓ It is not possible to create the object of interface.
- ✓ Inside the interface constructor declaration not allowed.
- ✓ All methods in interfaces are by default public and abstract.
- ✓ One class can extends only one class but one interface is able to extend more than one interface.
- ✓ Interface contains only static fields but not instance fields.

Ex-1:

- ✓ Interface contains abstract methods; write the implementation in implementation classes.
- ✓ The class which implements particular interface is called implementation class.
- ✓ The interface implementation methods must be public because interface methods are by default public & abstract. Otherwise compiler generate error message "**attempting to assign weaker access privileges**".
- ✓ The interface is able to hold the implementation class object.

Interface-name reference-variable = new class-name ();

```
interface it1
{
    void m1();
    void m2();
}
Class Test implements it1
{
    Public void m1()
    {
        System.out.println("m1-method implementation ");
    }
    Public void m2()
    {
        System.out.println("m2-method implementation");
    }
    Public static void main(String[] args)
    {
        Test t=new Test();
        t.m1();
        t.m2();
    }
}

It1 i = new Test();
i.m1();
i.m2();
```

- ✓ The above example is overriding concept only, the parent contains declarations so in child class override that methods to write the implementations.
- ✓ Overriding all rules applicable when we provide the implementation of interface methods.
- ✓ While overriding possible to maintain same level permissions or increasing order but not decreasing order.

Ex-2: one interface can have multiple implementation classes.

```
interface Message
{
    void wish();
}

class Ratan implements Message
{
    public void wish()
    {
        System.out.println("good morning");
    }
}

class Anu implements Message
{
    public void wish()
    {
        System.out.println("good evening");
    }
}

class Test
{
    public static void main(String[] args)
    {
        Message m1 = new Ratan();
        m1.wish();
        Message m2 = new Anu();
        m2.wish();
    }
}
```

Ex-3:

- ✓ Interface contains abstract methods write the implementation in implementation class.
- ✓ If the implementation class is unable to provide the implementation of all abstract methods then declare implementation class with abstract modifier & complete the remaining abstract method implementation in next created child classes.
- ✓ In java it is possible to take any number of child classes but at final complete the implementation of all abstract methods.

```
interface It1
{
    void m1();
    void m2();
    void m3();
}

abstract class Test implements It1
{
    public void m1(){System.out.println("m1 method");}
}

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

class Test2 extends Test1
{
    public void m3(){System.out.println("m3 method");}
}

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

```

    {
        Test2 t = new Test2();
        t.m1();          t.m2();          t.m3();
    }
}

```

ex :

```

interface Message
{
    void morn();
    void even();
}

abstract class Morning implements Message
{
    public void morn()
    {
        System.out.println("good morning");
    }
}

class Evening extends Morning
{
    public void even()
    {
        System.out.println("good evening");
    }

    public static void main(String[] args)
    {
        Message m = new Evening();
        m.morn();
        m.even();
    }
}

```

ex: interface methods it is possible to any return type & any number of arguments.

```

interface it1
{
    String m1(int a,int b);
    int m2(char ch,String str)
    void m3(int a);
}

```

Complete the impl in two child classes

Ex: possible to overload the abstract methods of interface.

```

interface it1
{
    void m1(int a,int b);
    void m1(int a)
    void m1(char ch)
}

```

Complete the implementation in two child classes

Interfaces vs. Inheritance:

Class **extends** class
 Interface **extends** interface
 Class **implements** interface

Case 1 : one interface can extends another interface.

```
interface It1
{
    void m1();
}
interface It2 extends It1
{
    void m2();
}
interface It3 extends It2
{
    void m3();
}
class Test implements It3
{
    Must override 3 methods
}
```

Case 2: One class can extends only one class at a time

But one interface is able to extend more than one interface at a time.

```
interface It1
{
    void m1();
}
interface It2
{
    void m2();
}
interface It3 extends It1,It2
{
    void m3();
}
class Test implements It3
{
    Must override 3 methods
}
```

Case 3: one interface can extends multiple interfaces, in this case If the interfaces contains command method then override that method only once.

```
interface It1
{
    void m1();
    void m2();
}
interface It2
{
    void m2();
    void m3();
}
interface It3 extends It1,It2
```

```

    {
        void m4();
    }
    class Test implements It3
    {
        Must override 4 methods.
    }

```

Case-4 : one class is able to implements more than one interface.

```

interface It1
{
    void m1();
}

interface It2
{
    void m2();
}

interface It3
{
    void m3();
}

class Test implements It1
{
    Must override 1 methods
}

class Test1 implements It1,It2
{
    Must override 2 methods
}

class Test2 extends It1,It2,It3
{
    Must override 3 methods
}

```

Case 2: If more than one interface is having same method then just provide the implementation only once.

```

interface It1
{
    void m1();
    void m2();
}

interface It2
{
    void m2(int a);
    void m3();
}

class Test implements It1,It2
{
    public void m1()
    {
        System.out.println("m1 method");
    }

    public void m2()
    {
        System.out.println("m2 method");
    }

    public void m3()
    {
        System.out.println("m3 method");
    }

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

```

```

        t.m2();
        t.m3();
    }
}

```

Possibility of extends & implements keywords :

<i>class extends class</i>	
<i>interface extends interface</i>	
<i>class implements interface</i>	
<i>class A extends B</i>	--->valid
<i>class A extends B,C</i>	--->Invalid
<i>class A extends A</i>	--->Invalid
<i>class A implements It</i>	--->valid
<i>class A implements It1,It2</i>	--->valid
<i>interface It1 extends It2</i>	--->valid
<i>interface It1 extends It2,It3</i>	--->valid
<i>interface It1 extends It1</i>	--->invalid
<i>interface It1 extends A</i>	--->invalid
<i>interface It1 implements It2</i>	--->invalid
<i>class A extends B implements It1,It2</i>	--->valid
<i>class A implements It1,It2 extends B</i>	--->Invalid

When we declare extends & implements in single line, in this case extends must be first statement.

Interface variables: Interface variables are by default **public static final**.

- ✓ Interface variables are final so modifications are not allowed.
- ✓ Inside the interface it is not possible to declare the instance variables.
- ✓ Inside the interface it is not possible to declare constructors, instance blocks, static blocks.

```

interface It1
{
    int a=10;
}
interface It2
{
    int a=100;
}
class Test implements It1,It2
{
    public void disp()
    {
        //System.out.println(a); error: reference to a is ambiguous
        System.out.println(It1.a);
        System.out.println(It2.a);
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.disp();
    }
}

```

```

        }
    }
}
```

Nested interfaces:**Case 1: Declaring interface inside the interface is called nested interface.**

```

interface it1
{
    void m1();
    interface it2
    {
        void m2();
    }
}
class Test implements it1,it1.it2
{
    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();
    }
}
```

Case 2: It is possible to declare interfaces inside abstract class also.

```

abstract class A
{
    abstract void m1();
    interface it1
    {
        void m2();
    }
}
class Test extends A 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();
    }
}
```

Case 3: Declaring interface inside the normal class is called nested interface.

```

class A
{
    interface it1
    {
        void add();
    }
}
class Test implements A.it1
{
    public void add()
    {
        System.out.println("add method");
    }
}
```

```

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

```

Interface Java 8 features:

ex : Inside the interface it is possible to declare the default & static methods from java8 version.

```

interface It1
{
    void m1();
    default void m2()
    {
        System.out.println("default method");
    }
    static void m3()
    {
        System.out.println("static method");
    }
}
class Test implements It1
{
    public void m1()
    {
        System.out.println("abstract method");
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
        t.m2();
        It1.m3();
    }
}

```

ex : inside the interface it is possible to declare the main method from java 8.

```

interface It1
{
    public static void main(String[] args)
    {
        System.out.println("interface main");
    }
}

```

ex : If interfaces contains same default method then must override that method in implementation class.

```

interface It1
{
    default void m1(){      System.out.println("It1 m1() method"); }
}
interface It2
{
    default void m1(){      System.out.println("It2 m1() method"); }
}
class Test implements It1,It2
{
    public void m1()
    {
        System.out.println("m1 method common implementation");
    }
}

```

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

Functional interface: java8 version

The functional Interface contains only one abstract method, it may contain default, static methods .

case 1: valid

```
interface It1
{
    void m1();
}
```

case 2: valid

```
interface It1
{
    void m1();
    default void m2(){}
}
```

case 3: valid

```
interface It1
{
    void m1();
    default void m2(){}
    static void m3(){}
}
```

case 4: Invalid

```
interface It1
{
    void m1();
    void m2();
    default void m2(){}
}
```

Project implementation :**Level-1 :** pkg-1 : Interface: Contains only declaration : upto java7

```
interface Bank
{
    void deposit();
    void withdraw();
    void loan();
    void account();
}
```

Level-2 : pkg-2 : Abstract class: Continas some declaration & some implementation

```
abstract class Helper implements Bank
{
    void deposit() {implementation here}
    void withdraw() {implementation here}
}
```

Level-3 : pkg-3: Normal class: Contains only implementation

```
class Test extends Helper
{
    void account() {implementation here}
    void loan() {implementation here}
}
```

Level-4 : pkg-4 : Client Code

```
class TestClient
{
    public static void main(String[] rgs)
    {
        Test t = new Test();
        t.account();
        t.loan();
        t.deposit();
        t.withdraw();
    }
}
```

Developer level thinking:

```
class MyClass implements Inteface
{
    Must override all methods of interface
}

class MyClass extends AbstractClass
{
    Override some method impl : the abstract methos present in abstract class
}

class MyClass extends NormalClasses
{
    Override required methods
```

```
}
```

Adaptor class:

Limitation of interface:

If the interface contains 10 methods in implementation class must override 10 methods if you required or not.

To override above problem to override required methods use adaptor class concept.

The adaptor class is a normal java class contains empty implementation of interface methods.

Note :- If our class implementing interface must override all methods but whenever our class extending adaptor class it is possible to override required methods.

Example:-

interface It

```
{
    void m1();
    void m2();
    .....
    void m10();
}
class X implements It //adaptor class
{
    public void m1(){}
    public void m2(){}
    .....
    public void m10{}
};
```

//user defined class implementing interface must override all methods

```
class Test implements It
{
    Must provide 10 methods implementations.
};
```

//user defined class extending Adaptor class(X)

```
class Test extends X
{
    Override required methods because already X class contains empty implementations.
}
```

Interfaces vs abstract class :- project level usage.

Message.java:-

```
package com.sravya.declarations;
public interface Message
{
    void morn();
    void even();
    void gn();
}
```

Helper.java-

```
package com.sravya.helper;
import com.sravya.declarations.Message;
public abstract class Helper implements Message
```

```
{
    public void gn(){      System.out.println("good night from helper class");      }
}
```

TestClient1.java:-

```
package com.sravya.client;
import com.sravya.declarations.Message;
class TestClient1 implements Message
{
    public void morn(){System.out.println("good morning");}
    public void even(){System.out.println("good evening");}
    public void gn(){System.out.println("good 9t");}
    public static void main(String[] args)
    {
        TestClient1 t = new TestClient1();
        t.morn();           t.even();           t.gn();
    }
}
```

TestClient2.java:-

```
package com.sravya.client;
import com.sravya.helper.Helper;
class TestClient2 extends Helper
{
    public void morn(){System.out.println("good morning");}
    public void even(){System.out.println("good evening");}
    public static void main(String[] args)
    {
        TestClient2 t = new TestClient2();
        t.morn();           t.even();           t.gn();
    }
}
```

D:\>javac -d . Message.java

D:\>javac -d . Helper.java

D:\>javac -d . TestClient1.java

D:\>javac -d . TestClient2.java

D:\>**java com.sravya.client.TestClient1**

good morning

good evening

good 9t

D:\>**java com.sravya.client.TestClient2**

good morning

good evening

good night from helper class

Marker interface:

- ✓ The interface does not contain any methods but whenever our class implements that interface our class must acquire some capabilities to perform some operations, such type of interfaces are called marker interfaces.

java.io.Serializable	----> provides Serialization capabilities
java.lang.Cloneable	----> provides cloning capabilities
java.util.RandomAccess	----> data accessing capabilities

- ✓ Marker interface is empty interface but class is acquiring capabilities these capabilities are provided by JVM.

Object cloning:-

- ✓ The process of creating exactly duplicate object is called cloning.
- ✓ The main objective of cloning is to maintain backup copy.

- ✓ If we want to create the duplicate object then our class must implements cloneable interface.
- ✓ And it is a marker interface present in java.lang package.
- ✓ The cloneable is a marker interface it does not contains any methods but whenever our class is implementing cloneabe interface our class is acquiring some capabilities to perform some operations those capabilities are provided by JVM.
- ✓ To create the duplicate object use clone() method of object class.

```
class Test implements Cloneable
{
    int a=10,b=20;
    public static void main(String[] args) throws CloneNotSupportedException
    {
        Test t1 = new Test();
        Test t2 = (Test)t1.clone();
        System.out.println(t1.a+"---"+t1.b);
        t1.a=444;      t1.b=555;
        System.out.println(t1.a+"---"+t1.b);
        System.out.println(t2.a+"---"+t2.b);           //reading values from duplicate object
    }
}
```

Example :

```
class Emp implements Cloneable
{
    int eid;
    String ename;
    Emp(int eid,String ename)
    {
        this.eid=eid;
        this.ename=ename;
    }
    void disp()
    {
        System.out.println("Emp id="+eid+" Emp name="+ename);
    }
    public static void main(String[] args) throws CloneNotSupportedException
    {
        Emp e1 = new Emp(111,"ratan");
        Emp e2 = (Emp)e1.clone();
        e1.disp();
        e2.disp();
    }
}
```

Serialization:- The process of converting java object to network supported form or file supported form is called serialization. or

The process of saving an object to a file is called serialization. (or)

To do the serialization we required fallowing 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*

The perform serialization our class must implements Serializable interfaces.

Serializable is a marker interface it does not contains any methods but whenever our class is implementing Serializable interface our class is acquiring some capabilities to perform some operations those capabilities are provided by JVM.

Example :- Emp.java

```
import java.io.*;
class Emp implements Serializable
{
    int eid;
    String ename;
    Emp(int eid, String ename)
    {
        this.eid = eid;
        this.ename = ename;
    }
}
```

SerializationTest.java:-

```
Class SerializationTest
{
    public static void main(String[] args) throws Exception
    {
        Emp e = new Emp(111, "ratan");
        //serialization [write the object to file]
        FileOutputStream fos = new FileOutputStream("xxxx.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(e);
        System.out.println("serialization completed");
    }
}
```

DeserializationTest.java:-

```
Class DeserializationTest
{
    public static void main(String[] args) throws Exception
    {
        FileInputStream fis = new FileInputStream("xxxx.txt"); //deserialization reading obj
        ObjectInputStream ois = new ObjectInputStream(fis);
        Emp e1 = (Emp) ois.readObject(); //returns Object
        System.out.println(e1.eid + "----" + e1.ename);
        System.out.println("de serialization completed");
    }
}
```

Example :- It is possible to serialize the multiple objects but in which order we serialize same order we have to deserialize otherwise JVM will generate ClassCastException.

```
import java.io.*;
class Employee implements Serializable
{
    int eid;
    String ename;
    Employee(int eid, String ename)
    {
        this.eid = eid;
        this.ename = ename;
    }
}
```

```

class Dog implements Serializable
{
}
class Cat implements Serializable
{
}
class Test
{
    void serialization() throws Exception
    {
        Employee e = new Employee(111,"ratan");
        Dog d = new Dog();
        Cat c = new Cat();
        //serialization [write the object to file]
        FileOutputStream fos = new FileOutputStream("xxxx.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(e);           oos.writeObject(d);           oos.writeObject(c);
        System.out.println("serialization of multiple objects completed");
    }
    void deserialization() throws Exception
    {
        //deserialization [read object from text file]
        FileInputStream fis = new FileInputStream("xxxx.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Employee e1 = (Employee)ois.readObject(); //returns Object
        Dog d = (Dog)ois.readObject();
        Cat c = (Cat)ois.readObject();
        System.out.println(e1.eid+"----"+e1.ename);
        System.out.println(d);
        System.out.println(c);
        System.out.println("deserialization completed");
    }
    public static void main(String[] args) throws Exception
    {
        Test t = new Test();
        t.serialization();
        t.deserialization();
    }
}

```

Case 1: it is not possible to serialize static data members in java.

```

Class Test implements Serializable
{
    static int eid=111;
}

```

Case 2:

```

Class Address
{
    int dno;
}
Class Emp implements Serializable
{
    int eno;
}

```

```

String ename;
Address addr;
}
✓ It is not possible to serialize Emp class because the address class is not serializable.
✓ To perform emp class serialization the corresponding all members are must be serializable.

```

case 3:

Rule: In case of array or collection, all the objects of array or collection must be serializable. If any object is not serializable, serialization will be failed.

Transient Modifiers :-

- ✓ it is the modifier applicable for only variables.
- ✓ If a variable declared as a transient those variables are not participated in serialization instead of original values default values will be printed.
- ✓ To prevent the serialization use transient modifier.

Emp.java

```

import java.io.*;
class Emp implements Serializable
{
    transient int eid;
    transient String ename;
    Emp(int eid,String ename)
    {
        this.eid=eid;
        this.ename=ename;
    }
}

```

Output : 0----null

Difference between abstract classes & interfaces:-**Abstract class**

- 1) The purpose of abstract class is to specify default functionality of an object and let its sub classes explicitly implement that functionality. It stands it is providing abstraction layer that must be extended and implemented by the corresponding sub classes.

- 2) An abstract class is a class that declared with **abstract** modifier.

Ex: **abstract class A**
{ **abstract void m1();** }

- 3) The abstract allows declaring both abstract & concrete methods.
- 4) Abstract class methods must declare with abstract modifier.
- 5) If the abstract class contains abstract methods then write the implementations in child classes.
- 6) In child class the implementation methods need not be public it means while overriding it is possible to declare any valid modifier.

- 7) The abstract class is able to provide implementations of interface methods.
- 8) One java class is able to extends only one abstract class at a time.
- 9) Inside abstract class it is possible to declare main method & constructors.
- 10) It is not possible to instantiate abstract class.
- 11) For the abstract classes compiler will generate .class files.
- 12) The variables of abstract class need not be **public static final**.
- 3. The interface allows declaring only abstract methods.
- 4. Interface methods are by default **public abstract**.
- 5. The interface contains abstract methods write the implementations in implementation classes.
- 6. In implementation class the implementation methods must be public.
- 7. The interface is unable to provide implementation of abstract class methods.
- 8. One java class is able to implements multiple interfaces at a time.
- 9. Inside interface it is not possible to declare methods and constructors.
- 10. It is not possible to instantiate interfaces.
- 11. For the interfaces compiler will generate .class files.
- 12. The variables declared in interface by default **public static final**.

Interface

- 1. It is providing complete abstraction layer and it contains only declarations of the project then write the implementations in implementation classes.
 - 2. Declare the interface by using **interface** keyword.
- Ex:- **interface It1**
 { **void m1();**}

Interfaces

- a. What do you mean by interface how to declare interfaces in java?
- b. Interfaces allow normal methods or abstract methods or both?
- c. For the interfaces compiler generates .class files or not?
- d. What is the abstract method?
- e. Interface methods are by default which type?
- f. What is the purpose of **implements** keyword?
- g. Is it possible to declare variables in interface?
- h. Can abstract class have constructor? Can interface have constructor?
- i. What do you by implementation class?
- j. Is it possible to create object of interfaces?
- k. What do you mean by abstract class?

- l. When we will get compilation error like “attempting to assign weaker access privileges”?
- m. What is the difference between abstract class and interface?
- n. Which of the following declarations are valid & invalid?
- o. The interface reference variable is able to hold implementation class objects or not?
- p. One interface is able to extend more than one interface or not?
- q. One class able to implements more than one interface or not?
- r. What is the real-time usage of interfaces?
- s. What is the limitation of interfaces how to overcome that limitation?
- t. What do you mean by adaptor class?
- u. What is the difference between adaptor class interfaces?
- v. Is it possible to create user defined adaptor classes?
- w. Tell me some of the adaptor classes?
- x. What do you mean by marker interface and it is also known as?
- y. Define marker interfaces?
- z. What are the advantages of marker interfaces?
- aa. What is the purpose of the cloning process?
- bb. What do you mean by serialization?
- cc. Is it possible to serialize static members?
- dd. How to prevent serialization?
- ee. Is it possible to create user defined marker interfaces or not?
- ff. Is it possible to declare nested interfaces or not?
- gg. What is the purpose of the transient modifier?
- hh. Java 8 version interfaces allow what type of methods?
- ii. What do you mean by functional interface?

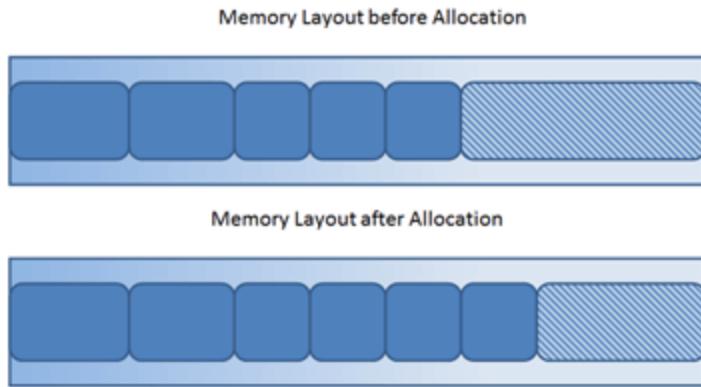
***** ***Thank you*** *****

Garbage Collector

- ✓ In C language we are allocating memory by using malloc() function and we are destroying memory by using free(), here the developer is responsible for both operations .
 - ✓ In CPP language we are allocating memory by using constructors and we are destroying memory by using destructors, here the developer is responsible for both operations.
 - ✓ In java programmer is responsible to allocate the memory by creating of object and memory will be destroyed by Garbage collector it is a part of the JVM.
- ❖ As long as an object is being referenced, the JVM considers the object alive.
- ❖ UN-referenced (Object without reference) is called garbage.
- ❖ To destroy the objects the garbage collector will follow mark-and-sweep algorithm.
- 1) Create the objects.
 - 2) Use the objects in application.
 - 3) Once the usage is completed give the object to garbage collector.
 - 4) The garbage collector is responsible to destroy the object.

In project level after using object make eligible that object to garbage collector then garbage collector will destroy the objects.

Object creation is faster because global synchronization with the operating system is not needed for every single object. An allocation simply claims some portion of a memory array and moves the offset pointer forward. The next allocation starts at this offset and claims the next portion of the array.



When an object is no longer used, the garbage collector reclaims the underlying memory and reuses it for future object allocation. This means there is no explicit deletion and no memory is given back to the operating system.

Advantages:

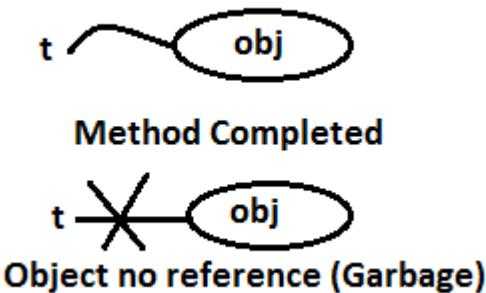
1. It makes the java memory efficient because garbage collector removes useless objects.
2. It is automatically called by JVM no need to write extra code.
3. Garbage collector eliminating memory leaks and other memory-related problems

There are four ways to make eligible your objects to garbage collector:

Approach 1 Whenever we are creating objects inside the methods one method is completed the objects are eligible for garbage collector.

```
class Test
{
    void m1()
    {
        Test t = new Test();
    }
}
```

In above case once the method is completed reference will destroy. Then object is no reference is called garbage.



Approach-2 Assigning null constants to our objects then objects are eligible for GC.

```
class Test
{
    public static void main(String[] args)
    {
        Test t1=new Test();
        System.out.println(t1);
    }
}
```

```

        t1=null;
        System.out.println(t1);

    }

}

```

Approach-3 : Whenever we reassign the reference variable the objects are automatically eligible to GC.

```

class Test
{
    public static void main(String[] args)
    {
        StringBuffer s1 = new StringBuffer("hello");
        StringBuffer s2 = new StringBuffer("ratan");
        s1 = s2;
        System.out.println(s1);
        System.out.println(s2);
    }
}

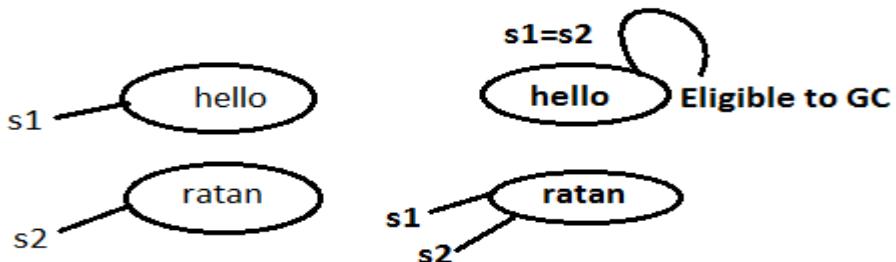
```

In above case s1 object is pointing to s2
So the "hello" object is eligible for garbage collector.

G:\>java Test

ratan

ratan



Approach-4 : Nameless object once the line is completed that object is eligible to gc() .

```

class Test
{
    public void finalize()
    {
        System.out.println("object destroyed.... ");
    }
    public static void main(String[] args)
    {
        new Test();
        System.gc();
    }
}

```

Example :-

- To call the garbage collector explicitly use gc() method it is a static method system class.
- Just before destroying object garbage collector will call finalize() method.
- Finalize() method present in object class & it is called by garbage collector just before destroying object.

- If we are overriding finalize() then our class finalize() executed , if we are not overriding finalize() method then object class finalize() method executed having empty implementation.

```
class Test
{
    public void finalize()
    {System.out.println("object destroyed");
    }
    public static void main(String[] args)
    {
        Test t1 = new Test();
        Test t2 = new Test();
        System.out.println(t1.toString());
        System.out.println(t2.toString());
        ;;;;;;;//using object
        t1=null;
        t2=null;
        System.gc();
    }
};
```

Observation :

- ✓ When JVM calls finalize method ,if any exceptions raised in finalize() method those exceptions are ignored & objects will be destroyed.
- ✓ When programmer calls finalize method , if any exception raised in finalize() method that exception raised program terminated abnormally.

```
public void finalize()
{System.out.println("object destroyed");
System.out.println(10/0);
}
```

Example :

- ✓ Just before destroying object that class finalize method will be executed.
 - Emp obj destroyed ---- Emp class finalize method executed
 - Student obj destroyed ---- Student class finalize method executed
 - String obj destroyed ----String finalize method executed (having empty implementation)

```
class Emp
{
    public void finalize()
    {   System.out.println("Emp obj destroyed");
    }
}
class Test
{
    public void finalize()
    {   System.out.println("Test obj destroyed");
}
```

```

        }
    public static void main(String[] args)
    {
        Test t = new Test();
        t=null;

        Emp e = new Emp();
        e=null;
        System.gc();
    }
}

```

java.lang.Runtime :-

- ✓ This class is used to interact with java runtime environment.
- ✓ The java Runtime class is provide the facility,to execute a process,to call GC,to check free memory & total memory.....etc

Example :-System class gc() & Runtime class gc()

It is possible to call the garbage collector in two ways

1. System class gc() method
 2. Runtime class gc() method
- ✓ Runtime class gc() method is a instance method it is directly interact with garbage collector.
 - ✓ System class gc() method is static method it is internally calling runtime class gc() method to call the garbage collector.

```

class Test
{
    public void finalize()
    {
        System.out.println("object destroyed");
    }
    public static void main(String[] args)
    {
        Test t1 = new Test();
        System.out.println(t1.toString());
        t1=null;
        Runtime r = Runtime.getRuntime();
        r.gc();
    }
};

```

Example :-

```

class Test
{
    public static void main(String[] args) throws Exception
    {
        Runtime r = Runtime.getRuntime();
        System.out.println("Total memory..... "+r.totalMemory());
        System.out.println("Free memory..... "+r.freeMemory());
        for(int i=0;i<100000;i++)
        {
            new Test();
        }
        System.out.println("Free memory after 10000 objects..... "+r.freeMemory());
        r.gc();
        System.out.println("Free memory after GC called..... "+r.freeMemory());
    }
}

```

Example :-opening notepad& shutdown the system & restart the system by using Runtime class.

```

class Test
{
    public static void main(String[] args) throws Exception
    {
        Runtime.getRuntime().exec("notepad");
        Runtime.getRuntime().exec("shutdown -s -t 0");
        Runtime.getRuntime().exec("shutdown -r -t 0");
    }
}

```

Example : in below example one object(e2) is eligible to GC.

```

class Emp{ }
class Test
{
    public void finalize()
    {
        System.out.println("object destroyed");
    }
    Emp m1()
    {
        Emp e1 = new Emp();
        Emp e2 = new Emp();
        System.out.println(e1);
        return e1;
    }
    public static void main(String[] args)
    {
        Emp e = new Test().m1();
        System.out.println(e);
        Runtime.getRuntime().gc();
    }
}

```

Example : Two objects are eligible to garbage collector.

```

class Emp{ }
class Test
{
    static Emp m1()
    {
        Emp e1 = new Emp();
        Emp e2 = new Emp();
    }
}

```

```
        return e1;
    }
    public static void main(String[] args)
    {
        Test.m1();
    }
}
```

Example :-one object(e1) is eligible to garbage collector.

```
class Emp{ }
class Test
{
    static Emp e;
static void m1()
{Emp e1 = new Emp();
e = new Emp();
}
}
```

Garbage Collector

1. What is the functionality of Garbage collector?
2. How many ways are there to make eligible our objects to Garbage collector?
3. How to call Garbage collector explicitly?
4. What is the purpose of gc() method?
5. What is the purpose of finalize() method?
6. If the exception raised in finalize block what happened **error or output?**
7. What is the purpose of RunTime class?
8. How to create object of RunTime class?
9. What is singleton class?
10. What is the algorithm followed by GC?
11. What is the difference between final , finally , finalize()?
12. When GarbageCollector calls finalize()?
13. Finalize method present in which class?
14. Which part of the memory involved in garbage collector Heap or Stack?
15. Who creates stack memory and who destroy that memory?
16. What do you mean by demon thread? Is Garbage collector is DemonThread?
17. How many times Garbage collector does call finalize() method for object?
18. What are the different ways to call Garbage collector ?
19. Is it possible to call finalize() method explicitly by the programmer?
20. When we destroy the Student class object which class finalize method executed.

***** **Thank you** *****

String manipulations

*Java.lang.String
Java.lang.StringBuffer
Java.lang.StringBuilder
Java.util.StringTokenizer*

Java.lang.String:

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

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

        String str1=new String("ratan");
        System.out.println(str1);

        char[] ch={'r','a','t','a','n'};
        String str3=new String(ch);
        System.out.println(str3);

        char[] ch1={'a','r','a','t','a','n','a'};
        String str4=new String(ch1,2,2);
        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(b1,3,3);
System.out.println(str6);
}
}

```

```

E:\>java Test
ratan
ratan
ratan
at
ABCDEF
DEF

```

Case 1: String vs. StringBuffer

- ✓ String & StringBuffer both classes are final classes present in **java.lang** package.
- ✓ The default value of String, StringBuffer is : **null**

Case 2: String vs StringBuffer

It is possible to create String object in two ways.

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

```

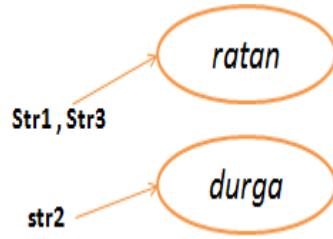
class Test
{
    public static void main(String[] args)
    {
        //Two approaches to create a String object
        String str1 = "ratan";
        String str2 = new String("anu");
        System.out.println(str1+str2);

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

```

Creating a string without using new operator:

- ✓ When we create String object without using new operator the objects are created in SCP (String constant pool) memory.
- ✓ **String str1="ratan";**
String str2="durga";
String str3="ratan";



- ✓ In String constant pool memory just before object creation it is always checking previous objects.
 - If the previous objects are not available then JVM will create new object.
 - If the previous object is available with the same content then it won't create new object that reference variable pointing to existing object.
- ✓ SCP does not allow duplicate objects.

Creating a string object by using new operator

- ✓ When we create String object by using new operator the object created in heap area.

```
✓ String str1=new String("ratan");
String str2 = new String("durga");
String str3 = new String("ratan");
```

== Operator:

By using == operator we can compare primitive data & reference data returns Boolean value.

- ✓ If the primitive data is same returns true otherwise false.

```
int a=10,b=10;
System.out.println(a==b);
```

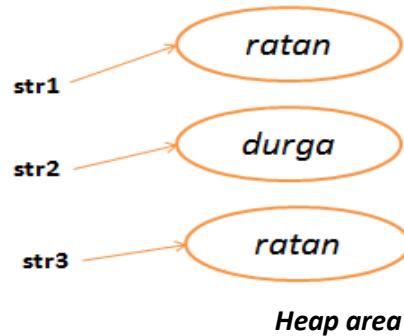
- ✓ If two reference variables are pointing to same object then it returns true otherwise false.

```
class Test{
    public static void main(String[] args){
        Object obj1 = new Object();
        Object obj2 = new Object();
        System.out.println(obj1==obj2); //false
        obj1=obj2;
        System.out.println(obj1==obj2); //true
    }
}
```

ex:

```
class Test
{
    public static void main(String[] args)
    {
        Test t1 = new Test();
        Test t2 = new Test();
        System.out.println(t1==t2);

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



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

- ✓ Heap memory allows duplicate objects.

```

String s1 = new String("anu");
String s2 = new String("anu");
System.out.println(s1==s2);

StringBuffer sb1 = new StringBuffer("sravya");
StringBuffer sb2 = new StringBuffer("sravya");
System.out.println(sb1==sb2);
}

E:\>java Test
false
true
false
false

```

Case 3: 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 equals() hence it uses object class equals() perform reference comparison
}

```

ex : class Test
 { Test(String str)
 {
 }

```

public static void main(String[] args)
{
    //Object class equals() method executed (reference comparison)
    Test t1 = new Test("ratan");
    Test t2 = new Test("ratan");
    System.out.println(t1.equals(t2));

    //String class equals() method executed (content comparison)
    String str1="anu";
    String str2="anu";
    System.out.println(str1.equals(str2));

    //String class equals() method executed (content comparison)
    String s1 = new String("Sravya");
    String s2 = new String("Sravya");
    System.out.println(s1.equals(s2));

    //StringBuffer class not overriding equals() method so object class equals executed
    StringBuffer sb1 = new StringBuffer("anu");
    StringBuffer sb2 = new StringBuffer("anu");
    System.out.println(sb1.equals(sb2));
}

== operator vs equals( )method :

```

✓ **== Operator :**

Used to check reference comparison & primitive data comparison, returns Boolean value.

- If the primitive data same returns true otherwise false.
- If two reference variables are pointing to same object returns true otherwise false.

✓ **equals() method :**

Used to compare only object data but not primitive data, returns Boolean value.

- Result of equals() method depends on overridden implementation.
- equals() method present in object class it perform reference comparison.
- String class overriding equals() method perform content comparison.
- StringBuffer class not overriding equals() method it uses parent class(Object) equals method perform reference comparison.

```

class Test
{
    public static void main(String[] args)
    {
        String str1 = "hello";
        String str2 = "hello";
        String str3= new String("hello");

        //identity checking
        System.out.println(str1==str2);
        System.out.println(str1==str3);
        System.out.println(str2==str3);
    }
}

```

```

        //equality checking
        System.out.println(str1.equals(str2));
        System.out.println(str1.equals(str3));
        System.out.println(str2.equals(str3));
    }
}

```

Case 4: immutable vs. mutable

- ✓ *String is immutable* 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.

```

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

        StringBuffer sb = new StringBuffer("anu");
        sb.append("soft");
        System.out.println(sb);      //anusoft
    }
}

```

concat () : This method is used combining two String objects and it is returning new String object.

public java.lang.String concat(java.lang.String);

```

class Test
{
    public static void main(String[] args)
    {
        String str="ratan";

```

```

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

```

Observation:

```

class Test
{
    public static void main(String[] args)
    {
        String str="ratan";
        str = str.concat("soft");
        System.out.println(str);           // ratansoft
    }
}

```

- ✓ In above example after concatenation we are assigning the data to same existing reference variable.
- ✓ In above example we are not performing modifications on existing objects, here the modification is performed in newly created object.

**Case 5 : `toString()` method :**

- ✓ In java when you print reference variable internally it calls `toString()` method.
- ✓ `toString()` method present in `Object` and it return String representation of object `classname@hashcode` in the form of String.

ex-1: `class Test`

```

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

```

G:\>java Test
Test@2a139a55
Test@2a139a55

Internal implementation of `toString()`

```

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

```

ex-2: `class Emp`

```

{
    int eid;
    String ename;
}

```

```

    Emp(int eid,String ename)
    {
        this.eid=eid;
        this.ename=ename;
    }
    //overriding toString() method to write our implementation
    public String toString()
    {
        return "Emp eid="+eid+" Emp name="+ename;
    }
    public static void main(String[] args)
    {
        Emp e = new Emp(111,"ratan");
        System.out.println(e);
        System.out.println(e.toString());
    }
}

```

If we are not overriding toString() then parent class toString() executed

```

D:\>java Emp
Emp@15db9742
Emp@15db9742

```

If we are overriding toString() then our class toString() executed

```

D:\>java Emp
Emp eid=111 Emp name=ratan
Emp eid=111 Emp name=ratan

```

ex-3:

- ✓ *toString()* method present in object class it returns a string representation of object(**classname@hashcode**).
- ✓ *String, StringBuffer, all wrapper classes* *toString()* method overriding to return content of the object.

*Note: possible to call the *toString()* method only on Object type but not primitive type.*

```

class Object
{
    public java.lang.String toString()
    {
        return getClass().getName() + '@' + Integer.toHexString(hashCode());
    }
}
class String extends Object
{
    //overriding toString() method
    public java.lang.String toString()
    {
        return "content of String";
    }
}
class StringBuffer extends Object
{
    //overriding toString() method
    public java.lang.String toString()
    {
        return "content of StringBuffer object";
    }
}

class Test

```

```

{
    public static void main(String[] args)
    {
        Test t = new Test();
        System.out.println(t);
        System.out.println(t.toString());

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

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

D:\>java Test
Test@530daa
Test@530daa
ratan
ratan
anu
anu

```

case 6 : conversion process

```

class Test
{
    public static void main(String... ratan)
    {
        //conversion of String to StringBuffer
        String str1="ratan";
        StringBuffer sb = new StringBuffer(str1);
        System.out.println(sb);

        //conversion of StringBuffer to String
        StringBuffer sb1 = new StringBuffer("anu");
        String s = sb1.toString();
        System.out.println(s);
    }
}

```

Observation 1: conversion of String to StringBuffer then apply reverse() method.

```

class Test
{
    public static void main(String... ratan)
    {
        String str1="ratan";
        StringBuffer sb = new StringBuffer(str1);
        System.out.println(sb.reverse());
    }
}

```

Observation 2 :

```

class Test
{
    public static void main(String... ratan)
    {
        String[] str={"119d","madhapur","hyderabad"};
        StringBuffer sb = new StringBuffer();
        for (String s:str)
        {
            sb.append(s+",");
        }

        String s = sb.toString();
        System.out.println(s);
    }
}

```

CompareTo () vs equals ():

- ✓ equals() method is used to compare two String object it returns Boolean value as a return value.
- ✓ If two Strings are equals it return true otherwise false.

public boolean equals(java.lang.Object);

- ✓ compareTo() method used to compare two String objects return int value as a return value.
- ✓ compareTo() we are comparing two strings character by character, such type of checking is called lexicographically checking or dictionary checking. It returns integer value.
 - if the two strings are equal then it return zero.
 - If the first string first character Unicode value is bigger than second string first character Unicode value then it return +ve value.
 - If the first string first character Unicode value is smaller than second string first character Unicode value then it return -ve value.

public int compareTo(java.lang.String);

```

class Test
{
    public static void main(String... ratan)
    {
        String str1="ratan";
        String str2="anu";
        String str3="ratan";

        System.out.println(str1.equals(str2));
        System.out.println(str1.equals(str3));
    }
}

```

```

        System.out.println(str2.equals(str3));
        System.out.println("ratan".equals("RATAN"));
        System.out.println("ratan".equalsIgnoreCase("RATAN"));

        System.out.println(str1.compareTo(str2));
        System.out.println(str1.compareTo(str3));
        System.out.println(str2.compareTo(str1));
        System.out.println("ratan".compareTo("RATAN"));
        System.out.println("ratan".compareToIgnoreCase("RATAN"));
    }
}

```

String class Methods:

Difference between length() method and length variable:

length variable used to find length of the Array.

length() is method used to find length of the String.

charAt(int) : possible to extract the character from particular index position.

public char charAt(int);

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

public java.lang.String[] split(java.lang.String);

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

public java.lang.String trim();

public java.lang.String toLowerCase();

public java.lang.String toUpperCase();

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

endsWith() is used to find out if the string is ending with particular character/string or not.

startsWith() used to find out the particular String starting with particular character/string or not.

public boolean startsWith(java.lang.String);

public boolean endsWith(java.lang.String);

substring() method used to find substring in main String.

public java.lang.String substring(int); **int = starting index**

`public java.lang.String substring(int, int);` **int=starting index to int =ending index**
`substring()` method it includes starting index & it excludes ending index.

ex: possible to concat the string data in two ways

```
class Test{
    public static void main(String[] args){
        String str1 = "Ratan";
        String str2 = "Soft";

        //Approach 1 : Using concat
        String str3 = str1.concat(str2);
        System.out.println(str3);

        //Approach 2 : Using "+" operator
        String str4 = str1 + str2;
        System.out.println(str4);
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        //length var & length() metod
        int [] a={10,20,30};
        System.out.println(a.length);
        String sc="ratan";
        System.out.println(sc.length());

        //cahrAt() method
        String ss="ratan";
        System.out.println(ss.charAt(1));
        //System.out.println(ss.charAt(10));StringIndexOutOfBoundsException
        char ch=ss.charAt(2);
        System.out.println(ch);

        //method chaning
        String s1 = " ratan ";
        System.out.println(s1.length());
        System.out.println(s1.trim().length());
        System.out.println(s1.trim().substring(2).length());
        System.out.println(s1.trim().substring(2,4).length());

        //spliting data
        String s2 = "hi sir how are you";
        String[] s = s2.split(" ");
        for (String ss : s)
```

```

    {
        System.out.println(ss);
    }

//start with & ends with
String str="hi sir how r u";
System.out.println(str.endsWith("u"));
System.out.println(str.endsWith("how"));
System.out.println(str.startsWith("d"));
System.out.println(str.startsWith("r"));

//checking the data & Upper case & lower case data
String str1 = "hi sir how r u";
System.out.println(str1.contains("how"));
System.out.println(str1.contains("ratan"));
System.out.println("ratan".toUpperCase());
System.out.println("RATAN".toLowerCase());
//replace the data
String str="ratan how r u";
System.out.println(str.replace('a','A'));
System.out.println(str.replace("how","who"));
}

}

```

StringBuffer class methods:

reverse()& delete() & deleteCharAt(),Append() insert() replace()

```

public class Test{
    public static void main(String[] args) {
        StringBuffer sb1 = new StringBuffer("ratanit");
        System.out.println(sb1.delete(2, 4));
        System.out.println(sb1.deleteCharAt(3));

        StringBuffer sb2 = new StringBuffer("durgasoft");
        System.out.println(sb2.reverse());

        StringBuffer sb3 = new StringBuffer("ratan");
        sb3.append("soft");
        System.out.println(sb3);

        StringBuffer sb4 = new StringBuffer("ratanit");
        sb4.insert(0, "hi");
        System.out.println(sb4);

        StringBuffer sb5 = new StringBuffer("hi durga soft");
        sb5.replace(3, 9, "ratan");
        System.out.println(sb5);
    }
}

```

Java.lang.StringBuilder: 1.5 version

StringBuilder is same as StringBuffer except one difference.

- *StringBuffer methods are synchronized only one thread is allowed to access, these methods are thread safe methods but performance will be decreased.*
- *StringBulder methods are non-synchronized it is not a thread safe but performance will be increased.*

StringTokenizer:

- ✓ *StringTokenizer present in java.util package & it is a legacy class.*
- ✓ *It used to split the string into number of tokens. The default splitting character is space.*
- ✓ *To check tokens are available or not use hasMoreElements() method & to print the token use nextElement() method.*

These are the methods inherited from Enumeration interface

```
public boolean hasMoreElements();
public java.lang.Object nextElement();
```

These methods from StringTokenizer

```
public boolean hasMoreTokens();
public java.lang.String nextToken();
public int countTokens();
```

```
import java.util.StringTokenizer;
public class Test8{
    public static void main(String[] args) {
        StringTokenizer st = new StringTokenizer("hi sir class completed");
        while(st.hasMoreElements())
        {
            String s = (String) st.nextElement();
            System.out.println(s);
        }
    }
}
```

```
 StringTokenizer st1 = new StringTokenizer("hi si.r how. are y.out", ".");  
 while(st1.hasMoreTokens())  
 {      String ss = st1.nextToken();  
     System.out.println(ss);  
 }  
 }
```

String manipulation interview Questions

- 1) How many ways to create a String object & StringBuffer object?
- 2) What is the difference between
 - a. String str="ratan";
 - b. String str = new String("ratan");
- 3) equals() method present in which class?
- 4) What is purpose of String class equals() method.
- 5) What is the difference between equals() and == operator?
- 6) What is the difference between by immutability & immutability?
- 7) Can you please tell me some of the immutable classes and mutable classes?
- 8) String & StringBuffer & StringBuilder & StringTokenizer presented package names?
- 9) What is the purpose of String class equals() & StringBuffer class equals()?
- 10) What is the purpose of StringTokenizer and this class functionality replaced method name?
- 11) How to reverse String class content?
- 12) What is the purpose of trim?
- 13) Is it possible to create StringBuffer object by passing String object as a argument?
- 14) What is the difference between concat() method & append()?
- 15) What is the purpose of concat() and toString()?
- 16) What is the difference between StringBuffer and StringBuilder?
- 17) What is the difference between String and StringBuffer?
- 18) What is the difference between compareTo() vs equals()?
- 19) What is the purpose of contains() method?
- 20) What is the difference between length vs length()?

- 21) What is the default capacity of StringBuffer?
- 22) What do you mean by factory method?
- 23) Concat() method is a factory method or not?
- 24) What is the difference between heap memory and String constant pool memory?
- 25) String is a final class or not?
- 26) StringBuilder and StringTokenizer introduced in which versions?
- 27) What do you mean by legacy class & can you please give me one example of legacy class?
- 28) How to apply StringBuffer class methods on String class Object content?
- 29) When we use String & StringBuffer & String
- 30) What do you mean by cloning and use of cloning?
- 31) Who many types of cloning in java?
- 32) What do you mean by cloneable interface present in which package and what is the purpose?
- 33) What do you mean by marker interface and Cloneable is a marker interface or not?
- 34) How to create duplicate object in java(by using which method)?

*****Thank you*****

Wrapper classes

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

Advantages of wrapper classes:

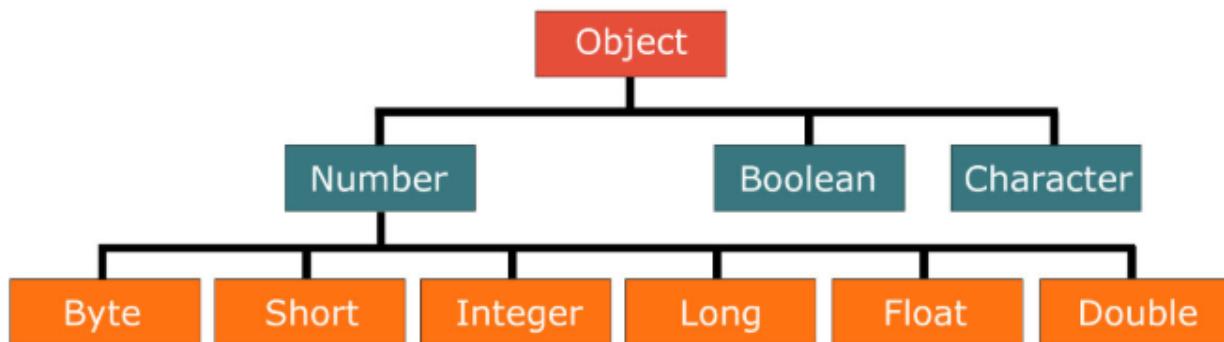
When we wrap the primitive into object format we have following advantages,

1. The Collection framework support only objects data but not primitive data in this case must convert your primitive data into object data.
2. To provide the type safety to the collection we are using generics but generics support only object data but not primitive data.

<code>ArrayList<int></code>	---- invalid
<code>ArrayList<Integer></code>	---- Valid

3. *Wrapper class objects allow null values while primitive data type doesn't allow it.*
4. *The Object data support more operations like*
 - cloning process
 - *toString()* method applied
 - achieve polymorphism
 - synchronization process....et

Wrapper Class Hierarchy



There are two approaches to create wrapper object,

- a. *constructor approach*
- b. *valueOf() method*

Wrapper object creation by using 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');
```

Wrapper object creation by using valueOf():

```
Integer x1 = Integer.valueOf(10);
Integer x2 = Integer.valueOf("10");
```

```
Float f1 = Float.valueOf(10.5);
Float f2 = Float.valueOf(10.5f);
Float f3 = Float.valueOf("10.5");
```

```
Character ch = Character.valueOf('a');
```

Primitive	Wrapper Class	Constructor Arguments
boolean	Boolean	boolean or String
byte	Byte	byte or String
char	Character	char
double	Double	double or String
float	Float	float, double, or String
int	Integer	int or String
long	Long	long or String
short	Short	short or String

ex 1: Conversion of primitive or String ----> wrapper object

```
class Test
{
    public static void main(String[] args)
    {
        //constructor approach
        Integer i1 = new Integer(10);
        Integer i2 = new Integer("10");
        System.out.println(i1+i2);

        //valueOf() method approach
        Integer x1 = Integer.valueOf(100);
        Integer x2 = Integer.valueOf("100");
        System.out.println(x1+x2);
    }
}
```

Observation-1:

`Integer i1 = new Integer("10");` **valid** : string format "10" converted into integer object format.
`Integer i2 = new Integer("ten");` **Invalid** : String format "ten" is unable to converted into Integer.

- ✓ In above line, to the integer constructor we are passing "10" value in the form of String it is automatically converted into Integer format.
- ✓ In above line, to the integer constructor we are passing "ten" in the form of String but this String is unable to convert into integer format JVM generate **java.lang.NumberFormatException**.
`Integer a = new Integer("ten");` **java.lang.NumberFormatException**

Observation-2:

`Integer i = new Integer("ten");` **java.lang.NumberFormatException**
`Integer x = Integer.valueOf("one");` **java.lang.NumberFormatException**
In both object creations we will get same error

```
Integer x1 = Integer.valueOf(null);           NumberFormatException
When we pass the null values we will get exception.
```

Observation-3: Boolean Object creation

To the Boolean constructor if we are passing other than true or false, always it return false.

Boolean b1 = new Boolean(true);	true
Boolean b2 = new Boolean("TRUE");	true
Boolean b3 = new Boolean("FaLsE");	false
Boolean b4 = new Boolean("ratan");	false
Boolean b5 = new Boolean(null);	false
Boolean b6 = new Boolean(TRUE);	Invalid

Ex-2: Conversion process : wrapper object ----> primitive/String

Wrapper object to primitive : xxxValue()
 Wrapper object to String : using toString().

```
public class Test {
    public static void main(String[] args) {
        Integer i1 = Integer.valueOf(100);
        Integer i2 = Integer.valueOf(100);

        //Wrapper object to primitive data: xxxValue()  xxx=all data types
        int x = i1.intValue();
        double d = i1.doubleValue();
        System.out.println(x+d);

        //wrapper object to String : toString()
        String s1 = i1.toString();
        String s2 = i2.toString();
        System.out.println(s1+s2);
    }
}
```

Ex-3: Conversion Process

String to primitive : parseXXX()
 Primitive to String : valueOf()

```
public class Test {
    public static void main(String[] args) {
        //primitive to String: valueOf()
        int a=10,b=20;
        String s1 = String.valueOf(a);
        String s2 = String.valueOf(b);
        System.out.println(s1+s2);

        //String to primitive : parseXXX()
        String str1 = "10";
        String str2 = "20";
```

```

        int x = Integer.parseInt(str1);
        float f = Float.parseFloat(str2);
        System.out.println(x+f);
    }
}

```

ex-4 : Command line arguments example

```

public class Test {
    public static void main(String[] args) {
        System.out.println(args.length);

        System.out.println(args[0]);
        System.out.println(args[1]);
        System.out.println(args[0]+args[1]);

        int a = Integer.parseInt(args[0]);
        float f = Float.parseFloat(args[1]);
        System.out.println(a+f);
    }
}

E:\>java Test 2 8 7 4 1
5
2
8
28
10.0

```

- ✓ The arguments which are passed from command at runtime are called command line arguments.
The command line argument separator is space.
- ✓ All the command line arguments are stored in String[] in the form of string. Then to convert String to primitive format use parseXXX() method.

ex-5 : `toString()` method

- ✓ `toString()` method will be automatically called when we print reference variable in java.
- ✓ `toString()` present in Object class it returns String representation of object (class-name@hashcode).
- ✓ String, StringBuffer, all wrapper classes are overriding `toString()` it returns content of the objects.

```

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

        String s="ratan";
        System.out.println(s);
        System.out.println(s.toString());
    }
}

```

```

StringBuffer sb = new StringBuffer("durga");
System.out.println(sb);
System.out.println(sb.toString());

Integer i = new Integer(100);
System.out.println(i);
System.out.println(i.toString());
}

}

```

Observation 1:

In java it is possible to call `toString()` method only on reference type but not primitive type.

```

Integer i1 = Integer.valueOf(100);
System.out.println(i1.toString());  Valid

```

```

int a=100;
System.out.println(a.toString());      Invalid : error: int cannot be dereferenced

```

Observation 2:

When you want to convert String format data use `toStirng()` method because this method return type is String data.

Conversion of Integer to String:

```

Integer i = new Integer(100);
String s = i.toString();

```

Conversion of StringBuffer to String:

```

StringBuffer sb = new StringBuffer("durga");
String str = sb.toString();

```

Autoboxing & Autounboxing:- (introduced in java 5 version)

- ✓ Up to 1.4 version
 - to convert primitive/String into Wrapper object we are having two approaches,
 - Constructor approach
 - `valueOf()` method
 - wrapper object to primitive use ,
 - `xxxValue()` method
- ✓ From 1.5 onwards the above two operations done automatically.
 - Automatic conversion of primitive to wrapper obj : **Autoboxing** : internally uses `valueOf()`
 - Automatic conversion of wrapper obj to primitive : **Autounboxing** :internally uses `xxxValue()`

Case 1: application without auto boxing & un-boxing

```

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

```

```

    {
        //Conversion of primitive to wrapper
        Integer i = Integer.valueOf(100);
        System.out.println(i);

        // conversion of wrapper to primitive
        Integer x = Integer.valueOf(10);
        int a = x.intValue();
        System.out.println(a);
    }
}

```

Case 2: application with auto boxing & un-boxing

```

class Test
{
    public static void main(String[] args)
    {
        //Autoboxing : automatic conversion of primitive to wrapper
        Integer i = 100;
        System.out.println(i);

        //auto unboxing : automatic conversion of wrapper to primitive
        int a = new Integer(100);
        System.out.println(a);
    }
}

```

Autoboxing & unboxing code after compilation : (.class file code) : open by using java decompiler

```

class Test
{
    public static void main(String[] args)
    {
        Integer integer = Integer.valueOf(100);
        System.out.println(integer);

        int i = (new Integer(100)).intValue();
        System.out.println(i);
    }
}

```

== vs. equals() :

- == operator always check the references (objects) return Boolean value**
- ✓ **equals() method present in object class perform : reference comparison**
- ✓ **String class overriding equals() : content comparison**
- ✓ **StringBuffer class no equals() uses object class equals : reference comparison**
- ✓ **All wrapper classes overriding equals() method : content comparison**

Creation of wrapper object by using constructor approach

```

class Test
{
    public static void main(String[] args)
    {
        Integer i1 = new Integer(10);
        Integer i2 = new Integer(10);
        if (i1==i2)      System.out.println("same");
        else           System.out.println("not same");
    }
}

```

```

        if (i1.equals(i2))      System.out.println("same");
        else                  System.out.println("not same");

        Integer x1 = new Integer(200);
        Integer x2 = new Integer(200);
        if (x1==x2)      System.out.println("same");
        else            System.out.println("not same");

        if (x1.equals(x2))      System.out.println("same");
        else                  System.out.println("not same");
    }
}

Output : not same    same   not same   same

```

Creation of wrapper object by using valueOf() approach:

```

class Test
{
    public static void main(String[] args)
    {
        //This method will always cache values in the range -128 to 127,
        Integer i1 = Integer.valueOf(10);
        Integer i2 = Integer.valueOf(10);
        if (i1==i2)      System.out.println("same");
        else            System.out.println("not same");

        if (i1.equals(i2))      System.out.println("same");
        else                  System.out.println("not same");

        Integer x1 = Integer.valueOf(200);
        Integer x2 = Integer.valueOf(200);
        if (x1==x2)      System.out.println("same");
        else            System.out.println("not same");

        if (x1.equals(x2))      System.out.println("same");
        else                  System.out.println("not same");
    }
}

Output : same  same  not same  same

```

Factory method: [Factory design pattern]

- ✓ One java class method returns same class object or different class object is called factory method.
- ✓ There are three types of factory methods in java.
 - **Instance factory method.** (method is called by using reference variable)
 - **Static factory method.** (Method is called by using class name)
 - **Pattern factory method.** (method is returning different class object).

Note : method returns the object is called factory method.

```

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

```

```

Integer i = Integer.valueOf(100);
System.out.println(i);
Runtime r = Runtime.getRuntime();
System.out.println(r);

//instance factory method
String str="ratan";
String str1 = str.concat("soft");
System.out.println(str1);

//pattern factory method
Integer a1 = Integer.valueOf(100);
String ss = a1.toString();
System.out.println(ss);
}

}

```

ex: Achieving polymorphism concept by using wrapper classes.

```

class Test
{
    void disp(Object o)
    {
        System.out.println("Result value="+o);
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.disp(new Integer(10));
        t.disp(new Float(10.5));
        t.disp(new Character('a'));
    }
}

```

ex: Another approach to create wrapper object.

```

public class Test {
    public static void main(String[] args) {
        Integer intWrapper = Integer.valueOf("12345");

        //Converting from binary to decimal
        Integer intWrapper2 = Integer.valueOf("11011", 2);

        //Converting from hexadecimal to decimal
        Integer intWrapper3 = Integer.valueOf("D", 16);

        System.out.println("Value of intWrapper Object: "+ intWrapper);
    }
}

```

```

        System.out.println("Value of intWrapper2 Object: "+ intWrapper2);
        System.out.println("Value of intWrapper3 Object: "+ intWrapper3);
        System.out.println("Hex value of intWrapper: " + Integer.toHexString(intWrapper));
        System.out.println("Binary Value of intWrapper2: "+ Integer.toBinaryString(intWrapper2));
    }
}

E:\>java Test
Value of intWrapper Object: 12345
Value of intWrapper2 Object: 27
Value of intWrapper3 Object: 13
Hex value of intWrapper: 3039
Binary Value of intWrapper2: 11011

```

ex : compareTo() method

```

public class Test {
    public static void main(String[] args) {
        Integer intObj1 = new Integer (25);
        Integer intObj2 = new Integer ("25");
        Integer intObj3= new Integer (35);
        //compareTo demo
        System.out.println("Comparing using compareTo Obj1 and Obj2: " + intObj1.compareTo(intObj2));
        System.out.println("Comparing using compareTo Obj1 and Obj3: " + intObj1.compareTo(intObj3));
        System.out.println("Comparing using compareTo Obj1 and Obj3: " + intObj3.compareTo(intObj1));
    }
}

```

```

E:\>java Test
Comparing using compareTo Obj1 and Obj2: 0
Comparing using compareTo Obj1 and Obj3: -1
Comparing using compareTo Obj1 and Obj3: 1

```

Wrapper classes interview questions

1. What is the purpose of wrapper classes?
2. How many Wrapper classes present in java what are those?
3. How many ways are there to create wrapper objects?
4. Define mutable & immutable?
5. When we will get NumberFormatException?
6. What do you mean by factory method?
7. What is the purpose of valueOf() method is it factory method or not?
8. How to convert wrapper objects into corresponding primitive values?
9. What is the implementation of toString() in all wrapper classes?
10. How to convert String into corresponding primitive?
11. What do you mean by Autoboxing and Autounboxng & introduced in which version?

12. Purpose of `parseXXX()` & `xxxValue()` method?
13. Which Wrapper classes is direct child class of `Object` class?
14. Which Wrapper classes is direct child class of `Number` class?
15. How to convert primitive to String?
16. When we will get compilation error like "int cannot be dereferenced"?
17. Wrapper classes are immutable classes or mutable classes?
18. Autoboxing internally uses which method performs operations?
19. What the difference creating wrapper object by using constructor & `valueOf()`?
20. Perform following conversions
 - a. `int-->String`
 - b. `String-->int`
 - c. `Integer-->int`
 - d. `int-->Integer` ?

***** **Thank you : completed** *****

RATAN

Exception Handling

Introduction:

- ✓ Dictionary meaning of the exception is abnormal termination.
- ✓ Exception is a object occurred at runtime to disturb the normal flow of the execution.
- ✓ **An exception is an event that occurs during execution of the program that disturbs normal flow of the program instructions.**
- ✓ An unexpected even that disturbs the normal termination of the application is called exception.

In application whenever the exception occurred,

1. Program terminated abnormally
2. Rest of the application is not executed.

To overcome above limitation in order to execute the rest of the application & to get normal termination of the application must handle the exception.

There are two ways to handle the exceptions in java.

- 1) By using try-catch block.
- 2) By using throws keyword.

Exception Handling:

- ✓ The main objective of exception handling is,
 - a. To get normal termination of the application
 - b. To execute the rest of the application code.
- ✓ Exception handling means just we are providing alternate code to continue the execution of remaining code & to get normal termination of the application.
- ✓ Every Exception is a predefined class present in different packages.

<i>java.lang.ArithmaticException</i>	<i>java.lang</i>
<i>java.io.IOException</i>	<i>java.io</i>
<i>java.sql.SQLException</i>	<i>java.sql</i>

Types of Exceptions:

As per the sun micro systems standards The Exceptions are divided into three types

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

Unchecked Exception:

- ✓ The exceptions which are not checked by the compiler are called unchecked Exception.
ArithmaticException,ArrayIndexOutOfBoundsException,NumberFormatException....etc
- ✓ The classes that extend *RuntimeException* class are called unchecked exceptions.

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println(10/0);           java.lang.ArithmaticException: / by zero

        int[] a={10,20,30};
        System.out.println(a[5]);          java.lang.ArrayIndexOutOfBoundsException: 5

        System.out.println("ratan".charAt(12)); java.lang.StringIndexOutOfBoundsException
    }
}
```

```

        }
    }
}

```

- ✓ If the application contains un-checked Exception code is compiled but at runtime JVM display exception message & program terminated abnormally.
- ✓ To overcome runtime problem must handle the exception either using try-catch blocks or by using throws keyword.

Checked Exception:

- ✓ The Exceptions which are checked by the compiler are called Checked Exceptions.
IOException,SQLException,InterruptedException.....etc
- ✓ The classes that extends Exception class are called checked exceptions.

```

import java.io.*;
class Test
{
    public static void main(String[] args)
    {
        FileInputStream fis = new FileInputStream("abc.txt");    FileNotFoundException
    }
}

```

- ✓ If you are trying to compile the above compilation the compiler will show the compilation error.
error: unreported exception FileNotFoundException; must be caught or declared to be thrown
- ✓ If the application contains checked Exception code is not compiled, the compiler will give the exception information in the form of compilation error but exception occurred at runtime.
- ✓ To overcome above problem to compile the application must declare the try-catch blocks or throws keyword then only code is compiled.

Note: Whether it is a checked Exception or unchecked exception exceptions are raised at runtime but not compile time.

Note: whether it is a checked Exception or unchecked Exception must handle the Exception by using try-catch blocks or throws keyword to get normal termination of application & to execute rest of the application.

Checked Exception scenarios:

1) java.lang.InterruptedException

When we used **Thread.sleep(2000)**; your thread is entered into sleeping mode, then other threads are able to interrupt, in this case program is terminated abnormally & rest of the application is not executed.

To overcome above problem compile time compiler is checking that exception & displaying exception information in the form of compilation error.

Based on compiler generated error message write the try-catch blocks or throws , if runtime any exception raised the try-catch or throws keyword executed program is terminated normally.

2) ***Java.io.FileNotFoundException***

If we are trying to read the file from local disk but at runtime if the file is not available program is terminated abnormally rest of the application is not executed.

To overcome above problem compile time compiler is checking that exception & displaying exception information in the form of compilation error.

Based on compiler generated error message write the try-catch blocks or throws , if runtime any exception raised the try-catch or throws keyword executed program is terminated normally.

3) ***Java.sql.SQLException***

If we are trying to connect to data base but at runtime data base is not available program is terminated abnormally rest of the application is not executed.

To overcome above problem compile time compiler is checking that exception & displaying exception information in the form of compilation error.

Based on compiler generated error message write the try-catch blocks or throws , if runtime any exception raised the try-catch or throws keyword executed program is terminated normally.

Exception vs. Error:

- ✓ The exception are occurred due to several reasons

- a. Developer mistakes
- b. End-user input mistakes.
- c. Resource is not available
- d. Networking problems.

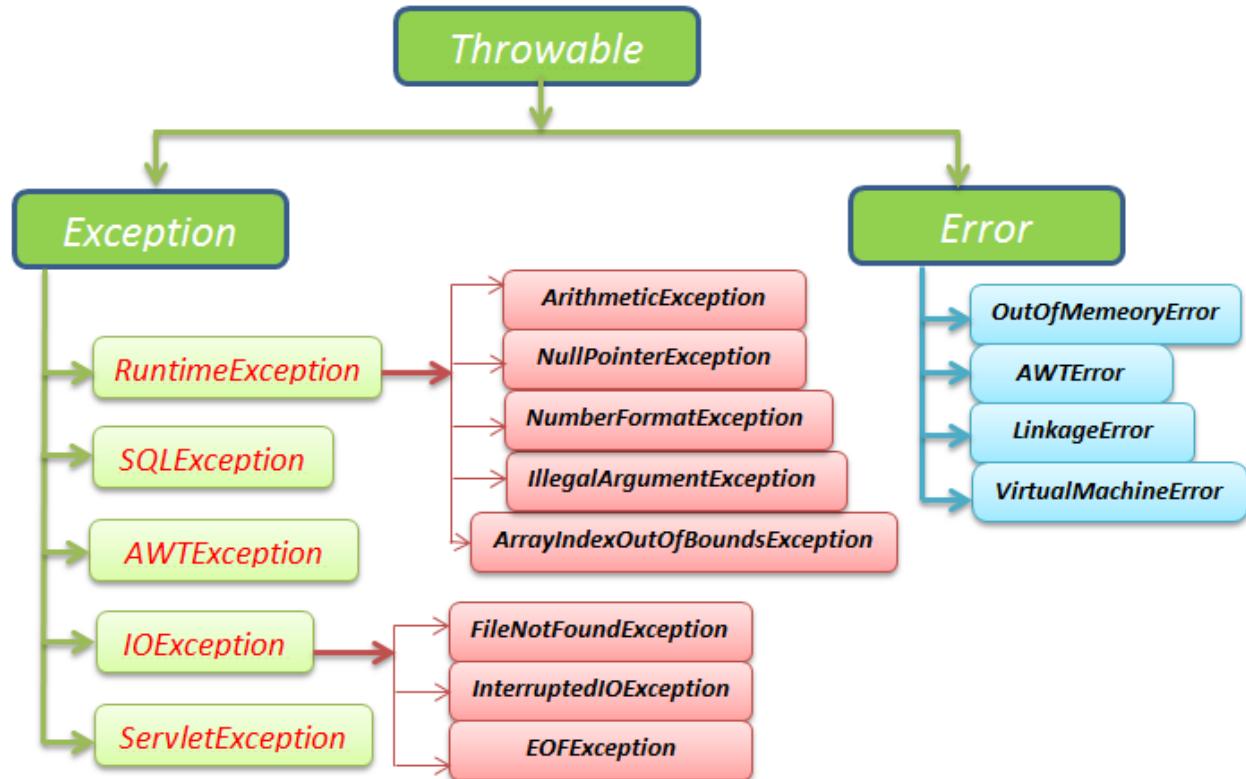
But errors are caused due to lack of system resources.

StackOverflowError, OutOfMemoryError, AssertionError.....etc

- ✓ It is possible to handle the exceptions by using try-catch blocks or throws keyword but it is not possible to handle the errors.
- ✓ Error is an un-checked type exception.

```
class Test
{
    public static void main(String[] args)
    {
        Test[] t = new Test[100000000];           "java.lang.OutOfMemoryError"
    }
}
```

Exception Handling Tree Structure:



- ✓ The root class of exception handling is **Throwable** class.
- ✓ In above tree Structure **RuntimeException** its child classes & **Error** its child classes are Unchecked remaining all exceptions are checked Exceptions.

Exception handling keywords:

1. Try
2. Catch
3. Finally
4. Throws
5. Throw

Default exception handler:

Whenever the exception raised default exception handler is responsible to create exception object & print the exception message.

Fully Checked vs. partially checked:

- ✓ The root class & all its child class are checked then that root class is called **fully checked exception**.
IOException, SQLException....etc
- ✓ The root class contains some child classes are checked exceptions & some child classes are un-checked exception then that root class is called **partially checked exception**.
Exception , Throwable..etc

There are two ways to handle the exceptions in java.

- 1) By using try-catch block.

2) By using throws keyword.

Exception handling by using Try –catch blocks:

Syntax:

```
try
{
    exceptional code;
}
catch (Exception_Name reference_variable)
{
    Alternate code: Code to run if an exception is raised
}
```

ex-1 : Whenever exception raised in the try block, the corresponding catch block executed.

Application without try-catch blocks

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("ratan");
        System.out.println(10/0);
        System.out.println("rest of the application");
    }
}
```

E:\>java Test
ratan
Exception : ArithmeticException: / by zero

Disadvantages:

1. program terminated abnormally.
2. rest of the application not executed.

Application with try-catch blocks:

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

E:\>java Test
ratan
5

Rest of the application

Advantages:

1. Program terminated normally
2. Rest of the application executed

ex-2 : In below example catch block is not matched hence program is terminated abnormally.

```

try
{
    System.out.println("sravya");
    System.out.println(10/0);
}
catch(NullPointerException e)
{
    System.out.println(10/2);
}

```

ex 3: If there is no exception in try block the corresponding catch blocks are not checked.

```

class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("sravya");
        }
        catch(NullPointerException e)
        {
            System.out.println(10/2);
        }
        System.out.println("rest of the app");
    }
}
E:\sravya>java Test
sravya
rest of the app

```

ex - 4: In Exception handling independent try blocks declaration are not allowed.

- Only try : invalid
- Only catch : invalid
- Try-catch : valid
- Try-finally : valid
- Try-with-resources : valid

```

class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("sravya");
        }
        System.out.println("rest of the app");
    }
}
E:\sravya>javac Test.java
Test.java:4: 'try' without 'catch' or 'finally' or resources

```

ex- 5: In between try-catch blocks it is not possible to declare any statements.

```

try
{
    System.out.println(10/0);
}
System.out.println("anu");
catch(ArithmeticException e)
{
    System.out.println(10/2);
}

```

ex 6:

- ✓ If the exception raised in other than try block it is always abnormal termination.
- ✓ In below example exception raised in catch block hence program is terminated abnormally.

```

try
{
    System.out.println(10/0);
}
catch(ArithmeticException e)
{
    System.out.println(10/0);
}

```

ex- 7:

- ✓ If the exception raised in try block the remaining code of try block is not executed.
- ✓ Once the control is out of the try block the control never entered into try block once again.
- ✓ Don't take normal code inside try block because no guarantee all statements in try-block will be executed or not.

```

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

```

E:\sravya>java Test
Durga
ratan
rest of the app

```

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

```

E:\sravya>java Test
5
rest of the app

Case 1 : The way of handling the exceptions is varied from exception to the exception hence it is recommended to write try with multiple catch blocks.

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Scanner s=new Scanner(System.in);
        System.out.println("enter the division value");
        int n=s.nextInt();
        try
        {
            System.out.println(10/n);
            System.out.println("ratan".charAt(10));
        }
        catch (ArithmaticException ae)
        {
            System.out.println("Ratanit");
        }
        catch (StringIndexOutOfBoundsException se)
        {
            System.out.println("durgasoft");
        }
        System.out.println("rest of the code");
    }
}
```

Output: enter the division value: 5
Write the output

Output: enter the division value: 0
Write the output

Case 2 : By using **Exception(root class)** class catch block it is possible to hold any type of exceptions.

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Scanner s=new Scanner(System.in);
        System.out.println("enter division value");
        int n=s.nextInt();
        try{
            System.out.println(10/n);
            System.out.println("ratan".charAt(10));
        }
        catch (Exception e)
        {
            System.out.println("Ratanit="+e);
        }
        System.out.println("rest of the code");
    }
}
```

Output: enter division value: 5
Write the output

Output: enter division value: 0
Write the output

Case 3: When we declare multiple catch blocks then the catch block order must be **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);
            System.out.println("ratan".charAt(20));
        }
        //catch block order is child to parent
        catch (ArithmaticException ae)
        {
            System.out.println("Exception"+ae);
        }
        catch (Exception ne)
        {
            System.out.println("Exception"+ne);
        }
        System.out.println("rest of the code");
    }
}

```

Output: provide the division value: 5
Write the output

Output: provide the division value: 0
Write the output

Case 4 : Invalid : catch block order is parent to child compiler generate error message.

```

try
{
    System.out.println(10/n);
    System.out.println("ratan".charAt(20));
}
catch (Exception ne)
{
    System.out.println("Exception"+ne);
}
catch (ArithmaticException ae)
{
    System.out.println("Exception"+ae);
}

```

G:\>javac Test.java
error: exception ArithmaticException has already been caught

Category -2 (pipe symbol)

Case 1:-

It is possible to handle more than one exception in single catch by using pipe(|) symbol.(java 7 version)

```
catch(ArithmaticException | StringIndexOutOfBoundsException a).
catch(NumberFormatException | NullPointerException | StringIndexOutOfBoundsException a)
```

```
import java.util.Scanner;
import java.io.*;
public class Test
{ public static void main(String[] args)
{ Scanner s = new Scanner(System.in);
System.out.println("enter a number");
int n = s.nextInt();
try { System.out.println(10/n);
System.out.println("ratan".charAt(13));
}
catch(ArithmaticException | ClassCastException a)
{ System.out.println("exception info="+a);
}
catch(NumberFormatException | NullPointerException | StringIndexOutOfBoundsException a)
{ System.out.println("exception info="+a);
}
System.out.println("Rest of the application");
}
}
```

- ✓ When we declared unchecked exception in catch block by using pipe symbol, those exceptions are not mandatory to present in try block.
- ✓ In above example,
 - In catch we declared ArithmaticException : this exception raised in try block.(chance)
 - In catch we declared ClassCastException : this exception not raised in try block.(no chance)

Case 2 :

```
import java.io.*;
class Test
{ public static void main(String[] args)
{ try
{ FileInputStream f = new FileInputStream("abc.txt");
}
catch(FileNotFoundException | InterruptedException a)
{ System.out.println("exception info="+a);
}
}
}
```

error: exception InterruptedException is never thrown in body of corresponding try statement

- ✓ When we declared checked exception in catch block by using pipe symbol, those exceptions must present in try block otherwise compiler generates error message.
- ✓ In catch block two exceptions are declared but try block contains one exception hence compiler generates error message.

Case 3: valid

```

import java.io.*;
class Test
{
    public static void main(String[] args)
    {
        try
        {
            FileInputStream f = new FileInputStream("abc.txt");
            Thread.sleep(1000);
        }
        catch(FileNotFoundException | InterruptedException a)
        {
            System.out.println("exception info="+a);
        }
    }
}
  
```

- ✓ When we declared checked exception in catch block by using pipe symbol, those exceptions must present in try block.
- ✓ In catch block two exceptions are declared & try block contains those two exceptions it is valid.

Case-4: By using pipe symbol possible it is possible to declare the both checked exceptions & un-checked exception but checked exceptions must be present in try block.

```

public static void main(String[] args)
{
    try
    {
        FileInputStream f = new FileInputStream("abc.txt");
    }
    catch(FileNotFoundException | ArithmeticException a)
    {
        System.out.println("exception info="+a);
    }
}
  
```

Case-5:

- ✓ It is not possible to declare the both parent & child classes by using pipe symbol.
- ✓ Here the FileNotFoundException is the child class of IOException

Invalid:

```

catch(FileNotFoundException | IOException a)
{
    System.out.println("exception info="+a);
}
  
```

Valid :

```

catch( IOException a)
{
    System.out.println("exception info="+a);
}
  
```

Category-3 try with resources (java 7 version)

When we declare the resource by using try block once the try block is completed resource is released.

Case 1:

- ✓ When we declare the try with resource, if the resource is throws unchecked exception in this case catch block is optional.
- ✓ In below example we declared scanner class as a resource it may raise InputMismatchException & it is unchecked exception hence catch block is optional.

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        try(Scanner s = new Scanner(System.in))
        {
            System.out.println("enter id");
            int a = s.nextInt();
            System.out.println("input value="+a);
        }
    }
}
```

Case 2:

- ✓ When we declare the try with resource, if the resource is throws checked exception in this case catch block is mandatory.
- ✓ In above example we declared File resource, it throws FileNotFoundException it is a checked exception hence catch block is mandatory.

```
import java.io.*;
class Test
{
    public static void main(String[] args)
    {
        try(FileInputStream fis = new FileInputStream("abc.txt"))
        {
            System.out.println("reading data from text file");
        }
        catch (FileNotFoundException e)
        {
            System.out.println("file is not available");
        }
    }
}
```

Case 3:

- ✓ By using try block it is possible to declare more than one resource but every resource is separated with semicolon.
- ✓ If the try block contains more than one resource in those resources at least one resource throws checked exception in such case catch block is mandatory.

```
try(Scanner s = new Scanner(System.in);FileInputStream fis = new FileInputStream("abc.txt"))
{
    //some code here
}
catch (FileNotFoundException e) {      //some code here      }
```

Note: the resources are internally calling close() method to close the resources.

```
public interface java.io.Closeable extends java.lang.AutoCloseable {
    public abstract void close() throws java.io.IOException; }
```

ex 1 : There are three methods to print Exception information

```

1) toString()
2) getMessage()
3) printStackTrace()

class Test
{
    void m3()
    {
        try{ System.out.println(10/0); }
        catch(ArithmeticException ae)
        {
            System.out.println(ae.toString());
            System.out.println(ae.getMessage());
            ae.printStackTrace();
        }
    }
    void m2()
    {
        m3();
    }
    void m1()
    {
        m2();
    }
    public static void main(String[] args)
    {
        new Test().m1();
    }
}
D:\DP>java Test
java.lang.ArithmetricException: / by zero
//toString() method output
//getMessage() method output
//printStackTrace() method
java.lang.ArithmetricException: / by zero
at Test1.m3(Test1.java:8)
at Test1.m2(Test1.java:5)
at Test1.m1(Test1.java:3)
at Test1.main(Test1.java:17)

```

Note: Internally JVM uses *printStackTrace()* method to print exception information.

ex 2: is it possible to pass exceptions as method parameters.

```

import java.io.*;
class Test
{
    void m1(ArithmetricException e)
    {
        System.out.println("m1 method code="+e);
    }
    void m1(Exception ee)
    {
        System.out.println("m2 method code="+ee);
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1(new ArithmetricException());
        t.m1(new IOException());
    }
}

```

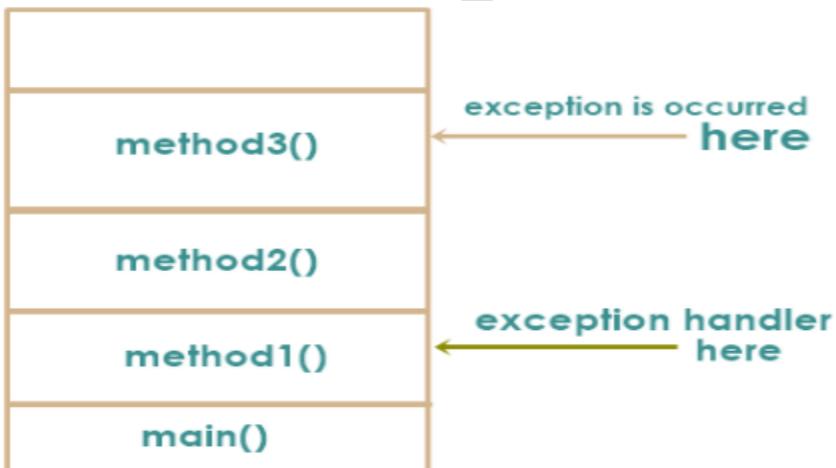
ex-3: Exception propagation

If the exception raised in top of the stack method but if you are not handled it drops down to the stack previous method, if you are not catch it drop down until end of the stack(up to main method) this is called exception propagation.

Note: only the unchecked Exceptions are propagated automatically but not checked.

```
class Test
{
    void m3()
    {
        System.out.println(10/0);
    }
    void m2()
    {
        m3();
    }
    void m1()
    {
        try{    m2();    }
        catch(ArithmaticException ae)
        {
            System.out.println("Arithmatic Exception propagation..... ");
        }
    }
    public static void main(String[] args)
    {
        new Test().m1();
    }
}
```

- ✓ In above example the exception raised in m3() method but it is not handled so it is propagated to m2() method.
- ✓ Here the m2() method is not handled exception so it is propagated to m1().
- ✓ In above example m1() is handled exception.



Possibilities of try-catch:

Case-1

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

Case-2

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

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

Case-3

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

Case-4

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

Case-5

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

Case-6

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

Finally block:

Finally block code is always executed irrespective of try and catch block code. It is used to write the resource releasing code like,

- a. connection closing.
- b. streams closing.
- c. channel closing
- d. Object destruction .

```
Connection.close();
inputstreamclose();
scanner.close();
Test t = new Test();t=null;
```

Finally block Syntax:-

```
try
{ risky code;
}
catch (Exception obj)
{   code to be run if the exception raised (handling code);
}
finally
{   Clean-up code;(database connection closing , streams closing.....etc)
}
```

final vs return statement :

if the try,catch,finally contains return statement the return value will be finally block return value.

```
class Test
{
    int m1()
    {
        try
        {
            return 10;
        }
        catch(Exception e)
        {
            return 20;
        }
        finally
        {
            return 30;
        }
    }
    public static void main(String[] args)
    {
        int a = new Test().m1();
        System.out.println("return value="+a);
    }
}
```

Output
G:\>java Test
return value=30

All possibilities of finally block execution :

Case 1:

```

try
{
    System.out.println("try");
}
catch (ArithmaticException ae)
{
    System.out.println("catch");
}
finally
{
    System.out.println("finally");
}

```

Output:

Try
finally

case 3:

```

try
{
    System.out.println(10/0);
}
catch (NullPointerException ae)
{
    System.out.println("catch");
}
finally
{
    System.out.println("finally");
}

```

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

case 5:

```

try
{
    System.out.println("try");
}
catch(ArithmaticException ae)
{
    System.out.println("catch");
}
finally
{
    System.out.println(10/0);
}

```

Output:-
*try
Exception in thread "main"
java.lang.ArithmaticException: / by zero*

case 2:

```

try
{
    System.out.println(10/0);
}
catch (ArithmaticException ae)
{
    System.out.println("catch");
}
finally
{
    System.out.println("finally");
}

```

Output:

catch
finally

case 4:

```

try
{
    System.out.println(10/0);
}
catch (ArithmaticException ae)
{
    System.out.println(10/0);
}
finally
{
    System.out.println("finally");
}

```

Output:
finally
*Exception in thread "main"
java.lang.ArithmaticException: / by zero
at Test.main(Test.java:7)*

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

```

try
{
    System.out.println("try");
}
finally
{
    System.out.println("finally");
}
System.out.println("rest of the code");

```

Output:-
try
finally
rest of the code

In two cases finally block won't be executed

Case 1: whenever the control is entered into try block then only finally block will be executed otherwise it is not executed.

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println(10/0);
        try
        {
            System.out.println("ratan");
        }
        finally
        {
            System.out.println("finally block");
        }
        System.out.println("rest of the code");
    }
}
```

Case 2: In your program when we used System. Exit (0) the JVM will be shutdown hence the rest of the code won't be executed .

```
class Test
{
    public static void main(String[] args)
    {
        try{ System.out.println("ratan");
        System.exit(0);
        }
        finally
        {
            System.out.println("finally block");
        }
        System.out.println("rest of the code");
    }
}
D:\>java Test
Ratan
```

Example :-if the try,catch finally blocks contains exception the default exception handler is able to display only one exception at a time that most recently raised.

```
try
{
    System.out.println(10/0);
}
catch(Exception e)
{
    System.out.println("ratan".charAt(20));
}
finally
{
    int[] a={10,20,30};
    System.out.println(a[9]);
}
```

G:\>java Test

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Array index out of range: 9

Example :
statement 1

```

statement 2
try
{
    statement 3
    statement 4
    try
    {
        statement 5
        statement 6
    }
    catch ()
    {
        statement 7
        statement 8
    }
}
catch ()
{
    statement 9
    statement 10
    try
    {
        statement 11
        statement 12
    }
    catch ()
    {
        statement 13
        statement 14
    }
}
Finally{
statement 15
statement 16
}
Statement -17
Statement -18

```

Case 1: No Exception in the above example.
1, 2, 3, 4, 5, 6, 15, 16, 17, 18 Normal Termination

Case 2:-if the exception is raised in statement 2.
1 , Abnormal Termination

*Case 3:-Exception is raised in the statement 3
the corresponding catch block Is
not matched.*
1,2,15,16Abnormal termination

*Case 4:- Exception is raise in the statement-4
corresponding catch block is
matched.*
1,2,3,9,10,11,12,15,16,17,18NT

*Case 5:-Exception is raised in the statement 5
and corresponding catch block
is matched.*
1,2,3,4,7,8,15,16,17,18 NT

*Case 6:-If the exception is raised in the
statement 6 and corresponding
catch block is notmatched but
outer catch block is matched.*
1,2,3,4,5,9,10,11,12,15,16,17,18 NT

Case 7:- *If the exception is raised in the statement 5 and the corresponding catch block is matched but while executing catch block exception raised in statement-7, the outer catch block is matched while executing outer catch exception raised in st-11, the inner catch block is matched but while executing inner catch the exception raised in st-13.*
1,2,3,4,9,10,15,16Abnormal termination.

Case 8:- *If the exception is raised in the statement 6 and the corresponding catch block is matched but while executing catch block exception raised in statement-8, the outer catch block is matched while executing outer catch exception raised in st-12, the inner catch block is matched but while executing inner catch the exception raised in st-14.*
1,2,3,4,5,7,9,10,11,13,15,16Abnormal termination.

Case 9:- *If the exception raised in statement 15.* **1,2,3,4,5 Abnormal termination.**

Case 10:- *if the Exception raised in statement 18.* **1,2,3,4,5,6,15,16,17 Abnormal termination.**

Throws keyword:

There are two approaches to handle the exceptions in java

- a. By using try-catch blocks.
- b. By using throws keyword.

Handling exception by using Try-catch

1. Try-catch blocks are used to write the exception handling code.
2. By using try-catch blocks it is possible to handle multiple exceptions by using multiple catch blocks.
3. We can write the try-catch blocks at method implementation level.
4. We can provide the try-catch blocks at method & constructor & blocks level.

Handling Exception by using throws keyword

Throws keyword is used to delegate the responsibilities of exception handling to caller method.

By using throws it is possible to handle multiple exceptions because one method is able to throw multiple exceptions at time.

We can write the throws keyword at method declaration level.

We can provide the throws keyword only at method & constructor level but not block level.

Example 1:

```
class Test
{
    void studentDetails() throws InterruptedException
    {
        System.out.println("suneel babu is sleeping");
        Thread.sleep(3000);
        System.out.println("do not disturb sir.....");
    }

    void hod()throws InterruptedException
    {
        studentDetails();
    }

    void principal()
    {
        try{hod();}
        catch(InterruptedException ie)
        {
            ie.printStackTrace();
        }
    }

    void officeBoy()
    {
        principal();
    }

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

- ✓ In above example exception raised in studentDetails() method but it delegating responsibilities of exception handling to hod() method by using throws keyword.
- ✓ But hod() method delegating responsibilities of exception handling to principal() method by using throws now principal handing this exception by using try-catch blocks.

Example 2:-

```
class Test
```

```

{
    void studentDetails() throws InterruptedException
    {
        System.out.println("suneel babu is sleeping");
        Thread.sleep(3000);
        System.out.println("do not disturb sir..... ");
    }
    void hod()throws InterruptedException
    {
        studentDetails();
    }
    void principal()throws InterruptedException
    {
        hod();
    }
    void officeBoy()throws InterruptedException
    {
        principal();
    }
    public static void main(String[] args) throws InterruptedException
    {
        Test t = new Test();
        t.officeBoy();
    }
}

```

- ✓ In above example method-by-method using throws keyword to delegate responsibilities of exception handling to caller method.
- ✓ At final main() method uses throws keyword to delegate the responsibilities of exception handling to JVM.

Example 3:- One method is able to throws more than one exception.

```

import java.io.*;
class Test
{
    void m2()throws FileNotFoundException,InterruptedException
    {
        FileInputStream fis = new FileInputStream("abc.txt");
        Thread.sleep(2000);
        System.out.println("Exceptions are handled");
    }
    void m1()
    {
        try{m2();}
        catch(FileNotFoundException | InterruptedException f){f.printStackTrace();}
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
    }
}

```

Example 4:- The root class is able to throws all exceptions (Exception root class)

```
import java.io.*;
```

```

class Test
{
    void m2()throws Exception
    {
        FileInputStream fis = new FileInputStream("abc.txt");
        Thread.sleep(2000);
        System.out.println("Exceptions are handled");
    }
    void m1()
    {
        try{m2(); }
        catch(Exception e){ e.printStackTrace(); }
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
    }
}

```

Example 5:-

```

import java.io.*;
class Test
{
    void m2()throws FileNotFoundException,InterruptedException
    {
        FileInputStream fis = new FileInputStream("abc.txt");
        Thread.sleep(2000);
        System.out.println("Exceptions are handled");
    }
    void m1()throws InterruptedException
    {
        try{m2();}
        catch(FileNotFoundException fn){fn.printStackTrace();}
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        try{ m1();}
        catch(InterruptedException ie){ie.printStackTrace();}
    }
}

```

- ✓ m2() method delegated two exceptions to caller method.
- ✓ The caller method is handled one exception(FileNotFoundException)& delegated one exception(InterruptedException) to caller method.
- ✓ The main method is handled that exception.

Throw keyword:-

In the application whenever the exception raised the JVM will create the exception object & JVM will print predefined exception message.

Case 1: It is possible to create the exception object explicitly by developer,

Step 1:- create the Exception object explicitly by the developer by using new keyword.

```
new ArithmeticException("ratan not eligible");
```

Step 2:- handover user created Exception object to jvm by using throw keyword.

```
throw new ArithmeticException("ratan not eligible");
```

The above approach is not recommended because ArithmeticException is predefined exception & it contains some fixed meaning(/ by zero)

Case 2: *InvalidAgeException : userdefined exception :-*

Step 1:- create the Exception object explicitly by the developer by using new keyword.

```
new InvalidAgeException ("ratan not eligible");
```

Step 2:- handover user created Exception object to jvm by using throw keyword.

```
throw new InvalidAgeException ("ratan not eligible");
```

the above approach is recommended because user defined must handled by user only.

Note: - throw keyword is used to handover user created exception object to JVM whether it is predefined exception class or user defined exception class but it is always recommended throw custom exception.

Example:-

```
import java.util.*;
class Test
{
    static void validate(int age)
    {
        if (age>18)
        {
            System.out.println("eligible for mrg");
        }
        else
        {
            throw new ArithmeticException("not eligible for marriage");
        }
    }
    public static void main(String[] args)
    {
        Scanner s=new Scanner(System.in);
        System.out.println("please enter your age ");
        int age = s.nextInt();
        Test.validate(age);
        System.out.println("rest of the code");
    }
}
```

E:\>java Test
please enter your age
45
Check the output

E:\>java Test
please enter your age
10
Check the output

The above example is not recommended because we are creating object of ArithmeticException but is contains some fixed meaning(/ by zero).

Customization of exception handling : user defined exception:-

There are two types of user defined exceptions

1. User defined checked exception.

- a. Default constructor approach.
 - b. Parameterized constructor approach.
2. User defined un-checked Exception.
- a. Default constructor approach.
 - b. Parameterized constructor approach.

Note: - while declaring user defined exceptions: the naming conventions are every exception suffix must be the word Exception.

Creation of userdefined checked Exception by using default constructor approach:-

Step-1:- create the user defined checked Exception

Normal java class will become Exception class whenever we are extends Exception class.

InvalidAgeException.java:-

```
package com.tcs.userexceptions;
public class InvalidAgeExcepiton extends Exception
{
    //default constructor
};
```

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

Test.java

```
package com.tcs.project;
import com.tcs.userexceptions.InvalidAgeExcepiton;
import java.util.Scanner;
class Test
{
    static void status(int age)throws InvalidAgeExcepiton
    {
        if (age>25)
        {
            System.out.println("eligible for mrg");
        }
        else
        {
            throw new InvalidAgeExcepiton(); //default constructor executed
        }
    }
    public static void main(String[] args) throws InvalidAgeExcepiton
    {
        Scanner s = new Scanner(System.in);
        System.out.println("enter u r age");//23
        int age = s.nextInt();
        Test.status(age);
    }
}
```

```
D:\morn11>java com.tcs.project.Test
enter u r age
20
Exception in thread "main" com.tcs.userexceptions.InvalidAgeExcepiton
```

Creation of userdefined checked exception by using parameterized constructor approach.

step-1:- create the userdefined checked exception class.

Normal java class will become checked exception class when we extend Exception class.

InvalidAgeException.java

```
package com.tcs.userexceptions;
public class InvalidAgeExcepiton extends Exception
{
    public InvalidAgeExcepiton(String str)
    {super(str); //super constructor calling in order to print your information
    }
}
```

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

Test.java

```
package com.tcs.project;
import com.tcs.userexceptions.InvalidAgeExcepiton;
import java.util.Scanner;
class Test
{
    static void status(int age)throws InvalidAgeExcepiton
    {
        if (age>25)
        {
            System.out.println("eligible for mrg");
        }
        else
        {
            throw new InvalidAgeExcepiton("not eligible try after some time");
        }
    }
    public static void main(String[] args)throws InvalidAgeExcepiton
    {
        Scanner s = new Scanner(System.in);
        System.out.println("enter u r age");
        int age = s.nextInt();
        Test.status(age);
    }
}
```

D:\morn11>javac -d . InvalidAgeExcepiton.java

D:\morn11>javac -d. Test.java

D:\morn11>java com.tcs.project.Test

enter u r age

28

eligible for mrg

D:\morn11>java com.tcs.project.Test

enter u r age

20

Exception in thread "main" com.tcs.userexceptions.InvalidAgeExcepiton: not eligible try after some time

Differences between checked Exception & unchecked Exception:-

User checked Exception

- Our normal java class will become checked Exception class when extends Exception class.

```
class InvalidAgeException extends Exception
{
    //logics here
}
```

- Must handle the checked Exceptions by using try-catch block or throws keyword.

User un-checked Exception

- Our normal java class will become un-checked Exception class when extends Exception class.

```
class InvalidAgeException extends RuntimeException
{
    //logics here
}
```

- Handling unchecked Exceptions is optional but it is recommended.

Different types of exceptions

ArrayIndexOutOfBoundsException:-

```
int[] a={10,20,30};
System.out.println(a[4]);//ArrayIndexOutOfBoundsException
```

NumberFormatException:-

```
String str1="abc";
int b=Integer.parseInt(str1);
System.out.println(b); //NumberFormatException
```

NullPointerException:-

```
String str1=null;
System.out.println(str1.length()); //NullPointerException
```

ArithmaticException:-

```
int b=10/0;
System.out.println(b); //ArithmaticException
```

IllegalArgumentception:-

```
Thread priority range is 1-10
1--->low priority      10--->high priority
Thread t=new Thread();
t.setPriority(11); //IllegalArgumentception
```

IllegalThreadStateException:-

```
Thread t=new Thread();
t.start();
t.start(); //IllegalThreadStateException
```

StringIndexOutOfBoundsException:-

```
String str="rattaiah";
System.out.println(str.charAt(13)); //StringIndexOutOfBoundsException
```

NegativeArraySizeException:-

```
int[] a=new int[-9];
System.out.println(a.length); //NegativeArraySizeException
```

InputMismatchException:-

```
Scanner s=new Scanner(System.in);
```

```

System.out.println("enter first number");
int a=s.nextInt();
D:\>java Test
enter first number
ratan
Exception in thread "main" java.util.InputMismatchException

```

ClassCastException:-

```

String s = new String("ratan");
Object o = (Object)s;
Object oo = new Object();
String str = (String)oo;           // java.lang.ClassCastException

```

java.lang.NoClassDefFoundError vs java.lang.ClassNotFoundException:-

```

class Test1
{
    void m1(){      System.out.println("Test1 class m1()");  }
}
class Test
{
    public static void main(String[] args) throws ClassNotFoundException
    {
        Test1 t = new Test1();
        t.m1();
        Class.forName("Emp");
    }
}

```

Observation-1:- In Test class we are hard coding Test1 object but in target location Test1.class file is not available it will generate **java.lang.NoClassDefFoundError**.

Observation-2:- In java to load .class file dynamically at runtime we are using forName() method but if runtime the class is not available it generate **java.lang.ClassNotFoundException**.

Different types of Errors:-**StackOverflowError****OutOfMemoryError:-**

```

class Test
{
    public static void main(String[] args)
    {
        int[] a=new int[100000000];   //OutOfMemoryError
    }
}

```

ExceptionInInitializerError:-

```

static int a=10/0;
ExceptionInInitializerErrorCaused by: java.lang.ArithmaticException: / by zero

```

```
// NoClassDefFoundError vs ClassNotFoundException :
public class NoClassDefFoundError extends LinkageError
```

Thrown if the Java Virtual Machine or a ClassLoader instance tries to load in the definition of a class (as part of a normal method call or as part of creating a new instance using the new expression) and no definition of the class could be found.

```
class Test
{
    public static void main(String[] args)
    {
        new Demo().m1();
    }
}
```

E:\>java Test

Exception in thread "main" java.lang.NoClassDefFoundError: Demo

public class ClassNotFoundException extends ReflectiveOperationException

Thrown when an application tries to load in a class through its string name using:

The forName method in class Class.

The findSystemClass method in class ClassLoader .

The loadClass method in class ClassLoader.

```
class Test
{
    public static void main(String[] args) throws ClassNotFoundException
    {
        Class.forName("Ratan");
    }
}
```

Exception handling interview Questions

1. What do you mean by Exception?
2. How many types of exceptions in java?
3. What is the difference between Exception and error?
4. What is the difference between checked Exception and un-checked Exception?
5. Is it possible to handle Errors in java?
6. Explain exception handling hierarchy?
7. What the difference is between partially checked and fully checked Exception?
8. What do you mean by exception handling?
9. How many ways are there to handle the exception?
10. What is the root class of Exception handling?
11. Can you please write some of checked and un-checked exceptions in java?
12. What are the keywords present in Exception handling?
13. What is the purpose of try block?
14. In java is it possible to write try without catch or not?
15. What is the purpose catch block?

16. What is the difference between try-catch?
17. Is it possible to write normal code in between try-catch blocks?
18. What are the methods used to print exception messages?
19. What is the purpose of printStackTrace() method?
20. What is the purpose of finally block?
21. If the exception raised in catch block what happened?
22. Independent try blocks are allowed or not allowed?
23. Once the control is out of try , is it remaining statements of try block is executed or not?
24. Try-catch , try-catch-catch , catch-catch , catch-try how many combinations are valid?
25. Try-catch-finally , try-finally ,catch-finally , catch-catch-finally how many combinations are valid?
26. Is possible to write code in between try-catch-finally blocks?
27. Is it possible to write independent catch blocks?
28. Is it possible to write independent finally block?
29. What is the difference between try-catch –finally?
30. Is it allows to use nested try-catch in java?
31. For the method argument it us possible to provide exceptions?
32. What do you mean by exception propagation?
33. is the checked Exceptions are propagated or not?
34. If the exception raised in finally block what happened?
35. What are the situations finally block is not executed?
36. What is the relation of finally & return statement.
37. Try,catch,finally three blocks are returning value then which one taken as a final value.
38. What is the purpose of throws keyword?
39. What is the difference between try-catch blocks and throws keyword?
40. What do you mean by default exception handler and what is the purpose of default exception handler?
41. What is the purpose of throw keyword?
42. If we are writing the code after throw keyword usage then what happened?
43. What is the difference between throw and throws keyword?
44. How to create user defined checked exceptions?
45. How to create user defined un-checked exceptions?
46. Where we placed clean-up code like resource release, database closeting inside the try or catch or finally and why ?
47. Write the code of ArithmeticException?
48. Write the code of NullPointerException?
49. Write the code of ArrayIndexOutOfBoundsException & StringIndexOutOfBoundsException?
50. Write the code of IllegalThreadStateException?
51. When we will get InputMismatchException?

52. When we will get IllegalArgumentException?
53. When we will get ClassCastException?
54. When we will get OutOfMemoryError?
55. What is the difference between ClassNotFoundException & NoClassDefFoundError?
56. When we will get compilation error like “unreportedException must be catch”?
57. When we will get compilation error like “Exception XXXException has already been caught”?
58. When we will get compilation error like “try without catch or finally”?
59. How many approaches are there to create user defined unchecked exceptions and un-checked exceptions?
60. How to create object of user defined exceptions?
61. How to handover user created exception objects to JVM?
62. Is it possible to handle different exceptions by using single catch block yes-->how no→why?
63. What is the purpose of try with resources?
64. Relation with Exception handling & overriding?
65. How to propagated checked & unchecked Exceptions?

***** **Thank you** *****

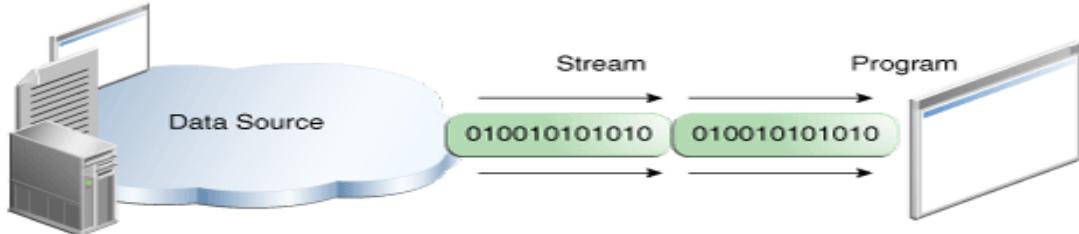
Java .io package

- ✓ Java.io package contains classes to perform input and output operations.
- ✓ By using java.io package we are performing file handling java. It possible to work with only text files but not word & excel files.
- ✓ To work with excel files & word file required extra jar file like **poi.jar** **jxl.jar** files.

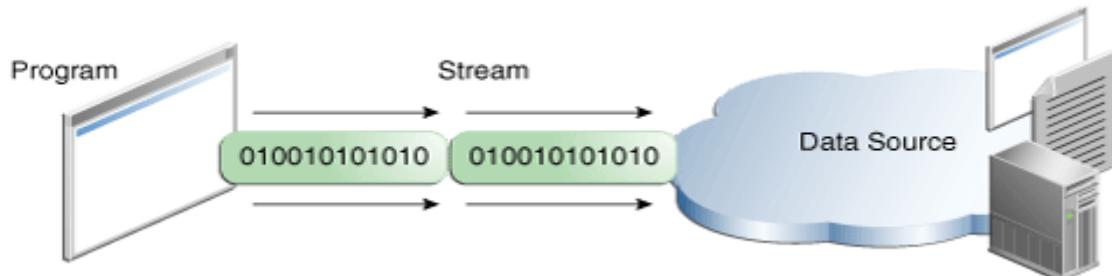
I/O Streams:-

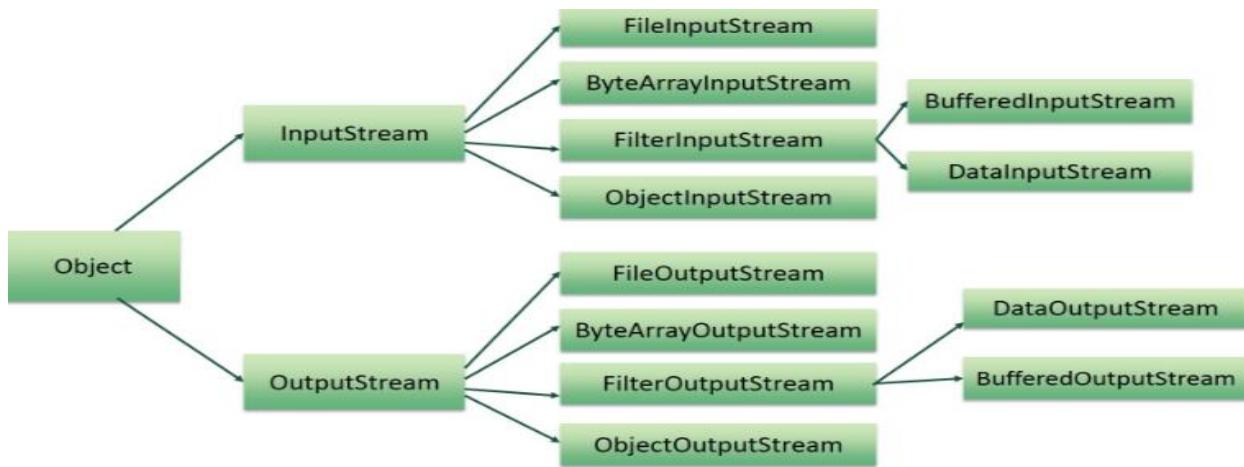
- Byte Streams handle I/O of raw binary data.
- Character Streams handle I/O of character data, automatically handling translation to and from the local character set.
- Buffered Streams optimize input and output by reducing the number of calls to the native API.

Input stream:- Program uses Input stream to read the data from a source one item at a time.



Output stream:- Program uses output stream to write the data to a destination one item at a time.



**Example :- creation of physical File**

```

import java.io.*;
class Test
{
    public static void main(String[] args) throws IOException
    {
        File f = new File("anu.txt");
        boolean b = f.createNewFile();
        if (b)
        {
            System.out.println("File is created successfully");
        }
        else
        {
            System.out.println("File is already existed in location");
        }
    }
}
  
```

Example : creation of directory

```

import java.io.*;
class Test
{
    public static void main(String[] args) throws IOException
    {
        //creation of File
        File f = new File("anu.txt");
        System.out.println(f.exists());
        f.createNewFile();
        System.out.println(f.exists());
    }
}
  
```

//creation of directory

```

File f1 = new File("durga");
System.out.println(f1.exists());
f1.mkdir();
System.out.println(f1.exists());
  
```

//creation of file inside the directory (directory must present)

```

File f2 = new File("durga","durga.txt");
f2.createNewFile();
  
```

```

}
  
```

There are two types of streams:-

1. Byte oriented channel
2. Character oriented channel

Byte streams:-

- ✓ Program uses byte stream to perform input & output of byte data. All byte stream classes developed based on *InputStream* & *OutputStream*.
- ✓ In byte channel the data is transferred in the form of bytes.
- ✓ Generally to transfer the images use byte streams.
- ✓ Program uses byte stream to perform input and output of 8-bit format.

To demonstrate how the byte stream works file I/O provided two main classes

- ✓ *FileInputStream*
 - It is used to read the data from source one item at a time.
 - To read the data from source use *read()* method of *FileInputStream* class.

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

read() method returns first character Unicode value in the form of integer value.
- ✓ *FileOutputStream*
 - It is used to write the data to destination one item at a time.
 - To write the data to destination use *write()* method of *FileOutputStream* class.

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

write() method is taking Unicode value of the character as a parameter.

Character streams:-

- ✓ Program uses character stream to perform input & output of character data. All character stream classes developed based on *Reader* & *Writer* classes.
- ✓ In character channel the data is transferred in the form of characters.
- ✓ Generally to transfer text file data use character channel.
- ✓ Here the data is transferred in 16 bit format.

To work with character channel required two java classes

FileReader

- It is used to read the data from source one item at a time.
- To read the data from source use *read()* method of *FileInputStream* class.

public int read() throws java.io.IOException;
read() method returns first character Unicode value in the form of integer value.

FileWriter

- It is used to write the data to destination one item at a time.
- To write the data to destination use write() method of FileOutputStream class.

public void write(int unicode) throws java.io.IOException;
write() method is taking Unicode value of the character as a parameter.

Steps to design application:-

Step 1:- create the channel.

Step 2:- read the data & store the data in temporary variable.

Step 3:- check the stream is ended or data (data flow completed or not).

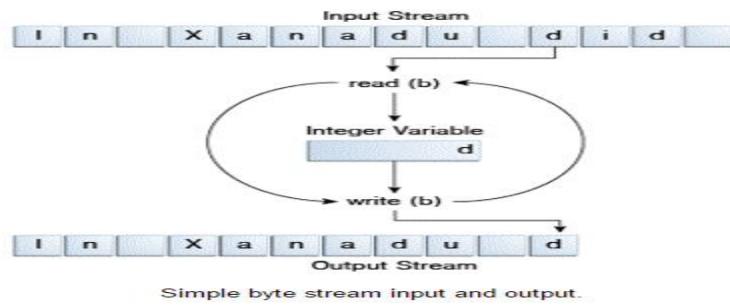
Step 4:- write the data to destination file.

Step 5:- close the streams.

Example :-

```
import java.io.*;
class Test
{
    public static void main(String[] args) throws IOException
    {
        //Byte channel creation
        FileInputStream fis = new FileInputStream("abc.txt");
        FileOutputStream fos = new FileOutputStream("xyz.txt");
        int c;
        while((c=fis.read())!=-1)
        {
            System.out.print((char)c);
            fos.write(c);
        }
        System.out.println("read() & write operations are completed");
        //stream closing operations
        fis.close();
        fos.close();
    }
}
```

- ✓ While working with streams we will get two exceptions mainly FileNotFoundException , IOException & these two exceptions are checked exceptions so must handle these exception by using try-catch blocks or throws keyword.
- ✓ The above example is not recommended because when the exception raised in the middle there may be chance of resources are not released.
- ✓ To overcome above problem use try-catch-finally blocks to release the resources.



Steps to design the application :-

1. Declare the resources.
2. Open the try block do the operations
3. Catch block handle the exception
4. Finally block release the resources.

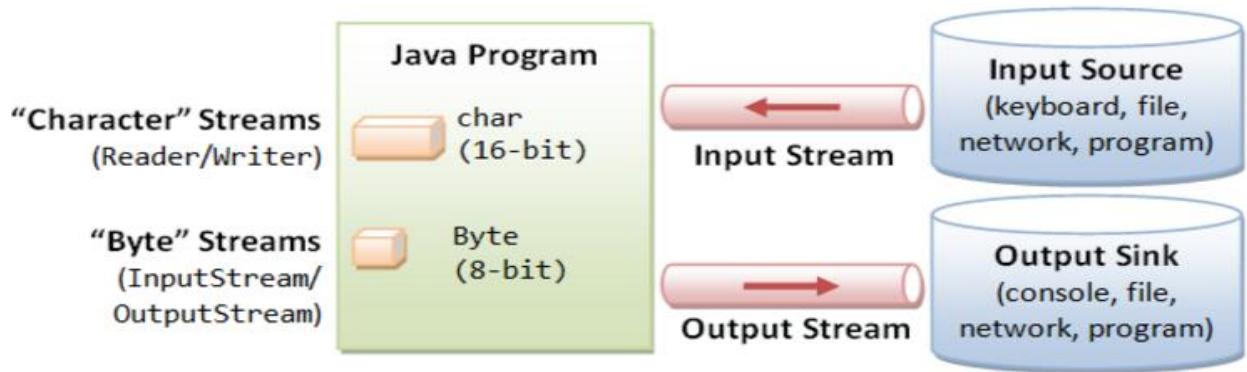
Example :- application with try-catch-finally

```
import java.io.*;
class Test
{
    public static void main(String[] args)
    {
        FileInputStream fis=null;
        FileOutputStream fos=null;
        try{
            fis = new FileInputStream("abc.txt");
            fos = new FileOutputStream("xyz.txt");
            int c;
            while((c=fis.read())!=-1)
            {
                System.out.println((char)c);
                fos.write(c);
            }
        }
        catch(IOException e){ e.printStackTrace(); }
        finally{
            try{ if(fis!=null)fis.close();
                  if(fos!=null)fos.close();
            }
            catch(IOException io){io.printStackTrace();}
        }
    }
}
```

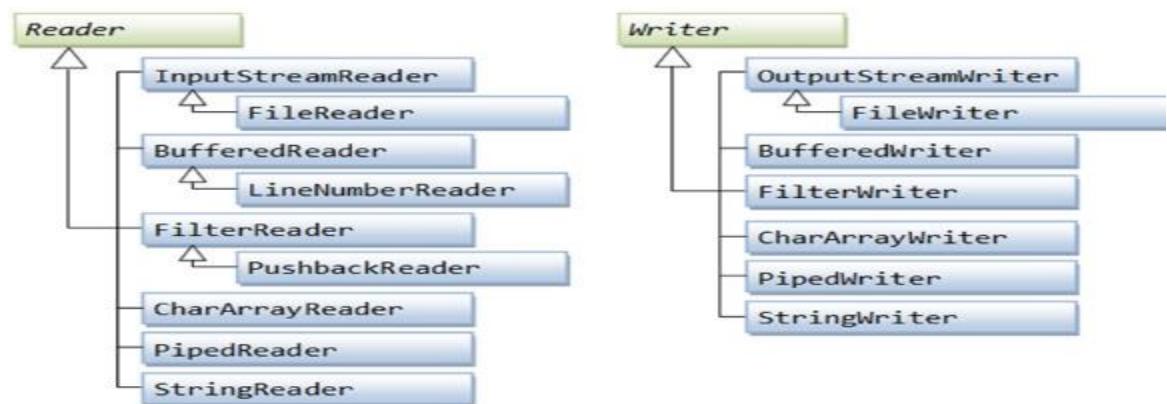
Example : try-with resources

when we declare the resource by using try block once the try block is completed resources are released.

```
import java.io.*;
class Test
{
    public static void main(String[] args)
    {
        try(FileInputStream fis = new FileInputStream("abc.txt");
            FileOutputStream fos = new FileOutputStream("xyz.txt"))
        {
            int c;
            while((c=fis.read())!=-1)
            {
                System.out.println((char)c);
                fos.write(c);
            }
        }
        catch(IOException e){ e.printStackTrace();}
    }
}
```



Note : In CopyCharacters, the int variable holds a character value in its last 16 bits; in CopyBytes, the int variable holds a byte value in its last 8 bits.



Example :

```
import java.io.*;
class Test
{
    public static void main(String[] args)
    {
        FileReader fr=null;
```

```

    FileWriter fw=null;
    try
    {
        fr=new FileReader("abc.txt");
        fw=new FileWriter("ratan.txt");
        int c;
        while ((c=fr.read())!=-1)
        {
            fw.write(c);
        }
    }
    catch (IOException ie){ ie.printStackTrace(); }
    finally
    {
        try{ if(fr!=null) fr.close(); if(fw!=null) fw.close(); }
        catch(IOException e) { e.printStackTrace(); }
    }
}
}

```

CharArrayWriter:-It is used to write the data to multiple files & this implements Appendable interface.

```

import java.io.*;
class Test
{
    public static void main(String[] args) throws IOException
    {
        CharArrayWriter ch = new CharArrayWriter();
        FileReader fr = new FileReader("abc.txt");
        int a;
        while((a=fr.read())!=-1)
        {
            ch.write(a);
        }
        FileWriter fw1 = new FileWriter("a.txt");
        FileWriter fw2 = new FileWriter("b.txt");
        ch.writeTo(fw1);
        ch.writeTo(fw2);
        fw1.close();
        fw2.close();
        fr.close();
        System.out.println("operations are completed");
    }
}

```

Normal streams vs. Buffered Streams:-

- ✓ **Normal streams**
`FileInputStream`
`FileOutputStream`
`FileReader`
`FileWriter`
- ✓ **Normal stream object creation**
`new FileInputStream("abc.txt");`
`new FileOutputStream("abc.txt");`
`new FileReader("abc.txt");`
`new FileWriter("abc.txt");`
- ✓ *In previous examples we are using un-buffered I/O (normal stream) .This means each read and write request is handled directly by the underlying OS.*
- ✓ *By using normal streams it is possible to read the data character by character ,In normal streams each request directly triggers disk access it is relatively expensive & performance is degraded.*
- ✓ *By reading the data character by character performance decreased.*

To overcome above limitations use buffered streams.

- ✓ **Buffered Streams**
`BufferedInputStream`
`BufferedOutputStream`
`BufferedReader`
`BufferedWriter`
- ✓ **Buffered Stream Object creation**
`new BufferedInputStream(new FileInputStream("abc.txt"));`
`new BufferedOutputStream(new FileOutputStream("abc.txt"));`

```
new BufferedReader(new FileReader("abc.txt"));
new BufferedWriter(new FileWriter("abc.txt"));
```

- ✓ The buffered streams are developed based on normal streams.
- ✓ Buffered input stream read the data from buffered memory and it interacting with hard disk only when buffered memory is empty.
- ✓ By using buffered streams it is possible to read the data line by line format.
- ✓ When we use buffered streams, performance increased.

Example :- To work with image data.

```
import java.io.*;
class Test
{
    public static void main(String[] args)
    {
        BufferedInputStream bis=null;
        BufferedOutputStream bos=null;
        try{
            bis=new BufferedInputStream(new FileInputStream("desert.jpg"));
            bos=new BufferedOutputStream(new FileOutputStream("ratan.jpg"));
            int str;
            while ((str=bis.read())!=-1)
            {
                bos.write(str);
            }
        }
        catch(IOException e)
        {
            System.out.println(e);
            System.out.println("getting Exception");
        }
        finally
        {
            try{ if (bis != null)  bis.close();
                  if (bos != null)  bos.close();
            }
            catch(IOException e) {e.printStackTrace();}
        }
    }
}
```

Example:- Buffered Character streams, to work with character data.

```
import java.io.*;
class Test
{
    public static void main(String[] args)
    {
        try(BufferedReader br=new BufferedReader(new FileReader("abc.txt")));

```

```
        BufferedWriter bw=new BufferedWriter(new FileWriter("ratan.txt"));
    {
        String str;
        while ((str=br.readLine())!=null)
        {
            System.out.println(str);
            bw.write(str);
        }
    }
    catch(Exception e)
    {
        System.out.println("getting Exception");
    }
}
```

File IO

1. What is the purpose of java.io package?
 2. What do you mean by stream?
 3. What do you mean by channel and how many types of channels present in java?
 4. What is the difference between normal stream & buffered Streams?
 5. What is the difference between FileInputStream & BufferedReader?
 6. What is the difference between FileOutputStream & printwriter?
 7. Println() method present in which class?
 8. Out is which type of variable(instance /static) present in which class?
 9. To create byte oriented channel we required two class what are those classes?
 10. To create character oriented channel we required two class what are those classes?
 11. What is the difference between byte oriented channel and character oriented channel?
 12. What is the difference between read() & readLine() method?
 13. What is the difference between normal Streams & bufferd streams?
 14. Wat is the purpose of write() & println() ?
 15. Example classes normal Streams & bufferd streams?
 16. What do you mean by serialization?
 17. What is the purpose of Serializable interface& it is marker interface or not ?
 18. How to prevent serialization concept?
 19. What do you mean deserialization?
 20. To perform deserialization we required two classes what are those classes?
 21. To perform serialization we required two classes what are those classes?
 22. What is the purpose of transient modifier?
 23. What are advantage of serialization?
 24. Serializable interface present in which package?
 25. When we will get IOException how many ways are there to handle the exceptions?
 26. IOException is checked Exception or unchecked Exception?

***** *Thank you* *****

Multi Threading

Uni Programming:

- ✓ *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.*
- ✓ *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*

Multiprogramming:

- ✓ *Multiprogramming means executing the more than one program at a time.*
- ✓ *All these programs are controlled by the CPU scheduler.*
- ✓ *CPU scheduler will allocate a particular time period for each and every program.*
- ✓ *The main advantage of multithreading is to provide simultaneous execution of two or more parts of a application to improve the CPU utilization.*

Thread:

- ✓ *Thread is a light weight task or small task of the application.*
- ✓ *Threads exist inside a process. Multiple threads can exist in a single process.*
- ✓ *Executing more than one thread simultaneously is called multithreading.*
- ✓ *The independent execution technical name is called thread. Thread is nothing but separate path of sequential execution.*
- ✓ *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.*

"Multithreading in java is a process of executing multiple threads simultaneously."

"Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread."

There are two types of threads in java:

- ✓ User defined thread
 - Threads are declared by user.
- ✓ daemon thread
 - Daemon threads are executing in background Garbage Collector, Thread Scheduler..etc).

Multitasking:

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- ✓ Process-based Multitasking (Multiprocessing)
- ✓ Thread-based Multitasking (Multithreading)

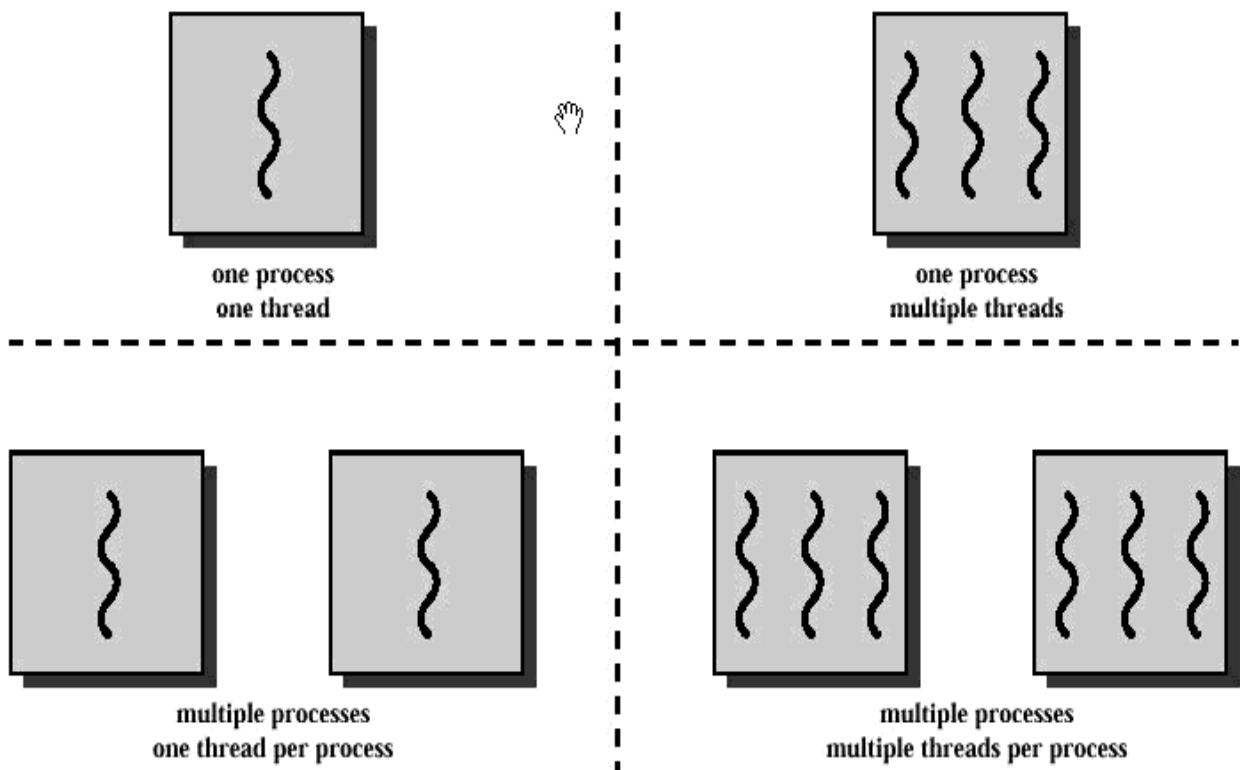
Process-based Multitasking (Multiprocessing):

- ✓ Each process has its own address in memory i.e. each process allocates separate memory area.
- ✓ Process is heavyweight.
- ✓ Cost of communication between the processes is high.(Interprocess communication)
- ✓ Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

Thread-based Multitasking (Multithreading)

- ✓ Threads share the same address space.
- ✓ Thread is lightweight.
- ✓ Cost of communication between the thread is low.
- ✓ At least one process is required for each thread.

Process vs. Thread:



There are two types of programming models

1. Single threaded model.
2. Multi threaded model.

Single threaded model:

- ✓ In java whenever execution starts from main thread, one thread started automatically is called main thread.
- ✓ The below example contains only one thread is called main thread.

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

Multi threaded model.

In this model create the multiple user defined threads, run those threads simultaneously is called multithreaded model.

There are two ways to create a thread in java,

- 1) By extending Thread class.
- 2) By implementing Runnable interface.

*"Thread class & Runnable interface present in **java.lang** package "*

First approach to create thread extending Thread class:

Step 1: Our normal java class will become Thread class when we extend predefined Thread class.

```
class MyThread extends Thread
{
}
```

Step 2: Override the run() method to write the business logic of the Thread. The run method present in Thread class with empty implementation.

```
class MyThread extends Thread
{
    public void run()
    {
        //logics here
    }
}
```

Step 3: Create user defined Thread class object.

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

Step 4: Start the Thread by using start() method of Thread class.

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

ex-1:

```
class MyThread extends Thread
{
    //logics of user thread
    public void run()
    {
        for (int i=0;i<10;i++)
        {
            System.out.println("userdefined Thread");
        }
    }
}
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();

        //business logic of main Thread
        for (int i=0;i<10;i++)
        {
            System.out.println("Main Thread");
        }
    }
}
```

First application Flow of execution:-

Whenever we are calling `t.start()` method then JVM will search `start()` method in the `MyThread` class since not available so JVM will execute parent class(`Thread`) `start()` method.

Thread class start() method responsibilities

- 1) It will register User defined thread is into Thread Scheduler then decide new Thread is created.
- 2) After registering thread `start()` automatically calls `run()` to execute logics of user Thread.

Thread Scheduler:

- ✓ If the application contains more than one thread then thread execution decided by thread scheduler.
- ✓ Thread scheduler is a part of the JVM. It decides thread execution.
- ✓ Thread scheduler is a mental patient we are unable to predict exact behavior of Thread Scheduler it is JVM vendor dependent.
- ✓ Thread Scheduler follows two algorithms to decide Thread execution.
 - 1) Preemptive algorithm.
 - 2) Time slicing algorithm.
- ✓ We can't expect exact behavior of the thread scheduler it is JVM vendor dependent. So we can't say expect output of the multithreaded examples we can say the possible outputs.

Preemptive scheduling:

In this highest priority task is executed first after this task enters into waiting state or dead state then only another higher priority task come to existence.

Time Slicing Scheduling:

A task is executed predefined slice of time and then return pool of ready tasks. The scheduler determines which task is executed based on the priority and other factors.

The main important application areas of the multithreading:

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

Importance of main Thread:

- ✓ It is created automatically when program start.
- ✓ Because this thread effects the other 'child' threads.
- ✓ Because it performs various shutdown actions.

Life cycle stages are:

- 1) New / born
- 2) Ready
- 3) Running state
- 4) Blocked / waiting / non-running mode
- 5) Dead state

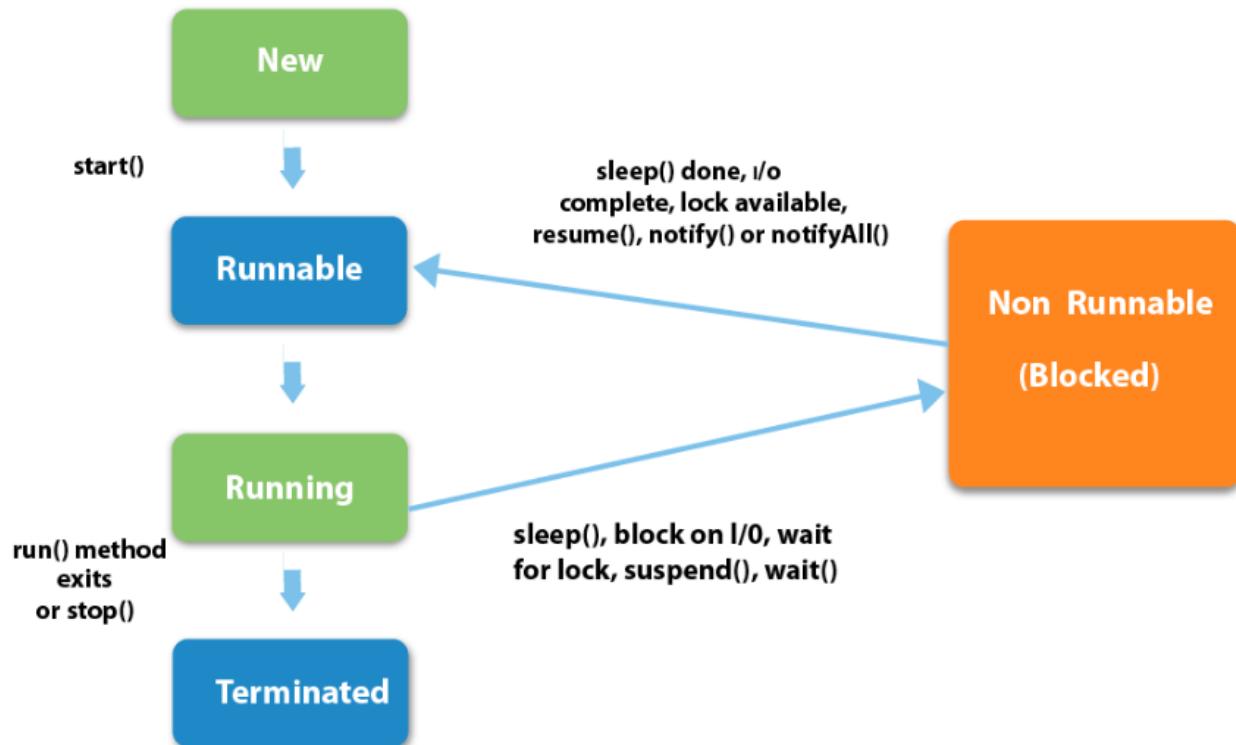
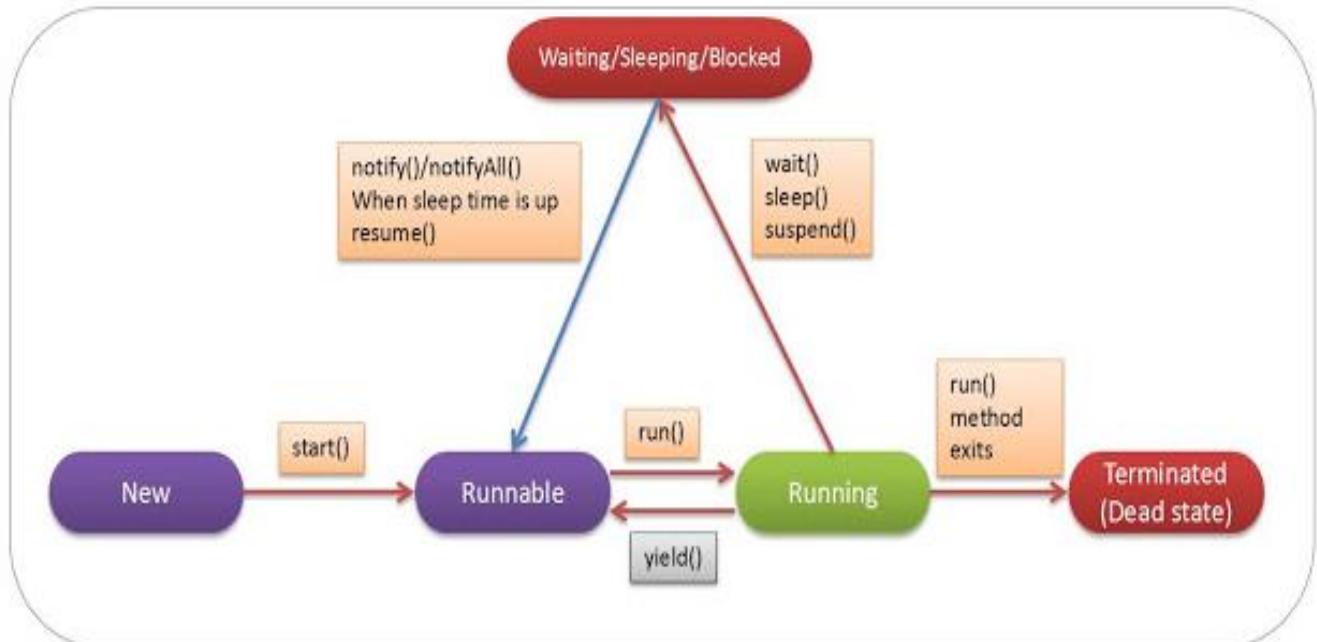
New: just thread is born : `MyThread t=new MyThread();`

Ready: ready to execute : `t.start()`

Running state: If thread scheduler allocates CPU for particular thread then thread goes to running state. The Thread is running state means the run() is executing.

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.



ex-2 : Difference between t.start() and t.run():-

- ✓ In the case of t.start() Thread class start() is executed, it register user defined thread into thread scheduler then decide thread is created.
- ✓ But in the case of t.run() method, no new thread will be created , run() is executed like a normal method call by the main thread.

ex-3 : No need of declare main method in separate class.

```
class MyThread extends Thread
{
    public void run()
    {
        System.out.println("Thread is running..... ");
    }
    public static void main(String[] args)
    {
        MyThread t = new MyThread();
        t.start();
    }
}
```

ex 4: It is not possible to start same thread twice : java.lang.IllegalThreadStateException

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

ex 5: Creating threads using anonymous inner classes approach.

```
class ThreadDemo
{
    public static void main(String[] args)
    {
        //anonymous inner class
        Thread t1 = new Thread()
        {
            public void run()
            {
                System.out.println("user Thread-1");
            }
        };
        t1.start();
    }
}
```

```
//anonymous inner class with nameless object
new Thread()
{
    public void run(){System.out.println("user Thread-1");}
}.start();
}
}
```

ex 6 : Overriding run method recommended process to write the logics

- ✓ It is recommended to override run() method to write the logics of the thread.
- ✓ In below example 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
{
    //not overriding run() 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");
        }
    }
}
```

ex 7 :

- ✓ It is not recommended to override start method, If we are overriding start() method then JVM is executes override start() method thread is not created.
- ✓ Thread is created only when thread class start method is executed.

```
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");
        }
    }
}
```

ex 8 : It is possible to overload run() but start() method always calling 0-arg run() method.

```

class MyThread extends Thread
{
    public void run()
    {
        System.out.println("run o-arg method");
    }
    public void run(int a)
    {
        System.out.println("run 1-arg method");
    }
}

```

ex-9 : possible to write the logics in different method then just call those methods in run() method.

```

class MyThread extends Thread
{
    public void run()
    {
        m1();
        m2();
        m3();
    }
    void m1(){System.out.println("m1 method");}
    void m2(){System.out.println("m2 method");}
    void m3(){System.out.println("m3 method");}
}
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
    }
}

```

ex-10 : Sleep() is overloaded method

sleep() method is a static method used to stop the thread particular amount of time. Once the time expires, thread goes to ready state.

```

public static native void sleep(long) throws java.lang.InterruptedException;
public static void sleep(long, int) throws java.lang.InterruptedException;

```

sleep() method throws **InterruptedException** & it is a checked exception so must handle the exception by using try-catch or throws keyword.

```

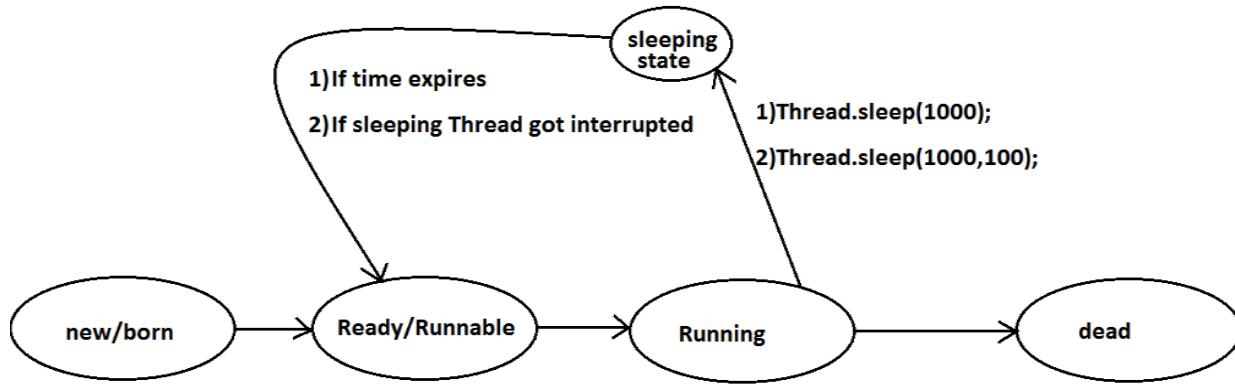
class MyThread extends Thread
{
    public void run()
    {
        for (int i=0;i<10;i++)
        {
            System.out.println("Thread is running..... "+i);
            try {
                Thread.sleep(1000);
            }
            catch(InterruptedException ie)
            {
                ie.printStackTrace();
            }
        }
    }
}

```

```

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

```

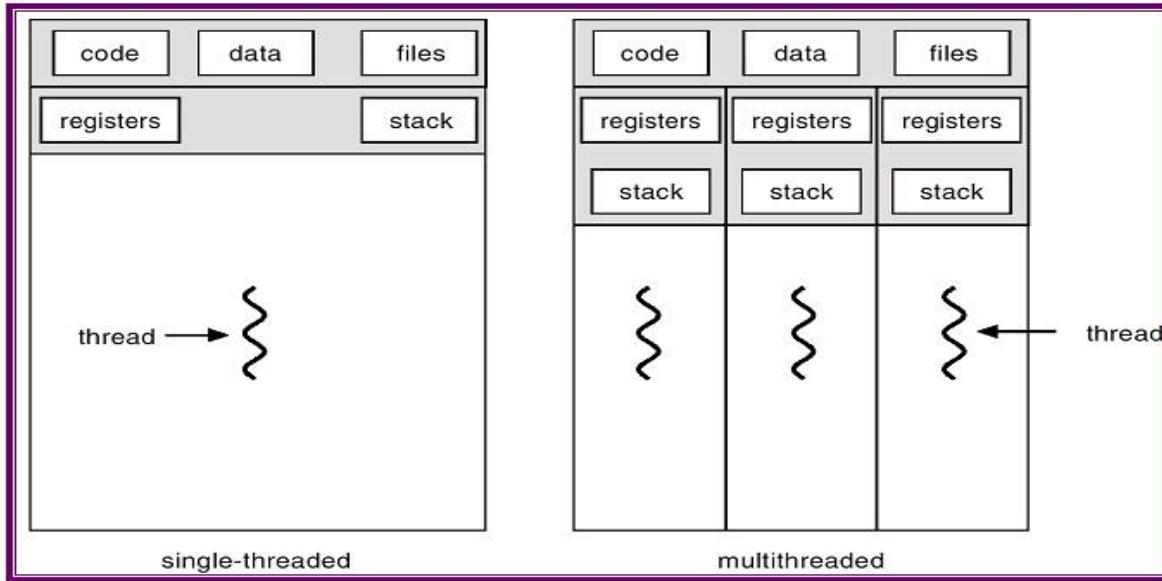
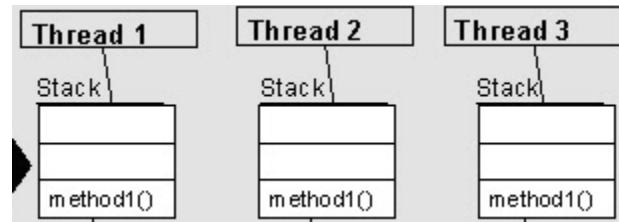
**ex-11 :**

- ✓ Here all threads are performing same tasks.
- ✓ JVM will create separate stack memory for each and every thread.
- ✓ The below application contains four thread hence JVM will create four stack memories.

```

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();
        MyThread t2=new MyThread();
        MyThread t3=new MyThread();
        t1.start();
        t2.start();
        t3.start();
    }
}

```

**ex-12 :**

- ✓ Here different threads are performing different tasks.
- ✓ JVM will create separate stack memory for each and every thread.
- ✓ The below application contains four thread hence JVM will create four stack memories.

```

class MyThread1 extends Thread
{
    public void run()
    {
        System.out.println("ratan task");
    }
}
class MyThread2 extends Thread
{
    public void run()
    {
        System.out.println("Sravya task");
    }
}
class MyThread3 extends Thread
{
    public void run()
    {
        System.out.println("anu task");
    }
}
class ThreadDemo

```

```

{
    public static void main(String[] args) //1- main Thread
    {
        MyThread1 t1 = new MyThread1();
        MyThread2 t2 = new MyThread2();
        MyThread3 t3 = new MyThread3();
        t1.start();
        t2.start();
        t3.start();

        //Reducing length of the code
        new MyThread1().start();
        new MyThread2().start();
        new MyThread3().start();
    }
}

```

ex-13: There are two types of methods in java

- a. Synchronized methods
- b. Non-synchronized methods

Only one thread is allows to access synchronized methods, these methods are thread safe methods but performance will be decreased.

More than one thread is allows to access non synchronized methods, these methods are not a thread safe methods but performance will be increased.

class A

```

{
    public static synchronized void status(String msg)
    {
        for (int i=0;i<3;i++)
        {
            System.out.println("hi="+msg);
            try{Thread.sleep(1000);}catch(InterruptedException ie){ie.printStackTrace();}
        }
    }
}

class MyThread1 extends Thread
{
    public void run(){
        A.status("ratan");
    }
}

class MyThread2 extends Thread

```

```

{
    public void run()
    {
        A.status("durga");
    }
}
class MyThread3 extends Thread
{
    public void run()
    {
        A.status("any");
    }
}
class ThreadDemo
{
    public static void main(String[] args)
    {
        new MyThread1().start();
        new MyThread2().start();
        new MyThread3().start();
    }
}

```

method is non-synchronized:

G:\>java ThreadDemo

hi=durga
hi=ratan
hi=any
hi=durga
hi=ratan
hi=any
hi=ratan
hi=durga
hi=any

ex-14 :- Thread name& id & isAlive

- ✓ Every Thread in java having name, default name of the main thread is main & default name of user threads starts from **Thread-0**.

t1 -->Thread-0
t2 -->Thread-1
t3 -->Thread-2

- ✓ To set the name use **setName()** & to get the name use **getName()**,

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

- ✓ To represent the current thread use **currentThread()** method of thread class.
public static native java.lang.Thread currentThread();

- ✓ To get id of a thread use **getId()** method.
public long getId();

- ✓ To check the particular thread is running or not use **isAlive()** method.
public final native boolean isAlive();

class MyThread extends Thread

method is synchronized:-

G:\>java ThreadDemo

hi=ratan
hi=ratan
hi=any
hi=any
hi=any
hi=durga
hi=durga
hi=durga

```

{
}
class ThreadDemo
{
    public static void main(String args[])
    {
        MyThread t1=new MyThread();
        MyThread t2=new MyThread();

        System.out.println("t1 Thread name="+t1.getName());
        System.out.println("t2 Thread name="+t2.getName());
        System.out.println(Thread.currentThread().getName());

        t1.setName("ratan");
        t2.setName("anu");
        Thread.currentThread().setName("durga");

        System.out.println("t1 Thread name="+t1.getName());
        System.out.println("t2 Thread name="+t2.getName());
        System.out.println(Thread.currentThread().getName());

        System.out.println("t1 Thread id="+t1.getId());
        System.out.println("t2 Thread id="+t2.getId());
        System.out.println(Thread.currentThread().getId());

        System.out.println("t1 Thread alive or not="+t1.isAlive());
    }
}

```

ex-15: Thread Priorities

- ✓ In java every Thread has some property. It may be default priority provided by the JVM or customized priority provided by the programmer.
- ✓ Based on priority the thread scheduler allocates the CPU.
- ✓ The valid range of thread priorities is 1 – 10. Where 1 is lowest priority and 10 is highest priority.
- ✓ The default priority of main thread is 5 **NORM_PRIORITY**. The priority of child thread is inherited from the parent.
- ✓ To represent the priority thread class contains three constants,


```
MIN_PRIORITY = 1
NORM_PRIORITY = 5
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)
```
- ✓ Thread priority decide when to switch from one running thread to another this process is called context switching.
- ✓ **Must set the priority before starting the thread.**
- ✓ If the more than one thread is having same priority then thread execution decide by thread scheduler.

```

class MyThread extends Thread
{
    public void run()
    {
        System.out.println("current Thread name = "+Thread.currentThread().getName());
        System.out.println("current Thread priority = "+Thread.currentThread().getPriority());
    }
};

class ThreadDemo
{
    public static void main(String[] args)//main thread started
    {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        t1.setPriority(Thread.MIN_PRIORITY);
        t2.setPriority(Thread.MAX_PRIORITY);
        t1.start();
        t2.start();
    }
};

```

Case : Priority range 1-10 if we set more than 10 JVM will generate `IllegalArgumentException`.

```

MyThread t1 = new MyThread();
t1.setPriority(15);      // java.lang.IllegalArgumentException

```

ex-16 : `yield()` method

- ✓ *Yield() method causes to pause current executing Thread for giving the chance for waiting threads of same priority.*
- ✓ *When we called yield() method the thread is not enter into waiting state, it will enter into ready state.*
- ✓ *If there are no waiting threads or all threads are having low priority then the same thread will continue its execution once again.*

```

class MyThread extends Thread
{
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            Thread.yield();
            System.out.println("child thread");
        }
    }
};

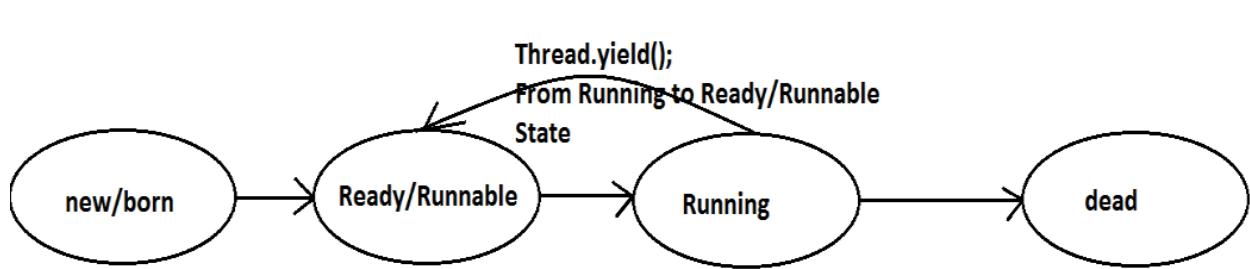
class ThreadYieldDemo
{
    public static void main(String[] args)

```

```

    {
        MyThread t1=new MyThread();
        t1.start();
        for(int i=0;i<10;i++)
        {
            System.out.println("main thread");
        }
    }
}

```



ex-17 : Thread.join(-,-) overloaded method

- ✓ To stop the current executing thread until completion of target thread use join() method.
- ✓ Join method allows one thread to wait for the completion of another thread.
- ✓ Join() method throws InterruptedException it is a checked exception hence handle the checked exception by using try-catch blocks or throws keyword.

```

public final void join() throws java.lang.InterruptedException;
public final synchronized void join(long) throws java.lang.InterruptedException;
public final synchronized void join(long, int) throws java.lang.InterruptedException;

```

join() method throws InterruptedException & it is a checked exception so must handle the exception by using try-catch or throws.

```

class MyThread extends Thread
{
    public void run()
    {
        for (int i=0;i<5;i++)
        {
            System.out.println("user thread");
            try{ Thread.sleep(2000); } catch(InterruptedException e){ e.printStackTrace(); }
        }
    }
}

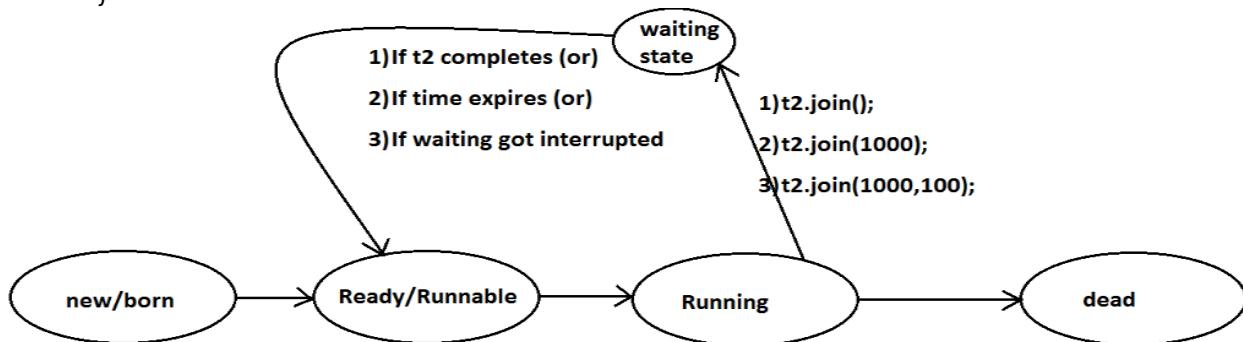
```

```

        }
    }
}

class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t1=new MyThread();
        t1.start();
        try{ t1.join(); }
        catch (InterruptedException ie) {ie.printStackTrace();}
        //logics of main thread
        for (int i=0;i<5;i++)
        {
            System.out.println("main thread");
            try{ Thread.sleep(2000); }
            catch(InterruptedException e){e.printStackTrace();}
        }
    }
}

```



ex 18: *Java.lang.Thread.Interrupted()*

- ✓ A thread can interrupt another sleeping or waiting thread.
- ✓ The interrupt() is effected whenever our thread enters into waiting state or sleeping state and if our thread doesn't enter into the waiting/sleeping state interrupted call will be wasted.

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 interrupted 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();
    }
}

```

ex-19: Hook Thread

- Shutdown hook used to perform cleanup activities when JVM shutdown normally or abnormally.
- Clean-up activities like
 - Resource release
 - Database closing
 - Sending alert message
- So if you want to execute some code before JVM shutdown use shutdown hook

The JVM will be shutdown in following cases.

- c. When you typed `ctrl+C`
- d. When we used `System.exit(int)`
- e. When the system is shutdownetc

To get the Runtime class object use static factory method `getRuntime()`

```
Runtime r = Runtime.getRuntime();
```

To add the shutdown hook to JVM use `addShutdownHook(obj)` method of Runtime Class.

```
public void addShutdownHook(java.lang.Thread);
```

To remove the shutdown hook from JVM use `removeShutdownHook(obj)` method of Runtime Class.

```
public boolean removeShutdownHook(java.lang.Thread);
```

Factory method: One java class method is able to return same class object or different class object is called factory method.

```

class MyThread extends Thread
{
    public void run()
    {
        System.out.println("shutdown hook");
    }
}
class ThreadDemo
{
    public static void main(String[] args) throws InterruptedException
    {
        MyThread t = new MyThread();
        //creating Runtime class Object by using factory method
        Runtime r = Runtime.getRuntime();
        r.addShutdownHook(t); //adding Thread to JVM hook
        for (int i=0;i<10;i++)
        {
            System.out.println("main thread is running");
            Thread.sleep(3000);
        }
    }
}
D:\DP>java ThreadDemo
main thread is running
main thread is running
main thread is running
main thread is running
shutdown hook
while running Main thread press Ctrl+C then hook thread will be executed.

```

ex-20: Daemon threads.

- ✓ The threads which are executed at background to give the support to foreground thread is called daemon threads.
ex : garbage collector, ThreadScheduler.default exceptional handler....etc
- ✓ Generally in java we have two types of threads
 1. User thread.
 2. Daemon thread.
- ✓ Generally all threads are created by user are called user threads if want make the daemon thread use the following method.

<code>public final void setDaemon(boolean);</code> <code>public final boolean isDaemon();</code>	used to specify the daemon thread. to check the thread is daemon or not.
-----------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------

Properties of daemon thread:

- ❖ These threads are executing background to give support to fore ground threads.
- ❖ These threads are low priority threads.
- ❖ Whenever user threads are completes its execution all daemon threads are automatically stopped.

```
class MyThread extends Thread
```

```

    {
        public void run()
        {
            for (int i=0;i<10 ;i++)
            {
                System.out.println("Daemon Thread.....");
                try{Thread.sleep(1000);}
                catch(InterruptedException ie) {ie.printStackTrace();}
            }
        }
    }
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t = new MyThread();
        t.setDaemon(true); //setting daemon nature to Thread
        t.start();

        //main thread logic
        for (int i=0;i<5 ;i++)
        {
            System.out.println("main Thread.....");
            try{Thread.sleep(1000);}
            catch(InterruptedException ie){ ie.printStackTrace();      }
        }
    }
}

```

Second approach to create thread by implementing Runnable interface:

Step 1: Our normal java class will become Thread class when we are implementing Runnable interface.

```

class MyRunnable implements Runnable
{
}

```

Step2: Override run method to write logic of Thread.

```

class MyClass extends Runnable
{
    public void run()
    {
        //logics here
    }
}

```

Step 3: Creating a object.

```
MyClass obj=new MyClass();
```

Step 4: start the thread by using Thread class start method of thread class.

ex -21 : creation of Thread implementing Runnable interface :-

```

class MyThread implements Runnable
{
    public void run()
    {
        for (int i=0;i<10;i++)
        {
            System.out.println("userdefined Thread");
        }
    }
}
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread r=new MyThread();
        Thread t=new Thread(r);
        t.start();
    }
}

//business logic of main Thread
for (int i=0;i<10;i++)
{
    System.out.println("Main Thread");
}
}

```

There are two approaches to create a thread but the recommended approach is implementing Runnable interface.

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:-

- ✓ *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.*
- ✓ *By implementing the Runnable interface, the class can still extend other base classes if necessary.*

Example-8:-

```

class MyThread implements Runnable
{
    public void run()
    {
        System.out.println("Thread is running..... ");
    }
}
public static void main(String[] args)
{
    MyThread r = new MyThread();
    Thread t = new Thread(r);
    t.start();
}
}

```

Internal Implementation of multiThreading:-

```

interface Runnable
{
    public abstract void run();
}
class Thread implements Runnable

```

```

    {
        public void run()
        {
            //empty implementation
        }
    };
    class MyThread extends Thread
    {
        public void run()      //overriding run() to write business logic
        {
            //write the logics here
        }
    };
}

```

Example-5 Creating two threads by implementing Runnable interface using anonymous inner classes

```

class ThreadDemo
{
    public static void main(String[] args)
    {
        //application with anonymous inner classes
        Runnable r1 = new Runnable()
        {
            public void run()
            {
                System.out.println("user Thread-1");
            }
        };
        Thread t1 = new Thread(r1);
        t1.start();

        //application with anonymous inner classes
        new Thread(new Runnable()
        {
            public void run()
            {
                System.out.println("user Thread-1");
            }
        }).start();

        //application with lambda
        new Thread(()->System.out.println("user thread")).start();
    }
}

```

Volatile:-

- ✓ Volatile modifier is also applicable only for variables.
- ✓ 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.

synchronized blocks:-

- ✓ Synchronized vlock can be used to perform synchronization on any specific resource of the code.
- ✓ if the application method contains 100 lines but if we want to synchronized only 10 lines of code use synchronized blocks.
- ✓ The synchronized block contains less scope compare to method.

Syntax:-

```
synchronized(object)
{
    //code
}
```

Example 25 :

```
class Heroin
{
    public void message(String msg)
    {
        synchronized(this){
            System.out.println("hi "+msg+" "+Thread.currentThread().getName());
            try{Thread.sleep(5000);}
            catch(InterruptedException e){e.printStackTrace();}
        }
        System.out.println("hi Sravyasoft");
    }
};

class MyThread1 extends Thread
{
    Heroin h;
    MyThread1(Heroin h)
    {
        this.h=h;
    }
    public void run()
    {
        h.message("Anushka");
    }
};

class MyThread2 extends Thread
{
    Heroin h;
    MyThread2(Heroin h)
```

```

    {
        this.h=h;
    }
    public void run()
    {
        h.message("Ratan");
    }
};

class ThreadDemo
{
    public static void main(String[] args)
    {
        Heroin h = new Heroin();
        MyThread1 t1 = new MyThread1(h);
        MyThread2 t2 = new MyThread2(h);
        t1.start();
        t2.start();
    }
};

```

Inter thread communication :

- ✓ The process of communicating more than one thread each other is called inter thread communication.
 - ✓ The threads are communicating each other by using three methods
 - wait()
 - notify()
 - notifyAll()
- the above three methods are present in Object class.

Wait vs Sleep :-**Example-26 :-**

```

class MyThread extends Thread
{
    int total;
    public void run()
    {
        synchronized(this){
            for (int i=0;i<10 ;i++)
            {
                total=total+i;
            }
            notify();
        }
    }
}

class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t = new MyThread();
        t.start();
        synchronized(t)
        {
            System.out.println("MyThrad total is waiting for MyThread completion...");
            try{
                t.wait();
            }catch(InterruptedException ie){System.out.println(ie);}
        }
    }
}

```

```

        }
        System.out.println("MyThrad total is =" + t.total);
    }
}

```

Thread group in java :-

Java provides simple way to group more than one thread in a single object. In this case we can suspend , resume , interrupt group of threads in single method call.

Example 27 :

```

class MyRunnable implements Runnable
{
    public void run(){      System.out.println(Thread.currentThread().getName());      }
}

class ThreadDemo
{
    public static void main(String[] args)
    {
        ThreadGroup tg1 = new ThreadGroup("Ratan");
        new Thread(tg1,new MyRunnable()).start();
        new Thread(tg1,new MyRunnable()).start();

        ThreadGroup tg2 = new ThreadGroup("anu");
        new Thread(tg2,new MyRunnable(),"one").start();
        new Thread(tg2,new MyRunnable(),"two").start();
    }
}

```

Example-28 : Application with lambda

```

class ThreadDemo
{
    public static void main(String[] args)
    {
        ThreadGroup tg1 = new ThreadGroup("Ratan");
        new Thread(tg1,()>System.out.println(Thread.currentThread().getName())).start();
        new Thread(tg1,()>System.out.println(Thread.currentThread().getName())).start();

        ThreadGroup tg2 = new ThreadGroup("anu");
        new Thread(tg2,()>System.out.println(Thread.currentThread().getName()),"one").start();
        new Thread(tg2,()>System.out.println(Thread.currentThread().getName()),"two").start();

        System.out.println(tg1.getName());
        tg1.list();
    }
}

```

}

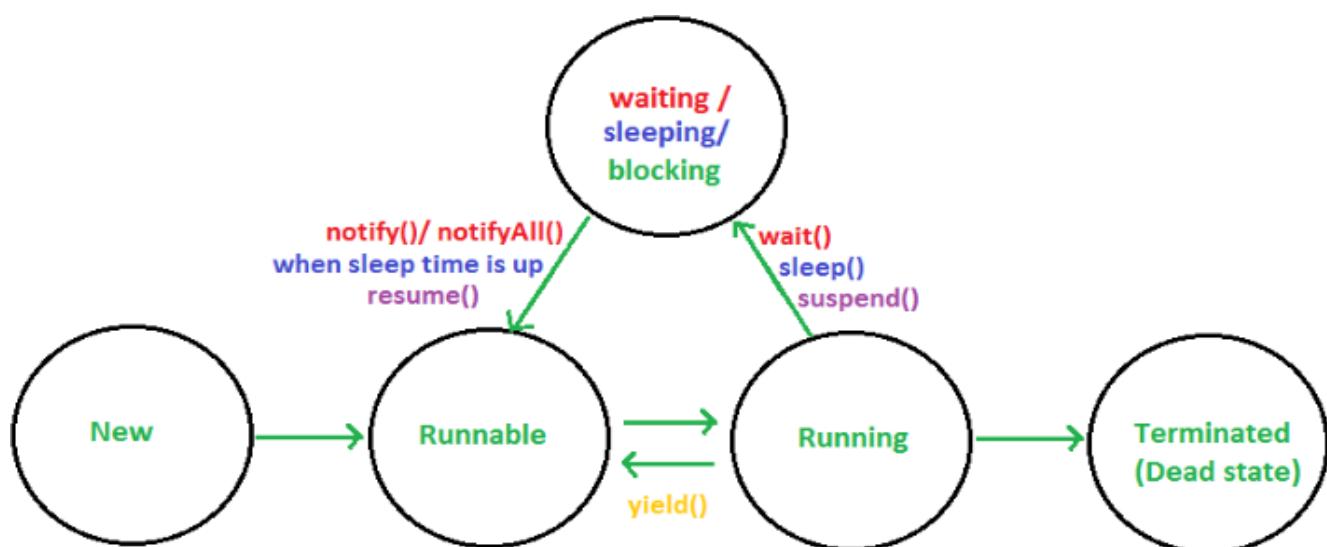
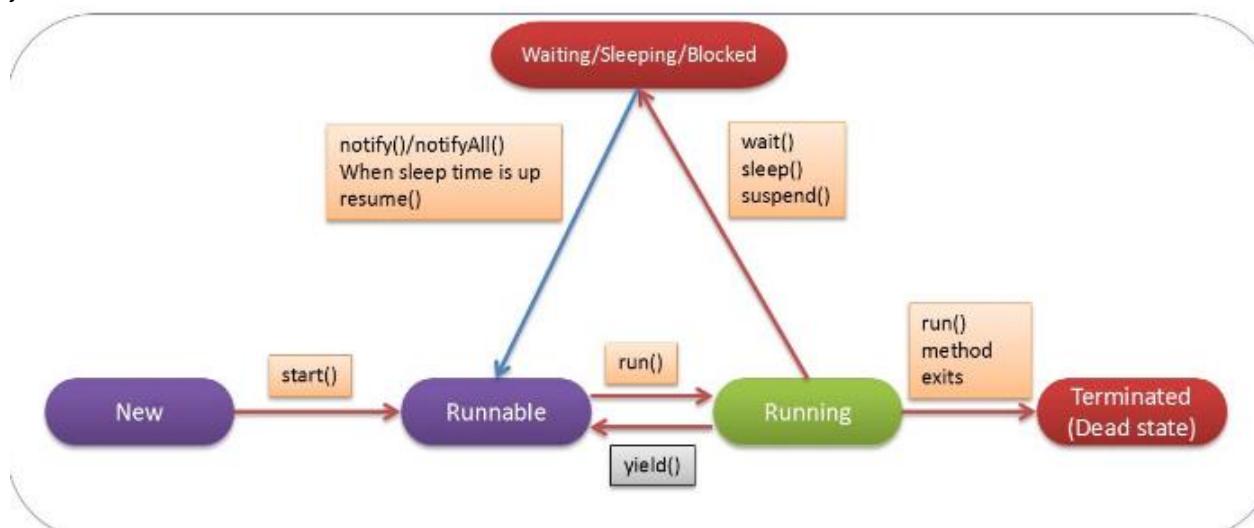
Example-29 : ThreadGroup interruption

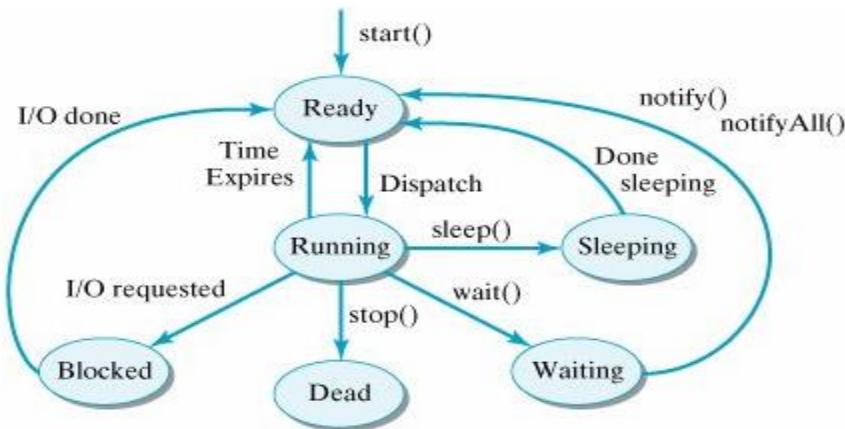
```

class MyRunnable implements Runnable
{
    public void run()
    {
        System.out.println(Thread.currentThread().getName());
        try{Thread.sleep(1000);}catch(InterruptedException ie){ie.printStackTrace();}
    }
}

class ThreadDemo
{
    public static void main(String[] args)
    {
        ThreadGroup tg1 = new ThreadGroup("Ratan");
        new Thread(tg1,new MyRunnable()).start();
        new Thread(tg1,new MyRunnable()).start();
        tg1.interrupt();
    }
}

```





Multithreading

1. What do you mean by Thread?
2. What do you mean by single threaded model?
3. What is the difference single threaded model and multithreaded model?
4. What do you mean by main thread and what is the importance?
5. What is the difference between process and thread?
6. How many ways are there to create thread which one prefer?
7. Thread class & Runnable interface present in which package?
8. Runnable interface is marker interface or not?
9. What is the difference between `t.start()` & `t.run()` methods where `t` is object of Thread class?
10. How to start the thread?
11. What are the life cycle methods of thread?
12. Run() method present in class/interface ? Is it possible to override run() method or not?
13. Is it possible to override start method or not?
14. What is the purpose of thread scheduler?
15. Thread Scheduler follows which algorithm?
16. What is purpose of thread priority?
17. What is purpose of sleep() & isAlive() & isDaemon() & join() & getId() & activeCount() methods?
18. Jvm creates stack memory one per Thread or all threads only one stack?
19. What is the thread priority range & how to set priority and how to get priority?
20. What is the default name of user defined thread and main thread? And how to set the name and how to get the name?
21. What is the default priority of main thread?
22. Which approach is best approach to create a thread?
23. What is the difference between synchronized method and non-synchronized method?
24. What is the purpose of synchronized modifier?
25. What is the difference between synchronized method and non synchronized method?
26. What do you mean by demon thread tell me some examples?
27. what is the purpose of volatile modifier?
28. What is the difference between synchronized method and synchronized block?
29. Wait(), notify(), notifyAll() methods are present in which class?

30. When we will get Exception like "IllegalThreadStateException" ?
31. When we will get Exception like "IllegalArgumentException" ?
32. If two threads are having same priority then who decides thread execution?
33. How two threads are communicate each other?
34. What is race condition?
35. How to check whether the thread is demon or not? Main thread is demon or not?
36. How a thread can interrupt another thread?
37. Explain about wait() notify() notifyAll()?
38. Once we create thread what is the default priority?
39. What is the max priority & min priority & norm priority?
40. What is the difference between preemptive scheduling vs time slicing?

***** Thank you *****

Nested classes

Declaring the class inside another class is called nested classes. It is introduced in the 1.1 version.

There are two types of nested classes in java,

1. Static nested classes
2. Non static nested classes (these are called inner classes)
 - a. Normal inner classes
 - b. Method local inner classes
 - c. Anonymous inner classes



Inner class's vs nested classes:

Inner classes are part of nested classes. The non-static nested classes are called inner classes.

Important points:

```

class Outer
{
    class Inner
    {
    }
}
  
```

Note 1: The compiler will generate .class files for both inner & outer classes.

Outer class .class file name	:	Outer.class
Inner class .class file name	:	Outer\$Inner.class

Note 2: It is possible to create the objects for both inner & outer classes.

Outer class Object creation	:	Outer o = new Outer();
Inner class object creation	:	Outer.Inner I = new Outer().new Inner();

Note 3: By using outer class object it is possible to call only outer class properties.
By using inner class object it is possible to call only inner class properties.

Note 4: The inner class methods are able to access outer class methods but,
Outer class methods are unable to access inner class methods.

Advantages of nested classes:

- ✓ **It is the way logically grouping classes that are only used in the one place.**
If one class required another class only one time then it is logically embedded it into that classes make the two classes together.
 - it increase the encapsulation
 - it improves readability
 - the inner class is able to access outer class private properties.
- ✓ **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. It improves maintenance of the project.
- ✓ **Code optimization :**
Anonymous inner classes reduce the length of the code.
- ✓ **Nested classes always for one time usage.**
If we are declaring class inside the method once the method is complete class destroyed.

Without existing one type of object there is no chance of existing another type of object we should use Inner classes.

One functionality can exists with presence of another functionality use inner classes.

ex : University contains several departments but without university no chance of existing departments.

```
class University           //outer class
{
    class Department     //inner class
    {
    }
}
```

ex: Map is a collection of key-value pairs & each key-value pair is entry. So map contains group of entry's .The entry can exists with presents of Map only.

```
interface Map
{
    interface Entry
    {
    }
}
```

Single line about nested class's advantage

"Nested classes enable us to logically group classes that are only used in one place, write more readable and maintainable code and increase encapsulation."

1. Member inner classes / regular inner classes / normal inner classes / instance inner classes:

Declaring the class inside another class is called normal inner classes.

ex- 1:

```
class Outer
{
    private int a=10,b=20;
    void m1()
    {
        System.out.println("outer class m1()");
    }
    class Inner
    {
        int i=100,j=200;
        void m2()
        {
            System.out.println("inner class m2()");
            System.out.println(a+b);
            System.out.println(i+j);
            m1();
        }
    }
}
class Test
{
    public static void main(String... ratan)
    {
        Outer o = new Outer();
        o.m1();
        Outer.Inner i = o.new Inner();
        i.m2();
    }
}
```

ex-2: Different variable names.

```

class Outer
{
    int a=10,b=20;
    class Inner
    {
        int x=100,y=200;
        void add(int i,int j)
        {
            System.out.println(i+j);
            System.out.println(x+y);
            System.out.println(a+b);
        }
    }
    public static void main(String[] args)
    {
        new Outer().new Inner().add(1000,2000);
    }
}

```

- o It is possible to declare the main method inside the outer class.

ex-3 : Same variable names.

```

class Outer
{
    private int a=10,b=20;
    class Inner
    {
        int a=100,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
        }
    }
    public static void main(String... ratan)
    {
        new Outer().new Inner().m1(1000,2000);
    }
}

```

ex-4 : inside the inner class not possible to declare static members.

```

class Outer
{
    class Inner
    {
        public final static int a=10;
    }
}

```

error: Illegal static declaration in inner class Outer.Inner

- ✓ Inside the inner classes it is not possible to declare static members.
- ✓ Inside the inner classes it is not possible to declare main method because main is static.

ex -5 : Inner classes: constructor , instance blocks , static blocks**Outer classes:** contractors, instance blocks

```

class Outer
{
    Outer()
}

```

```

    {
        System.out.println("outer class cons ");
    }
    {
        System.out.println("outer clas ins blocks");
    }
    static
    {
        System.out.println("outer clas static block");
    }
    class Inner
    {
        Inner()
        {
            System.out.println("inner cons");
        }
        {
            System.out.println("inner ins block");
        }
    }
    public static void main(String[] args)
    {
        new Outer().new Inner();
    }
}

```

2. Method local inner classes:

- ✓ Declaring the class inside the method is called method local inner classes.
- ✓ The scope of method local inner class is only within the method. It means whenever the method is completed inner class object destroyed.
- ✓ The applicable modifiers on method local inner classes are **final & abstract** .

Syntax:

```

class Outer
{
    void m1()
    {
        class Inner
        {
        }
    }
}

```

ex-1:

```

class Outer
{
    private int a=100;
    void m1()
    {
        class Inner           //method local inner class
        {
            void m2()
            {
                System.out.println("inner class method");
                System.out.println(a);
            }
        }
        Inner i=new Inner();
        i.innerMethod();
    }
    public static void main(String[] args)
    {
        Outer o=new Outer();
        o.m1();
    }
}

```

```

        }
    }

ex-2: class Outer
{
    void show()
    {
        for (int i=0;i<10 ;i++ )
        {
            class Inner
            {
                public void info()
                {
                    System.out.println("method local inner class");
                }
            }
            new Inner().info();
        }
    }
    public static void main(String[] args)
    {
        new Outer().show();
    }
}

```

1. Static nested classes:-

- ✓ Declaring the class inside another class with static modifier is called static inner classes.
- ✓ The normal inner class is able to access both static & non-static members of outer class but static inner class is able to access only static members of outer class.

Normal inner classes

- ✓ Allows both static & non-static members of Outer class.

```

class Outer
{
    int a=10;
    static int b=20;
    class Inner
    {System.out.println(a);
     System.out.println(b);
    };
};

✓ Static declarations are not allowed.
✓ Main method , static blocks are not allowed.
✓ Object creation
    New Outer().new Inner();

```

Static nested classes

- ✓ Allows only static members of outer class.

```

class Outer
{
    static int a=10;
    int b=20;
    static class Inner
    { System.out.println(a);
     System.out.println(b);//not possible
    };
};

✓ Static declarations are allowed.
✓ Main method & static blocks are allowed.
✓ Object creation
    New Outer.Inner();

```

Static nested classes:-

Example :-

```

class Outer
{
    static int a=10;
    static int b=20;
    static class Inner
    {
        void m1()
        {
            System.out.println(a);
        }
    }
}

```

```

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

public static void main(String[] args)
{
    Outer.Inner i=new Outer.Inner();
    i.m1();
}

};

Example :-
class Outer
{
    static class Inner
    {
        public static void main(String[] args)
        {
            System.out.println("inner class main");
        }
    }
}

```

G:\>javac Test.java

G:\>java Outer\$Inner

inner class main

4. Anonymous inner class:-

- ✓ The name less inner class in java is called anonymous inner class.
- ✓ it can be used to provide the implementation of normal class or abstract class or interface

Application without anonymous inner classes:

```

class A           //assume predefined class
{
    void m1(){}
    void m2(){}
}

class Test extends A      //user class contains logics
{
    void m1(){System.out.println("m1 method");}
    void m2(){System.out.println("m2 method");}
}

class TestClient      //client code
{
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
        t.m2();
    }
}

```

Application with anonymous inner classes:-

Application with anonymous inner classes:- declaration of class out side of the main

```

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

```

```

    {
        TestClient t = new TestClient();
        t.a.m1();
        t.a.m2();
    }
}

```

App with anonymous inner classes reduce the length of the code : declaration of class out side of main

```

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

```

Example :-

```

class A           //predefined class
{
    void m1(){}
}

class TestClient
{
    A a = new A()
    {
        void m1()
        {
            System.out.println("m1 method");
            System.out.println(a.getClass().getName());
        }
    };
    public static void main(String[] args)
    {
        TestClient t = new TestClient();
        t.a.m1();
    }
}
G:\>java TestClient
m1 method
TestClient$1

```

- ✓ In above example when we create the object of **A** class internally **A** class object is not creating, the compiler will generate one new class that class object will be created & that class is called **anonymous inner class**.
- ✓ To get the compiler generated class use **getClass()** method of **Object** class.

TestClient\$1 (.class file code) open by using java de-compiler software

```

class TestClient$1 extends A
{
    void m1()
    {
        System.out.println("m1 method");
        System.out.println(a.getClass().getName());
    }
}

```

```

    }
final TestClient this$0;
TestClient$1()
{ this$0 = TestClient.this;
super();
}
}

```

The above example different ways :- reducing length of the code

```

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

```

Example :Abstract classes vs. Anonymous inner class.

```

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();

        new Animal()
        {
            void eat(){ System.out.println("animals eating gross"); }
        }.eat();
    }
}

```

Example :-interfaces vs. anonymous inner class.

```

interface It1
{
    void m1();
};

class TestClient
{
    public static void main(String[] args)
    {
        It1 i = new It1()
        {
            public void m1(){System.out.println("m1 method");}
        };
        i.m1();

        new It1()

```

```
{      public void m1(){System.out.println("m1 method");}
}.m1();
}
}
```

Example :

```
interface It1
{
    void m1();
}

class A
{
    void firstMethod(It1 i)
    {
        i.m1();
    }
}

class Test
{
    public static void main(String[] args)
    {
        //approach-1
        new A().firstMethod(new X());

        //approach-2
        new A().firstMethod(new It1(){public void m1()
        {
            System.out.println("implementation with anonymous inner classes");
        }});

        //approach-3
        new A().firstMethod(()->System.out.println("lambda code"));
    }
}

class X implements It1
{
    public void m1()
    {
        System.out.println("implementations in separte class");
    }
}
```

Some possibilities:-**Case 1:-**

```
class A
{
    class B
    {
    }
}
```

Case 2:-

```
class A
{
    interface It1
    {
    }
}
```

Case 3:-

```
interface It1
{
    interface It2
    {
    }
}
```

Case 4:-

```
interface It1
{
    class A
    {
    }
}
```

- ✓ Abstract
- ✓ Strictfp
- ✓ Final
- ✓ Default (no modifier)

The applicable modifiers on Outer classes

- ✓ Public

The applicable modifiers on inner classes

- ✓ final
- ✓ abstract
- ✓ public
- ✓ private
- ✓ protected
- ✓ strictfp
- ✓ static
- ✓ default (no modifier)

Example 1:- private : it is possible to access private members inside the class only.

```
class Outer
{
    private class Inner1
    {
    }
}

class Test
{
    public static void main(String[] args)
    {
        Outer.Inner1 i = new Outer().new Inner1();
    }
}
```

Compilation Error: Outer.Inner1 has private access in Outer

Example 2:- abstract : it is not possible to instantiate the class.

```
class Outer
{
    abstract class Inner
    {
    }
}
```

```

    }
    public static void main(String[] args)
    {
        Outer.Inner i = new Outer().new Inner();
    }
}

Compilation Error: Outer.Inner is abstract; cannot be instantiated
Outer.Inner i = new Outer().new Inner();

```

Example 3:- final : it is not possible to create sub class.

```

class Outer
{
    final class Inner1
    {
    }

    class Inner2 extends Inner1
    {
    }
}

```

Compilation error : cannot inherit from final Outer.Inner1

Nested classes

- 1) What are the advantages of inner classes?
- 2) How many types of nested class?
- 3) How many types of inner classes?
- 4) What do you by static inner classes?
- 5) What is the relation between inner classes & nested classes?
- 6) The inner class is able to access outer class private properties or not?
- 7) The outer class is able to access inner classes properties& methods or not?
- 8) How to create object inner class and outer class?
- 9) .class file names of Inner & outer classes?
- 10) The outer class object is able to call inner class properties & methods or not?
- 11) The inner class object is able to call outer class properties and methods or not?
- 12) What is the difference between normal inner classes and static inner classes?
- 13) How to prevent the object creation of inner classes?
- 14) One inner class able to extends another inner class or not?
- 15) What do you mean by anonymous inner classes?
- 16) What do you mean by method local inner classes?
- 17) Is it possible to create inner class object without outer class object?
- 18) Java supports inner methods concept or not?
- 19) Is it possible to declare main method inside inner classes & outer class?
- 20) Is it possible to declare constructors inside inner classes?
- 21) If outer class variables and inner class variables are having same name then how to represent outer class variables and how to represent inner class variables?
- 22) Is it possible to declare same method in both inner class and outer class?
- 23) What are the applicable modifiers on Outer classes?
- 24) What are the applicable modifiers on inner classes?

25) What are the applicable modifiers on method local inner classes?

Functional interface & lambda expressions

java 8 version

Functional interface:- java 8 version

- ✓ Interface with only one abstract method is called Functional interface.
- ✓ The functional interface allows default & static methods also.
- ✓ This interface is also known as single abstract method interface (SAM interface).
- ✓ Lambda expression is used to provide the implementations of abstract methods.
- ✓ @FunctionalInterface annotation which can be used for compilation errors when the interface you annotated violates the contracts of functional interface.

Examples: predefined Function Interface

```
java.awt.event.ActionListener  
java.lang.Runnable  
java.util.Comparator
```

case 1:**valid** : Functional interface contains only one abstract method.

```
@FunctionalInterface  
interface Greetings  
{      void morning();  
}
```

case 2:**Valid** : Functional interface contains only one abstract method & default, static methods also.

```
@FunctionalInterface  
interface Greetings  
{      void morning();  
      default void even(){  
      static void goodnight(){  
    }
```

case 3: Invalid : The functional interface allows only one abstract method but here we are declared more than one abstract method.

```
@FunctionalInterface
interface Greetings
{
    void morning();
    void evening();
}
```

Lambda expression

- ✓ Lambda expression is introduced in java8 version.
- ✓ Lambda expression will reduce the length of the code & reduce the byte code increases performance of the application.
- ✓ Used to provide the implementation of Functional interface.

Syntax :-

(Argument-list) -> {body}

Argument-list: It can be empty or non-empty as well.

Arrow-token: It is used to link arguments-list and body of expression.

Body: It contains expressions and statements for lambda expression.

Lambda expression Body:-

- ✓ Body can be a single expression or a statement block.
- ✓ If we are declaring only one statement braces are optional.
- ✓ If we are declaring more than one statement brace are mandatory.
- ✓ In body if we are declaring only return value specifying return is also optional.

Example :-

```
interface Message
{
    void morn();
}

class Test
{
    public static void main(String[] args)
    {
        //application with anonymous inner class
        Message m = new Message()
        {
            public void morn(){System.out.println("good morning Ratan");}
        };
    }
}
```

```

    };
    m.morn();

//Application with Lambda Expression
Message m1 = ()->System.out.println("good morning Ratan");
m1.morn();
}
}
}

```

Example :-Lambda expression with arguments

```

interface Message
{
    void morn(String str,int a);
}

class Test
{
    public static void main(String[] args)
    {
        //application with anonymous inner class
        Message m = new Message()
        {
            public void morn(String str,int a)
            {System.out.println("arg-1 "+str);
             System.out.println("arg-2 "+a);

            }
        };
        m.morn("Anu",10);
    }
}

```

//Application with Lambda Expression

```

Message m1 = (String str,int a)->{System.out.println("arg-1 "+str);
                                     System.out.println("arg-2 "+a);};

m1.morn("Anu",10);

Message m2 = (str,a)->{System.out.println("arg-1 "+str);
                        System.out.println("arg-2 "+a);};

m2.morn("Anu",10);
}
}

```

- ✓ A lambda expression can contain zero or more arguments.
- ✓ You can eliminate the argument type while passing it to lambda expressions

(int a) and (a) both are same.
✓ Invalid : Message m2 = (str,int a)->{ -----};

Example :- Lambda expression with return value

```
interface Printable
{
    public String print();
}

class Test
{
    public static void main(String[] args)
    {
        Printable p = ()->{return "hi ratan";};
        System.out.println(p.print());

        Printable p1 = ()->"hi ratan";
        System.out.println(p1.print());
    }
}
```

Example :

```
interface isTypeOne
{
    void hasOne();
}

interface isTypeTwo
{
    void hasTwo();
}

class Test
{
    void first(isTypeOne o)
    {
        o.hasOne();
    }

    void second(isTypeTwo t)
    {
        t.hasTwo();
    }

    public static void main(String[] args)
    {
        Test t = new Test();
        t.first(()->System.out.println("one"));
        t.second(()->System.out.println("second"));
    }
}
```

Example :-

```
interface Executable
{
    void execute();
}

class Runner
{
    public void run(Executable e)
    {
        System.out.println("run method code....");
        e.execute();
    }
}
```

```

        }
    };
public class Test
{
    public static void main(String[] args)
    {
        Runner r = new Runner();
        //anonymous inner class
        r.run(new Executable(){
            public void execute()
            {
                System.out.println("execute method block of java code.....");
            }
        });

        //lambda expression
        r.run(() -> System.out.println("execute method block of code....."));
    }
}

```

Example:- lambda expression ambiguity problems

```

interface Executable
{
    int execute(int a);
}

interface StringExecutable
{
    int execute(String a);
}

class Runner
{
    public void run(Executable e)
    {
        System.out.println("run method code....");
        int x = e.execute(10);
        System.out.println("return value="+x);
    }

    public void run(StringExecutable e)
    {
        System.out.println("run method code....");
        int x = e.execute("ratan");
        System.out.println("return value="+x);
    }
};

public class Test
{
    public static void main(String[] args)
    {
        Runner r = new Runner();
        //lambda expression
        r.run((int a) -> 10);
        r.run((String a) -> 20);
        // r.run(a ->100);      compilation error: ambiguity problem
    }
};

```

Example :-

```

interface Executable
{
    int execute(int a,int b);
}

class Runner
{
    public void run(Executable e)
    {
        int x = e.execute(100,200);
        System.out.println("return value="+x);
    }
};

public class Test
{
    public static void main(String[] args)
    {
        int c=10;
        new Runner().run((int a,int b) -> a+b+c);
    }
};

```

Annotations (meta data)

- ✓ Annotations are introduced in 1.5 versions it represent metadata of the program.
- ✓ Annotations can be used attach the some additional information to the interfaces,classes,methods,constructors which can be used by compiler and JVM.
- ✓ Annotations are executed by using predefined tool APT(Annotation Processing Tool).

Meta annotations:-

- ✓ It specifies information about annotation. These are present in **java.lang.annotation** package.
- ✓ The meta annotations are
 - **@Retention**
 - **@Target**
 - **@Documented**
 - **@Inherited**
 - **@Repeatable**

Annotation @Retention : Retention policy determines at what point the annotation is available
@Retention(RetentionPolicy.SOURCE)

The marked annotation is retained only in the source level and is ignored by the compiler.

@Retention(RetentionPolicy.CLASS)

The marked annotation is retained by the compiler at compile time, but is ignored by (JVM).

@Retention(RetentionPolicy.RUNTIME)

The marked annotation is retained by the JVM so it can be used by the runtime environment

Annotation Target specification: @Target

- ✓ Marks another annotation to restrict what kind of Java elements the annotation may be applied to.
- ✓ The target annotation specify one of the following element,
@Target(ElementType.ANNOTATION_TYPE) can be applied to an annotation type.

<code>@Target(ElementType.CONSTRUCTOR)</code>	<i>can be applied to a constructor.</i>
<code>@Target (ElementType.FIELD)</code>	<i>can be applied to a field or property.</i>
<code>@Target(ElementType.LOCAL_VARIABLE)</code>	<i>can be applied to a local variable.</i>
<code>@Target(ElementType.METHOD)</code>	<i>can be applied to a method-level annotation.</i>
<code>@Target(ElementType.PACKAGE)</code>	<i>can be applied to a package declaration.</i>
<code>@Target(ElementType.PARAMETER)</code>	<i>can be applied to the parameters of a method.</i>
<code>@Target(ElementType.TYPE)</code>	<i>can be applied to any class, interface or enumeration.</i>

@Documented - Marks another annotation for inclusion in the documentation.

Whenever we are using this annotation those elements should be documented by using javadoc tool.

@Inherited -

Marks another annotation to be inherited to subclasses of annotated class (by default annotations are not inherited to subclasses).

@Repeatable - Specifies that the annotation can be applied more than once to the same declaration.

Uses of annotations:-

❖ Information for the compiler

Annotations are used by the compiler to detect suppress warnings or errors based on rules.

*Example :- `@Override` this one makes the compiler to check the method correctly override or not
`@FunctionalInterface` makes the compiler to validate interface is functional interface or not.*

❖ Documentation

Annotations can be used in software applicationsto ensure the quality of the code like bug finding, report generation...etc

❖ Code generation

Annotations used to generate the code or xml files using meta data information present in the code.

❖ Runtime processing

Annotations that are used in runtime objectives like unit testing, dependency injection...etc

1) `@Override`

- ✓ *It instructs the compiler to check parent class method is overriding in child class or not if it is not overriding compiler will generate error message.*
- ✓ *In below example if are not declaring `@override` annotation a new method of marry(int a) created in child class.*

Predefined support :-

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

Example :

```

class Parent
{
    void marry()
    {}
};

class Child extends Parent
{
    @Override
    void marry()
    {
    }
};

E:\>javac Test.java
error: method does not override or implement a method from a supertype

```

2) @SuppressWarnings –

- ✓ Instructs the compiler to suppress the compile time warnings specified in the annotation parameters.

Predefined support :

```

@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
    String[] value();
}

```

Example-1:-

```

import java.util.*;
class Student
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add("ratan");
        al.add("anu");
        al.add("sravya");
        System.out.println(al);
    }
};

```

3) @Deprecated - Marks the method as obsolete. Causes a compile warning if the method is used.

This annotation represent the marked element is no longer be used. The compiler generates warning message when we used that marked element.

Example :-

```
class Test
```

```
{      @Deprecated
    void m1()
    {
        System.out.println("m1 method");
    }
}
class Demo
{
    @SuppressWarnings("deprecation")
    public static void main(String[] args)
    {
        new Test().m1();
    }
}
```

Example -2:-

```
import java.awt.*;
class Student
{
    public static void main(String[] args)
    {
        Frame f = new Frame();
        f.show();
    }
}
```

- 4) **@FunctionalInterface** - Specifies that the type declaration is intended to be a functional interface, since Java 8.Annotations applied to other annotations (also known as "Meta Annotations"):

Predefined support :

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FunctionalInterface { }
```

Example :

```
@FunctionalInterface
interface It1
{
    void m1();
}
```

ENUMARATION (1.5 version)

- ✓ Enumeration is used to declare group of named constants.
- ✓ Declare the enum by using enum keyword.
- ✓ Enum used to declare the constants like
 - Directions
 - Month names
 - Week days.....etc

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

Without enum if you want constants:

```
class Test
{
    public static final String str1;
    public static final String str2;
    public static final String str3;
}
```

With enum if we want constants

```
enum Directions
{
    NORTH, SOUTH, EAST, WEST;
}
```

ex 2: It is possible to declare the enum inside the class.

```
class Gmail
{
    enum Mail
    {
        INBOX, COMPOSE, SENT;
    }

    public static void main(String[] args)
    {
        Mail[] mm = Mail.values();
        for (Mail mmm : mm)
        {
            System.out.println(mmm);
        }
    }
}
```

```

        }
    }
}
```

ex 3 : inside the enum possible to declare constructor this constructor executed for every constant because every constant is object.

```

enum Week
{
    MON,TUE,WED;
    Week()
    {
        System.out.println("0-arg cons");
    }
}
class Test
{
    public static void main(String[] args)
    {
        Week[] w = Week.values();
        for (Week ww : w)
        {
            System.out.println(ww+" "+ww.ordinal());
        }
    }
}
```

ex -1:

```

enum Week
{
    MON,TUE,WED; //public static final
}
class Test
{
    public static void main(String[] args)
    {
        Week w1 = Week.MON;
        Week w2 = Week.TUE;
        Week w3 = Week.WED;
        System.out.println(w1+" "+w2+" "+w3);

        Week[] w = Week.values();
        for (Week ww : w)
        {
            System.out.println(ww+" "+ww.ordinal());
        }
    }
}
```

//internal enum code

```

public final class Week extends Enum
{
    private Week(String s, int i)
    {
        super(s, i);
    }
    public static final Week MON;
    public static final Week TUE;
    public static final Week WED;
    private static final Week VALUES[];
```

static

```

{ MON = new Week("MON", 0);
TUE = new Week("TUE", 1);
WED = new Week("WED", 2);
VALUES = (new Week[] {
    MON, TUE, WED
});
}
}

```

1. enums internally treated as class format.
2. enums are by default final so we can not create child enums.
3. enums are by default public
4. the default super cls of enum is : Enum
5. enum keyword to declare the enum : Enum is default super
6. enums also compiler generated .class files
7. enums contains private cons : so not possible to create the obj
8. Every enum constant by default **public static final**.
9. enums to declare user defined cons
10. every enum constant is object.

ex 4: Observation

Case 1: Inside the enum if are declaring only constants these constants ends with semicolon is optional.

```

enum Week
{
    MON,TUE,WED
}

```

case 2: Inside the enum if we are declaring constants along with some other elements like constructor or method in this case group of constants must be first line must ends with semicolon.

```

enum Week
{
    MON,TUE,WED;
    Week()
    {
        System.out.println("0-arg cons");
    }
}

```

ex 5: inside the enum possible to declare the constructor,main method instance,static blocks.

```

enum Week
{
    MON,TUE(10),WED(10,20); //public static final
    static
    {
        System.out.println("staic block");
    }
    {
        System.out.println("ins block");
    }
    Week()
    {
        System.out.println("0-arg cons");
    }
    Week(int a)
    {
        System.out.println("1-arg cons");
    }
}

```

```

Week(int a,int b)
{
    System.out.println("2-arg cons");
}
public static void main(String[] args)
{
    System.out.println("enum main method");
}
}

```

if the application contains static variables & static blocks then the execution order is top to bottom.

- a. first static constants are executed
- b. static blocks are executed

E:\>java Week

```

ins block
0-arg cons
ins block
1-arg cons
ins block
2-arg cons
staic block
enum main method

```

ex 6 : It is possible to take the enum as a switch argument.

```

enum Day
{
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
class Test
{
    public static void main(String args[])
    {
        Day day = Day.SUNDAY;
        switch(day)
        {
            case SUNDAY: System.out.println("sunday");
                           break;
            case MONDAY: System.out.println("monday");
                           break;
            default: System.out.println("other day");
                      break;
        }
    }
}

```

in above example we are passing enum as switch argument so the case labels must be enums constants.

ex 7: Accessing the enums constants

- a. using normal import
- b. using static import

//Fish.java

```

package com.tcs.enumscons;
enum Fish
{
    GOLD,STAR,CAT;
}

```

//normal import : access the static data using class-name

```

package com.tcs.client;
import com.tcs.enumscons.Fish;
class MainTest
{
    public static void main(String[] args)
    {
        System.out.println(Fish.STAR);
        System.out.println(Fish.GOLD);
        System.out.println(Fish.CAT);
    }
}
//static import : access static data directly without using class-name
package com.tcs.client;
import static com.tcs.enumscons.Fish.*;
class MainTest
{
    public static void main(String[] args)
    {
        System.out.println(STAR);
        System.out.println(GOLD);
        System.out.println(CAT);
    }
}

```

Enumeration

- 1) What is the purpose enum in java?
- 2) How to declare enum & compiler will generate .class files or not?
- 3) enum constants are by default modifiers are?
- 4) What is the difference enum & Enum?
- 5) What is the purpose of values() & ordinal() methods?
- 6) What is the difference between java enum & other language enums.
- 7) Is it possible to declare main method & constructor inside the enum or not?
- 8) Is it possible to provide parameterized constructor inside the enum?
- 9) Inside the enum group of constants ends with semicolon is optional or mandatory?
- 10) What is the difference between java enum and java class?
- 11) Is it possible to create object for enum or not?
- 12) For enum inheritance concept is applicable or not?
- 13) When the enum constants are loaded?
- 14) Enums are able to implement interfaces or not?
- 15) What is the difference between enum& Enumeration & Enum?
- 16) Can you use enum constants switch case in java?
- 17) What is the modifier applicable for enum constructor?
- 18) Is it possible to declare the enum inside the class & inside the method ?
- 19) What is the difference between normal import & static import?
- 20) When we access enum constants outside of the package directly without using enum name then
normal import required or static import required?



Material by Mr. Ratan.....

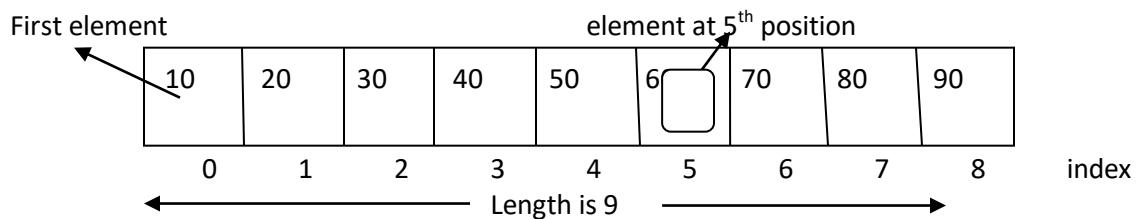
***** ***Thank you : Completed: Thank you: Believes only practical Knowledge*** *****

Arrays

- ✓ Arrays are used to represent group of elements as a single entity but these elements are homogeneous & fixed size.
 - ✓ The size of Array is fixed it means once we created Array it is not possible to increase and decrease the size.
 - ✓ Array in java is index based first element of the array stored at 0 index.
 - ✓ By using arrays possible to store primitive data & object data.

Advantages of array:-

- ✓ **Code optimization:** Instead of declaring individual variables we can declare group of elements by using array it reduces length of the code.
 - ✓ **Flexibility:** We can store the group of objects easily & we are able to retrieve the data easily.
 - ✓ **RandomAccess :** We can access the random elements present in the any location based on index.



There are two approaches to declare the array

Approach 1:-

```

int[] a;           declaring
a = new int[5];   instantiation
a[0]=10;          initialization
a[1]=20;

```

Approach 2:- int a[]={10,20,30,40}; //declaring, instantiation, initialization

Possible syntaxes:

```

int[] values;
int []values;
int values[];

```

One Dimensional array

Initialization `int a[] = new int [12];`

Value	1	2	3	4	5	6	7	8	9	10	11	12
Index	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]

`System.out.print(a[5]);`

Output: 6

Example :- different ways to print array data.

```

package com.dss;
public class Test
{
    public static void main(String[] args)
    {
        int[] a = {10,20,30};
        //1-way
        System.out.println(a[0]);
        System.out.println(a[1]);
        System.out.println(a[2]);

        //2-way
        for(int i=0;i<a.length;i++)
        {
            System.out.println(a[i]);
        }

        //3-way
        for(int aa:a)
        {
            System.out.println(aa);
        }
    }
}

```

Example :

- ✓ when we create the array the array is created with default values later initialization is performed.

```

package com.dss;
public class Test
{
    public static void main(String[] args)
    {
        int[] a = new int[3];
        for(int aa:a)
        {
            System.out.println(aa);
        }

        String[] s = new String[3];
        s[0]="ratan";
        s[1]="anu";
        for(String ss:s)
        {
            System.out.println(ss);
        }
    }
}

```

Example : finding null index values.

```

String[] s = new String[3];
s[0]="ratan";
for(int i=0;i<s.length;i++)
{
    if(s[i]==null)
        System.out.println(i);
}

```

Example:- adding the objects into Array and printing the object data.

```

class Test
{
    public static void main(String[] args)
    {
        Emp e1 = new Emp(111,"ratan");
        Emp e2 = new Emp(222,"anu");
        Emp e3 = new Emp(333,"sravya");

        Emp[] e = new Emp[3];
        e[0]=e1;
        e[1]=e2;
        e[2]=e3;
        for (Emp ee:e)
        {
            //System.out.println(ee);      in this line toString() executed prints hashCode
            System.out.println(ee.eid+"---"+ee.ename)
        }
    }
}

```

Example:- printing array elements with elements and default values.

```

class Test
{
    public static void main(String[] args)
    {
        Emp[] e = new Emp[5];
        e[0]=new Emp(111,"ratan");
        e[1]=new Emp(222,"anu");

```

```

e[2]=new Emp(333,"sravya");
for (Object e1:e)
{
    if (e1 instanceof Emp)
    {
        Emp e2 = (Emp)e1;
        System.out.println(e2.eid+"---"+e2.ename);
    }
    if (e1==null)
    {
        System.out.println(e1);
    }
}
}

```

Output:-

```

E:\>java Test
111---ratan
222---anu
333---sravya
null
null

```

- ✓ In above example if the location contains null value but we are trying to call the **e2.eid**JVM will generate **NullPointerException** so to overcome this problems use if-else condition to check the data.

Example:-process of adding different types Objects in Object array**Emp.java:**

```

class Emp
{
    int eid;
    String ename;
    Emp(int eid,String ename)
    {//conversion of local to instance
        this.eid=eid;
        this.ename=ename;
    }
}

```

Student.java:-

```

class Student
{
    int sid;
    String sname;
    Student(int sid,String sname)
    {//conversion of local to instance
        this.sid=sid;
        this.sname=sname;
    }
}

```

Test.java

```

class Test
{
    public static void main(String[] args)
    {
        Object[] a= new Object[6];
        a[0]=new Emp(111,"ratan");
        a[1]=new Integer(10);
        a[2]=new Student(1,"anu");
        a[3]=new String("ratan");

        for (Object a1:a)

```

```

    {
        if (a1 instanceof Emp)
        {
            Emp e1 = (Emp)a1;
            System.out.println(e1.eid+"---"+e1.ename);
        }
        if (a1 instanceof Student)
        {
            Student s1 = (Student)a1;
            System.out.println(s1.sid+"---"+s1.sname);
        }
        if (a1 instanceof Integer)
        {
            System.out.println(a1);
        }
        if (a1 instanceof String)
        {
            System.out.println(a1);
        }
        if (a1==null)
        {
            System.out.println(a1);
        }
    }
}

```

Example : Finding minimum & maximum element of the array

```

class Test
{
    public static void main(String[] args)
    {
        int[] a = new int[]{10,20,5,70,4};
        for (int a1:a)
        {
            System.out.println(a1);
        }

//minimum element of the Array
        int min=a[0];
        for (int i=1;i<a.length;i++)
        {
            if (min>a[i])
            {
                min=a[i];
            }
        }
        System.out.println("minimum value is =" +min);

//maximum element of the Array
        int max=a[0];
        for (int i=1;i<a.length;i++)
        {
            if (max<a[i])
            {
                max=a[i];
            }
        }
    }
}

```

```

        }
    }
    System.out.println("maximum value is =" + max);
}
}

```

Example :- copy the data from one array to another array

```

class Test
{
    public static void main(String[] args)
    {
        int[] copyfrom={10,20,30,40,50,60,70,80};
        int[] copyto = new int[7];
        System.arraycopy(copyfrom,1,copyto,0,7);
        for (int cc:copyto)
        {
            System.out.println(cc);
        }
    }
}

```

Example :- copy the data from one array to another array

```

class Test
{
    public static void main(String[] args)
    {
        int[] copyfrom={10,20,30,40,50,60,70,80};
        int[] newarray=java.util.Arrays.copyOfRange(copyfrom,1,4);
        for (int aa:newarray)
        {
            System.out.println(aa); //20 30 40
        }
    }
}

```

Example : To get the class name of the array

```

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

```

Example :- Taking array elements from keyboard by using scanner class.

```

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

        //printing the data
        for (int a1:a)

```

```

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

```

Example : Method parameter is array & method return type is array

```

class Test
{
    static void m1(int[] a) //method parameter is array
    {
        for (int a1:a)
        {
            System.out.println(a1);
        }
    }
    static int[] m2() //method return type is array
    {
        System.out.println("m1 method");
        return new int[]{100,200,300};
    }
    public static void main(String[] args)
    {
        Test.m1(new int[]{10,20,30,40});
        int[] x = Test.m2();
        for (int x1:x)
        {
            System.out.println(x1);
        }
    }
}

```

Example :- find the sum of the array elements.

```

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

```

declaration of multi dimensional array:-

```

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

```

Example :-

```

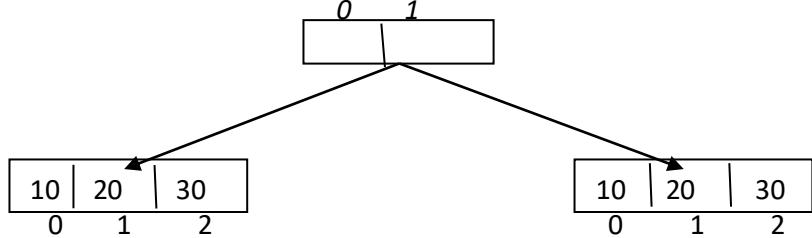
class Test
{
    public static void main(String[] args)
    {
        int[][] a={{10,20,30},{40,50,60}};
    }
}

```

```

        System.out.println(a[0][0]); //10
        System.out.println(a[1][0]); //40
        System.out.println(a[1][1]); //50
    }
}

```

**Example:-**

```

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

```

Collections framework (java.util)**Collection definition :**

- ✓ The main objective of collections framework is to represent group of object as a single entity.
- ✓ In java Collection framework provide very good architecture to store and manipulate the group of objects.
- ✓ Collection API contains group of classes and interfaces that makes it easier to handle group of objects.

Arrays vs. Collections:

Both Arrays and Collections are used to represent group of objects as a single entity but the differences are as shown below.

Limitations of Arrays

- 1) Arrays are used to store homogeneous data (similar data).
- 2) Arrays are capable to store primitive & Object type data

- 3) Arrays are fixed in size, it means once we created array it is not possible to increase & decrease the size based on our requirement.
- 4) Arrays does not contain underlying Data structure hence it is not supporting predefined methods so operations are become complex.

- 5) With respect to memory arrays are not recommended to use.
- 6) If you know size in advance arrays are recommended to use because it provides good performance.

Advantages of Collections

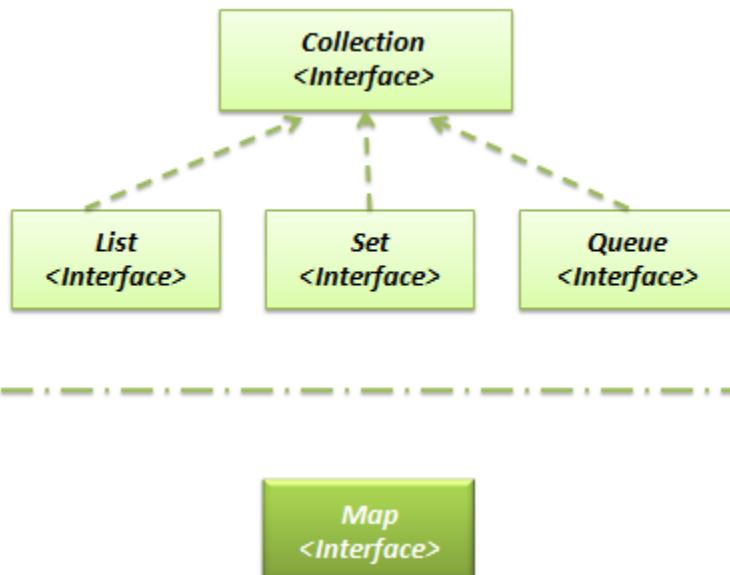
- 1) Collections store both homogeneous & heterogeneous data.
- 2) Collections are capable to store only object data.

- 3) Collections can grow and shrink in size automatically when objects are added or removed.
- 4) Collection classes contain underlying data structure hence it supports predefined methods so operations are easy.

- 5) With respect to memory collections are recommended to use.
- 6) In performance point of view collections will give low performance compare to arrays.



Collections Framework divided into two parts



- ✓ All collection framework classes & interfaces : **java.util package.**
- ✓ The root interface of the collection classes is : **Collection**
- ✓ The root interface of the map classes is : **Map**
- ✓ The parent interface of the Collation is : **Iterable (java.lang)**
- ✓ Collection framework introduced in : **jdk 1.2 version**

Collection vs. Collections:

Collection is a root interface of collection frame work whereas Collections is utility class it contains methods to perform operations.

The key interfaces of collection framework:

1. Java.util.Collection
2. Java.util.List
3. Java.util.Set
4. Java.util.SortedSet
5. Java.util.NavigableSet
6. Java.util.Queue
7. Java.util.Enumeration
8. Java.util.Iterator
9. Java.util.ListIterator
10. Java.lang.Comparable --->java.lang package
11. Java.util.Comparator

12. Java.util.Map
13. Java.util.SortedMap
14. Java.util.NavigableMap
15. Map.Entry

Characteristics of Collection frame work classes:

The collections framework contains group of classes but every class is used to represent group of objects as a single entity but characteristics are different.

1) The collection framework classes introduced Versions**2) Heterogeneous data allowed or not allowed.**

All most all collection framework classes allowed heterogeneous data except two classes

- i. TreeSet
- ii. TreeMap

3) Null insertion is possible or not possible.**4) Duplicate objects are allowed or not allowed.**

Inserting same object more than one time is called duplication.

```
add(e1)  
add(e1)
```

5) Insertion order is preserved or not preserved.

In which order we are inserting element same order output is printed then say insertion order is preserved otherwise not.

Input --->e1 e2 e3 output --->e1 e2 e3 insertion order is preserved

Input --->e1 e2 e3 output --->e2 e1 e3 insertion order is not-preserved

6) Collection classes' methods are synchronized or non-synchronized.

If the methods are synchronized only one thread is allowed to access, these methods are threading safe but performance is reduced.

If the methods are non-synchronized multiple threads are able to access, these methods are not thread safe but performance is increased.

Collection framework classes are non-synchronized except **java 1.0 version classes**(*Vector,HashTable*)

7) Collection classes underlying data structures.

Every collection class contains underlying data structure hence it supports predefined methods.

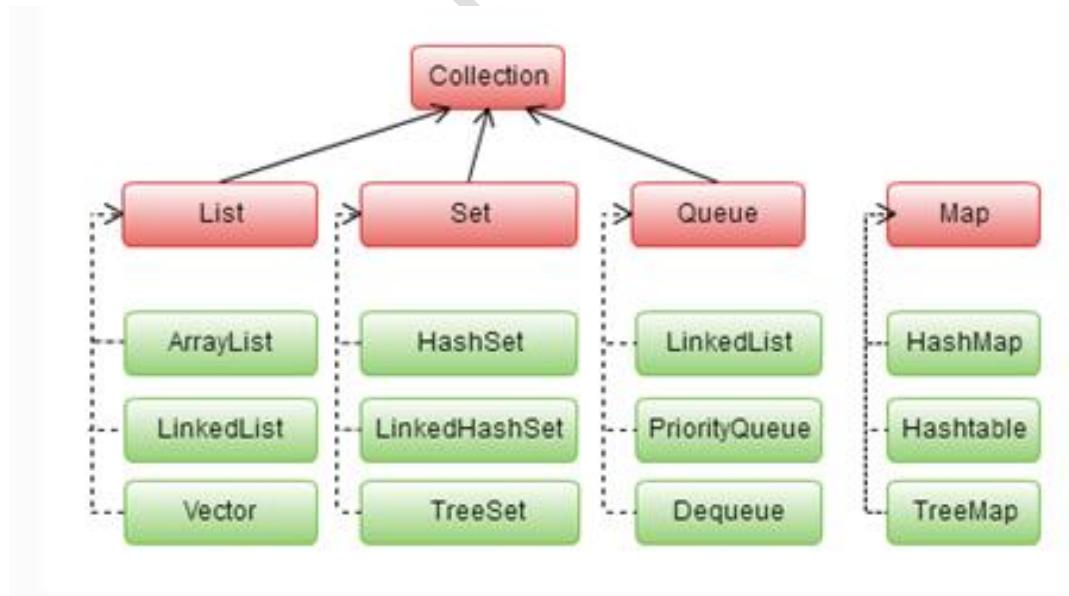
8) Collection classes supported cursors.

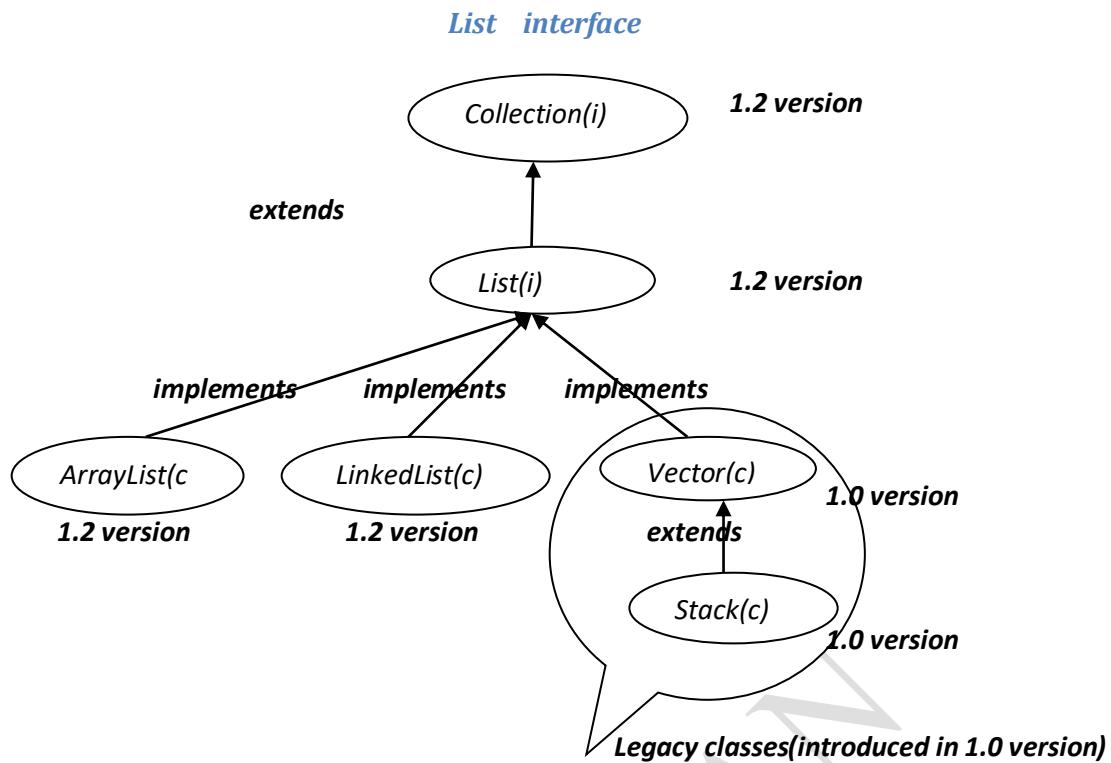
The collection classes are used to represent group of objects as a single entity & To retrieve the objects from collection class we are using cursors.

		Implementations				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table & Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Deque		ArrayDeque		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

Summary of collections framework:

- ✓ List interface implementation classes.
- ✓ Set interface implementation classes.
- ✓ Queue interface implementation classes.
- ✓ Map interface implementation classes
- ✓ Collection data sorting.
- ✓ Cursors in collections.





I = Interface C = class

List interface Implementation classes:

- 1) ArrayList
- 2) LinkedList
- 3) Vector
- 4) Stack

Legacy classes: The java classes which are introduced in 1.0 version .

- 1) HashTable
- 2) Properties
- 3) Stack
- 4) Vector
- 5) Dictionary <abstract class>
- 6) Enumeration<interface>

Java.util.ArrayList:

To check the predefined support use javap command,

D:\ratan>javap java.util.ArrayList

```

public class java.util.ArrayList<E>
    extends java.util.AbstractList<E>
    implements java.util.List<E>,
               java.util.RandomAccess,
               java.lang.Cloneable,
               java.io.Serializable
  
```

- ✓ Usually collection data is transferred from one JVM instance to other JVM instance. To support this requirement, every collection class must be inherited from `java.io.Serializable` interface. `Serializable` is a marker interface providing serialization capabilities.
- ✓ Also, collection data can be copied. Hence, every collection class must be inherited from `java.lang.Cloneable` interface. `Cloneable` is a marker interface providing Cloning capabilities.
- ✓ `ArrayList` and `Vector` classes implements `java.util.RandomAccess`, which is marker interface, to indicate that they support constant data access time.

ArrayList:

- 1) `ArrayList` Introduced in 1.2 version.
- 2) `ArrayList` stores Heterogeneous objects(different types).
- 3) In `ArrayList` it is possible to insert `Null` objects.
- 4) Duplicate objects are allowed.
- 5) `ArrayList` preserved Insertion order it means whatever the order we inserted the data in the same way output will be printed.
- 6) `ArrayList` methods are non-synchronized methods.
- 7) The underlying data structure is Resizable array.
- 8) By using cursor we are able to retrieve the data from `ArrayList` : **Iterator , ListIterator**

ex-1:

Case 1 : Up to 1.4 versions we must create wrapper class object then add that object into `ArrayList`.

```
import java.util.ArrayList;
class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(new Integer(10));
        al.add(new Character('c'));
        al.add(new Double(10.5));
        System.out.println(al);
        System.out.println(al.toString());
    }
}
```

Case 2:

From 1.5 versions onwards add the primitive data into `ArrayList` , that data is automatically converted into wrapper object format is called Auto boxing.

Code before compilation:

```
import java.util.ArrayList;
class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(10);
        al.add('a');
```

```

        al.add(10.5);
        System.out.println(al.toString());
    }
}

```

Code after compilation:

```

import java.util.ArrayList;
class Test
{
    public static void main(String args[])
    {
        ArrayList arraylist = new ArrayList();
        arraylist.add(Integer.valueOf(10));
        arraylist.add(Character.valueOf('a'));
        arraylist.add(Double.valueOf(10.5));
        System.out.println(arraylist.toString());
    }
}

```

- ✓ When we add the primitive data into ArrayList it is automatically converted into wrapper object format is called autoboxing.
- ✓ Whenever we are printing arraylist reference internally it calls toString() method on every object.
- ✓ toString() present in object class returns String representation of object(class-name@hashcode).
- ✓ String, StringBuffer, all wrapper classes toString() method overriding to return content of the object.

ex-2: working with object data.***Emp.java:***

```

class Emp
{
    int eid;
    String ename;
    Emp(int eid, String ename)
    {
        //conversion of local to instance
        this.eid=eid;
        this.ename=ename;
    }
}

```

Student.java

```

class Student
{
    int sid;
    String sname;
    Student(int sid, String sname)
    {
        //conversion of local to instance
        this.sid=sid;
        this.sname = sname;
    }
}

```

Test.java

```

package com.dss;
import java.util.ArrayList;
public class Test {
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        al.add(10);
        al.add("ratan");
        al.add(new Emp(111, "ratan"));
        al.add(new Student(1, "anu"));
        al.add(null);
        System.out.println(al.toString());
    }
}

```

```

for(Object o : al)
{
    if(o instanceof Emp)
    {
        Emp ee = (Emp)o;
        System.out.println(ee.eid+"--"+ee.ename);
    }
    if(o instanceof Student)
    {
        Student ss = (Student)o;
        System.out.println(ss.sid+"---"+ss.sname);
    }
    if(o instanceof String)
    {
        System.out.println(o);
    }
    if(o instanceof Integer)
    {
        System.out.println(o);
    }
    if(o==null)
    {
        System.out.println(o);
    }
}
}
}

```

ex 3:

- ✓ Arrays are by default type safe it means the array contains only specific type of data.
 Int array ---> stores only int data
 String array ---> stores only String data
- ✓ Collections are not type safe (no guarantee on data) it means the collections can store different types of objects.
`ArrayList al = new ArrayList();
al.add(10);
al.add("ratan");`
- ✓ If the collections are not type safe while reading the data at runtime we have to perform,
 1. Type checking
 2. Type casting

if the collections are not type safe, while reading the data we have to do (typecheckig& type casting).

```

package com.dss;
import java.util.ArrayList;
public class Test3 {
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        al.add(new Emp(111, "ratan"));
        al.add(new Student(1, "durga"));

        Student s = (Student)al.get(1);
        System.out.println(s.sid+" "+s.sname);
    }
}

```

```

Object o = al.get(0);
if(o instanceof Student)
{
    Student s1 = (Student)o;
    System.out.println(s1.sid+" "+s1.sname);
}
if(o instanceof Emp)
{
    Emp e1 = (Emp)o;
    System.out.println(e1.eid+" "+e1.ename);
}
}
}

```

To overcome above two problems to provide the type safety to the collections use generics.

- ✓ The generics are used to provide the type safety to the collections, it allows only similar type of data, if we are trying to add different types of object compiler generates error message.
- ✓ When we provide type safety to the collection the advantages are,
 - No type checking.
 - No type casting.

Generic version of ArrayList used to store only homogeneous data.(type safety)

```

ArrayList<String> al = new ArrayList<String>();
al.add("ratan");
al.add("anu");

```

ex-4:

```

import java.util.ArrayList;
public class Test {
    public static void main(String[] args) {
        //arrays are type safe
        int[] a={10,20,30};
        for(int aa:a)
        {
            System.out.println(aa);
        }
    }
}

//normal version of collection: not type safety
ArrayList al = new ArrayList();
al.add(new Emp(111, "ratan"));
al.add(new Student(1, "anu"));
for(Object o:al)
{
    if(o instanceof Emp)
    {
        Emp e = (Emp)o;
        System.out.println(e.eid+"---"+e.ename);
    }
    if(o instanceof Student)
    {
        Student s = (Student)o;
        System.out.println(s.sid+"---"+s.sname);
    }
}

```

```
//generic version of collection: provides type safety
ArrayList<Emp> a1 = new ArrayList<Emp>();
a1.add(new Emp(111,"ratan"));
a1.add(new Emp(222,"anu"));
for (Emp e : a1)
{
    System.out.println(e.eid+"---"+e.ename);
}
}
```

Observation : if the generic version contains null values we have to print the data using below code.

```
ArrayList<Student> a2 = new ArrayList<Student>();
a2.add(new Student(1,"ratan"));
a2.add(new Student(2,"anu"));
a2.add(null);
for (Student s : a2)
{
    if(s==null)    System.out.println(s);
    else          System.out.println(s.sid+"---"+s.sname);
}
```

Note : Arrays are used to store the homogeneous data & collection generic version used to store homogeneous data but collections provides more flexibility with respect to the memory & operations.

Ex-5: Basic operations on ArrayList

```
package com.dss;
import java.util.ArrayList;
public class Test5 {
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        al.add(10);
        al.add(10.5);
        al.add("durga");
        al.add("ratan");
        al.add(10);
        al.add(null);
        System.out.println(al);
        System.out.println(al.size());           //finding size of the ArrayList

        al.add(3, "anu");                      //Adding the data at specified index
        System.out.println(al);                //|[10, 10.5, durga, anu, ratan, 10, null]

        al.remove(2);                         // int value is by default index format to remove
        al.remove("anu");                     // passing object data to remove
        System.out.println(al);                //|[10, 10.5, ratan, 10, null]

        al.set(1, "xxx");                    //set() : used to replace the data
        System.out.println(al);
    }
}
```

```

        System.out.println(al.isEmpty());           //checking isempty or not
        al.clear();                                //removes all elements
        System.out.println(al.isEmpty());
    }
}

```

observation: when we remove the data by passing numeric value it is by default treated as index value.

```

ArrayList al = new ArrayList();
al.add(10);
al.remove(10);          // java.lang.IndexOutOfBoundsException:

```

ArrayList constructors :

Constructor-1 public java.util.ArrayList();

The default capacity of the ArrayList is 10 once it reaches its maximum capacity then size is automatically increased by

$$\text{New capacity} = (\text{old capacity} * 3) / 2 + 1 = 16$$

Constructor-2 public java.util.ArrayList(int user-capacity); create ArrayList with initial capacity

```

ArrayList al = new ArrayList(20);
New capacity = (old capacity * 3) / 2 + 1 = 31

```

Constructor-3 public java.util.ArrayList(java.util.Collection)

Adding one collection data into another collection.

```

ArrayList a1 = new ArrayList();
a1.add(10);
ArrayList a2 = new ArrayList(a1);
a2.add(20);
System.out.println(a2); //10 20

```

Ex-6 There are two ways to add one collection data into another collection.

i. By using constructor approach.

ii. By using addAll() method

- ✓ To add only one collection data into another collection use constructor approach.
- ✓ To add more than one collection into single collection use addAll() method.

```
import java.util.ArrayList;
public class Test {
    public static void main(String[] args) {
        //constructor approach
        ArrayList<String> a1 = new ArrayList<String>();
        a1.add("ratan");
        ArrayList<String> a2 = new ArrayList<String>(a1);
        a2.add("durga");
        System.out.println(a2);

        //addAll() method to add the data
        ArrayList<String> b1 = new ArrayList<String>();
        b1.add("aaa");
        ArrayList<String> b2 = new ArrayList<String>();
        b2.add("bbb");
        ArrayList<String> b3 = new ArrayList<String>();
        b3.addAll(b1);
        b3.addAll(b2);
        b3.add("ccc");
        System.out.println(b3);
    }
}
```

Ex-7:

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Emp e1 = new Emp(111, "ratan");
        Emp e2 = new Emp(222, "Sravya");
        Emp e3 = new Emp(333, "aruna");
        Emp e4 = new Emp(444, "anu");

        ArrayList<Emp> a1 = new ArrayList<Emp>();
        a1.add(e1);
        a1.add(e2);

        ArrayList<Emp> a2 = new ArrayList<Emp>();
        a2.addAll(a1);
        a2.add(e3);
        a2.add(e4);

        System.out.println(a2.contains(e1));
        System.out.println(a2.containsAll(a1));
        a2.remove(e1);
        System.out.println(a2.contains(e1));
        System.out.println(a2.containsAll(a1));
    }
}
```

```

    //printing the data
    for (Emp e:a2)
    {
        System.out.println(e.eid+"---"+e.ename);
    }
}

```

Observation :

`a2.removeAll(a1);` // it removes all **a1** data except **a2**.
`a2.retainAll(a1);` // it removes all **a2** data except **a1**

ex 8: Adding similar elements n times

```

import java.util.ArrayList;
import java.util.Collections;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> a1 = new ArrayList<String>(Collections.nCopies(10, "ratan"));
        for (String s: a1)
        {
            System.out.println(s);
        }
    }
}

```

Ex 9: Creation of sub ArrayList & swapping data

Create sub ArrayList by using **subList(int,int)** method of ArrayList.

`public java.util.List<E> subList(int, int);`

To swap the data from one index position to another index position then use **swap()** method.

`public static void swap(java.util.List<?>, int, int);`

`package com.dss;`

```

import java.util.ArrayList;
import java.util.Collections;

public class Test10 {
    public static void main(String[] args) {
        ArrayList<String> a1 = new ArrayList<String>();
        a1.add("ratan");
        a1.add("anu");
        a1.add("durga");
        a1.add("sravya");

        System.out.println("Before swapping :" + a1);
        Collections.swap(a1, 1, 3);
        System.out.println("After swapping :" + a1);

        ArrayList<String> a2 = new ArrayList<String>(a1.subList(1, 4));
        a2.add("aaa");
        System.out.println(a2);
    }
}

```

Ex-10: conversion process

<i>Arrays to collection</i>	:	Arrays.asList()
<i>Normal version of collection to array</i>	:	toArray()
<i>Generic version of collection o array</i>	:	toArray(argument)

```

import java.util.*;
class ArrayListDemo
{
    public static void main(String[] args)
    {
        //Conversion of array to ArrayList by using asList() method
        String[] str={"ratan", "Sravya", "aruna"};
        ArrayList<String> a1 = new ArrayList<String>(Arrays.asList(str));
        a1.add("ratan");
        a1.add("anu");
        for (String s: a1)
        {
            System.out.println(s);
        }

        //Conversion of generic version of ArrayList to array by using toArray(T) method
        ArrayList<String> a2= new ArrayList<String>();
        a2.add("anu");
        a2.add("Sravya");
    }
}

```

```

String[] s = new String[al.size()];
a2.toArray(s);
for (String ss:s)
{
    System.out.println(ss);
}

//conversion of normal version ArrayList to Array by using toArray() method
ArrayList a3= new ArrayList();
a3.add(10);
a3.add('c');
a3.add("ratan");

Object[] o = a3.toArray();
for (Object oo :o)
{
    System.out.println(oo);
}
}
}

```

java.util.LinkedList: Before the *LinkedList* examples check the cursors.

```

public class java.util.LinkedList extends java.util.AbstractSequentialList
    implements java.util.List<E>, java.util.Deque<E>,
    java.lang.Cloneable,java.io.Serializable.

```

- 1) Introduced in 1.2 version.
- 2) Heterogeneous objects are allowed.
- 3) Null insertion is possible.
- 4) Insertion order is preserved.
- 5) Linked List methods are non-synchronized.
- 6) Duplicate objects are allowed.
- 7) The under laying data structure is double linkedlist.
- 8) cursors :- Iterator, ListIterator.

LinkedList constructors: To check the constructor use **javadoc** command or API documentation.

LinkedList();	it builds empty <i>LinkedList</i> .
LinkedList(java/util/Collection)	Used to add one collection data into another collection.

ex-1: LinkedList basic operations.

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        LinkedList<String> l=new LinkedList<String>();

```

```

l.add("B");
l.add("C");
l.add("D");
l.add("E");
l.addLast("Z");
l.addFirst("A");
l.add(1, "A1");
System.out.println("original content:-"+l);

l.removeFirst();
l.removeLast();
System.out.println("after deletion first & last:-"+l);

l.remove("E");
l.remove(2);
System.out.println("after deletion :-"+l);

l.set(2,"ratan");
System.out.println("after seting:-"+l);

System.out.println(l.isEmpty());
l.clear();
System.out.println(l.isEmpty());
}
}

```

Ex-2:

```

package com.dss;
public class Book {
    int id;
    String name,author;
    public Book(int id, String name, String author) {
        this.id = id;
        this.name = name;
        this.author = author;
    }
}

```

```

package com.dss;
import java.util.LinkedList;
import java.util.ListIterator;
public class Test2 {
    public static void main(String[] args) {
        LinkedList<Book> l = new LinkedList<Book>();
        l.add(new Book(111, "java", "ratan"));
        l.add(new Book(222, "c", "durga"));
        l.add(new Book(333, "cpp", "anu"));
    }
}

```

//By using listIterator Adding new Book object & remove : C

```

ListIterator<Book> lstr=l.listIterator();
lstr.add(new Book(444, "Python", "sravya"));
while(lstr.hasNext())
{
    Book b = lstr.next();
    if(b.name.equals("c"))
        lstr.remove();
}

//print the data using for-each loop
for(Book b : l)
{
    System.out.println(b.id+" "+b.author+" "+b.name);
}
}
}

```

ex-3: Adding one collection data into another Collection.

```

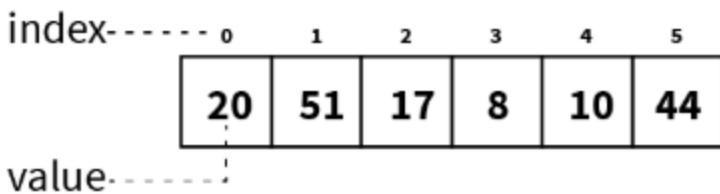
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> al = new ArrayList<String>();
        al.add("ratan");
        LinkedList<String> linked = new LinkedList<String>(al);
        linked.add("anu");
        System.out.println(linked);
    }
}

```

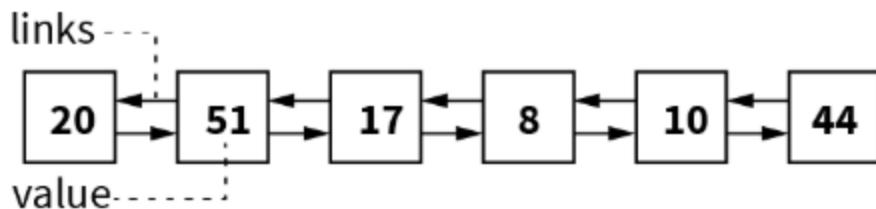
ArrayList vs. LinkedList:

- ✓ *ArrayList internally uses dynamic resizable array to store the elements
LinkedList internally uses double linkedlist to store the elements.*
- ✓ *Insertion, remove data on ArrayList is slow because when we perform the operations internally it requires more shift operations.
But insertion,remove on LinkedList is faster because shift operations are not required.*
- ✓ *With respect the memory ArrayList is recommended.
With respect the memory LinkedList is not recommended.*
- ✓ *ArrayList implements RandomAccess interface hence the data Access fast.
But LinkedList not implementing RandomAccess interface hence the data Access slow.*
- ✓ *ArrayList needs to be resized when runs out of space.
LinkedList nodes are allocated & released when necessary.*
- ✓ *RandomAccess is marker interface present in java.util package provides read operations capabilities.*

ArrayList



LinkedList



Vector: (legacy class introduced in 1.0 version)

```
public class java.util.Vector extends java.util.AbstractList
    implements java.util.List<E>,java.util.RandomAccess,
    java.lang.Cloneable,java.io.Serializable
```

- 1) Introduced in 1.0 version it is a legacy class.
- 2) Heterogeneous objects are allowed.
- 3) Duplicate objects are allowed.
- 4) Null insertion is possible.
- 5) Insertion order is preserved.
- 6) The underlying data structure is growable array.
- 7) Vector methods are synchronized.
- 8) Applicable cursors are Iterator, Enumeration, ListIterator.

Note : Vector is same as ArrayList but.

Vector methods are synchronized.

ArrayList methods are non-synchronized.

Constructor 1:- `public java.util.Vector();`

```
new Vector();
```

The default initial capacity of the Vector is 10

Once it reaches its maximum capacity it means when we trying to insert 11 element that capacity will become double[20 40etc].

Constructor 2: `public java.util.Vector(int user-capacity);`

```
Vector<String> v = new Vector<String>(2);
System.out.println(vv.capacity());           //2
v.add("aaa");
v.add("bbb");
System.out.println(v.capacity());           //4
```

Constructor 3: `public java.util.Vector(int user-capacity, int user-increment-value);`

```
Vector<String> v = new Vector<String>(2,5);
System.out.println(v.capacity());           //2
v.add("ratan");
v.add("aruna");
v.add("Sravya");
System.out.println(v.capacity());           //7
```

Constructor 4: `public java.util.Vector(java.util.Collection):Adding one collection data into another`

```
ArrayList<String> al = new ArrayList<String>();
al.add('no1');
Vector<String> v = new Vector<String>(al);
v.add("ratan");
System.out.println(v);
```

Ex-1:

```
package vectorEx;

import java.util.Iterator;
import java.util.Vector;

public class Test2 {

    public static void main(String[] args) {
        Vector<Integer> v = new Vector<Integer>();
        for(int i=1;i<=30;i++)
        {
            v.add(i);
        }
        System.out.println(v); //1-30

        //remove the odd elements by using Iterator cursor
        Iterator<Integer> itr = v.iterator();
        while(itr.hasNext())
        {
            Integer i = itr.next();
            if(i%2==0)
                System.out.println(i); // even printed
        }
    }
}
```

```

        else
            itr.remove();
    }
    System.out.println(v); // even printed
}
}

```

Ex-2 : Copying data from Vector to ArrayList:- use copy() method of Collections class.

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> al = new ArrayList<String>();
        al.add("ratan");
        al.add("anu");

        Vector<String> v = new Vector<String>();
        v.add("durga");
        v.add("sravya");
        Collections.copy(al,v);
        System.out.println(al);
    }
}

```

ex-3:

```

package vectorEx;
public class Product {
    int id;
    String name;
    int cost;
    public Product(int id, String name, int cost) {
        this.id = id;
        this.name = name;
        this.cost = cost;
    }
}

```

```

package vectorEx;

import java.util.Enumeration;
import java.util.Iterator;
import java.util.ListIterator;
import java.util.Vector;
public class Test3 {
    public static void main(String[] args) {

```

```

Vector<Product> products = new Vector<Product>();
products.add(new Product(111, "pen", 20));
products.add(new Product(222, "fan", 300));
products.add(new Product(333, "light", 200));

//print the data using Enumeration
Enumeration<Product> e = products.elements();
while(e.hasMoreElements())
{
    Product p = e.nextElement();
    System.out.println(p.id+" "+p.name+" "+p.cost);
}

//print the data by using iterator
Iterator<Product> itr = products.iterator();
while(itr.hasNext())
{
    Product p = itr.next();
    System.out.println(p.id+" "+p.name+" "+p.cost);
}

//print the data by using ListIterator
ListIterator<Product> lstr = products.listIterator();
while(lstr.hasNext())
{
    Product p = lstr.next();
    System.out.println(p.id+" "+p.name+" "+p.cost);
}
}
}

```

Stack: (legacy class introduced in 1.0 version)

- 1) It is a child class of vector.
- 2) Introduce in 1.0 version, it is a legacy class.
- 3) It is designed for LIFO(last in fist order).

ex-1 :

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Stack<String> s = new Stack<String>();
        s.push("ratan");
        s.push("anu");
        s.push("durga");
        System.out.println(s);
        System.out.println(s.size());

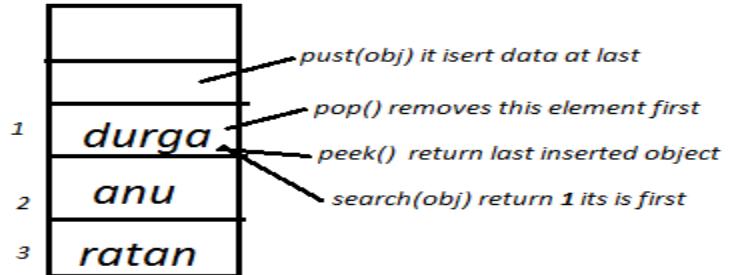
        System.out.println(s.peek());           //to return last element of the Stack
        s.pop();                            //remove the data top of the stack
        System.out.println(s);
        System.out.println(s.search("durga")); //1 last added object will become first
    }
}

```

```

        System.out.println(s.isEmpty());
        s.clear();
        System.out.println(s.isEmpty());
    }
}

```



Ex-2: conversion of arrays to stack.

```

package com.dss;
import java.util.Stack;
public class Test {
    public static void main(String[] args) {
        String[] s = {"ratan", "anu", "durga"};
        Stack<String> stack = new Stack<String>();
        for(String str : s)
        {
            stack.push(str);
        }
        System.out.println(stack);
    }
}

```

Collection framework Cursors:

There are three ways to read the data from collection classes

1. By using for-each loop
2. By using get() method
3. By using cursor

There are three types of cursors in java

- a. Enumeration
- b. Iterator
- c. listIterator

Property	Enumeration	Iterator	ListIterator
Purpose	Read the data	Read the data	Read the data
Is legacy?	Yes 1.0	No 1.2 version	No 1.2 version
It is applicable for	Only legacy classes	For all classes	Only for list classes
Universal or not	No	Yes	no
How to get the object	Using elements() method	Using iterator() method	Using listIterator() method
Navigation	Only forward	Only forward	Bi-directional
Methods	hasMoreElements(); nextElement();	hasNext() next(), remove()	9-methods
Operations	Only read	Read & remove	Read & remove &update & add
Class or interface	Interface	Interface	Interface
Normal & generic type	Supports both	Support both	Support both

ListIterator methods:

```
public abstract boolean hasNext();
public abstract E next();
public abstract boolean hasPrevious();
public abstract E previous();
public abstract int nextIndex();
public abstract int previousIndex();
public abstract void remove();
public abstract void set(E); //replacement
public abstract void add(E);
```

ex-1: Enumeration Cursor

```
package com.dss;
```

```

import java.util.Enumeration;
import java.util.Vector;
public class Test1 {
    public static void main(String[] args) {
        Vector<String> v = new Vector<String>();
        v.add("ratan");
        v.add("anu");
        v.add("durga");
        //Read the data by using Enumeration :normal version : type Casting Required
        Enumeration e = v.elements();
        while(e.hasMoreElements())
        {
            String s = (String)e.nextElement();
            System.out.println(s);
        }
        //Read the data by using Enumeration : generic version : type Casting not Required
        Enumeration<String> e1 = v.elements();
        while(e1.hasMoreElements())
        {
            String s = e1.nextElement();
            System.out.println(s);
        }
    }
}

```

Ex-2: Iterator Cursor

```

package com.dss;
import java.util.ArrayList;
import java.util.Iterator;
public class Test2 {
    public static void main(String[] args) {
        ArrayList<String> al = new ArrayList<String>();
        al.add("ratan");
        al.add("anu");
        al.add("durga");
        //Read the data by using iterator : normal version : Type casting required
        Iterator itr = al.iterator();
        while(itr.hasNext())
        {
            String s = (String)itr.next();
            System.out.println(s);
        }
        //read the data by using iterator : generic version : Type Casting not required
        Iterator<String> itr1 = al.iterator();
        while(itr1.hasNext())
        {
            String s = itr1.next();
            System.out.println(s);
        }
    }
}

```

Ex-3: ListIterator Cursor.

```
package com.dss;
```

```

import java.util.ArrayList;
import java.util.ListIterator;

public class Test3 {
    public static void main(String[] args) {
        ArrayList<String> al = new ArrayList<String>();
        al.add("ratan");
        al.add("anu");
        al.add("durga");

        //ListIterator cursor : with generic version : print the data forward & backward direction
        ListIterator<String> lstr2 = al.listIterator();
        while(lstr2.hasNext())
        {
            String s = lstr2.next();
            System.out.println(s);
        }
        while(lstr2.hasPrevious())
        {
            String s = lstr2.previous();
            System.out.println(s);
        }
    }
}

```

ex-4 : Reading the data using (for-each , get() , cursor)

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> al =new ArrayList<String>();
        al.add("ratan");
        al.add("anu");
        al.add("sravya");

        //1st approach to read the data using for-each loop
        for (String a : al)
        {
            System.out.println(a);
        }
        //2nd approach to print Collection data using get () method
        String s = al.get(0);
        System.out.println(s);
        // 3rd approach : Generic version of cursor
        Iterator<String> itr2 = al.iterator();
        while (itr2.hasNext())
        {
            String str =itr2.next();
            System.out.println(str);
        }
    }
}

```

Ex 5: Iterator cursor removes data.

```

package com.dss;
import java.util.ArrayList;
import java.util.Iterator;
public class Test {
    public static void main(String[] args) {
        ArrayList<String> al = new ArrayList<String>();
        al.add("ratan");
        al.add("anu");
        al.add("sravya");

        Iterator<String> itr = al.iterator();
        while(itr.hasNext())
        {
            String s1 = itr.next();
            if(s1.equals("ratan"))
                itr.remove();
        }
        System.out.println(al);
    }
}

```

ex -6:

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> al = new ArrayList<String>();
        al.add("ratan");
        al.add("anu");
        al.add("sravya");

        ListIterator<String> lstr = al.listIterator();
        lstr.add("suneel");
        while(lstr.hasNext())
        {
            String s = lstr.next();
            if (s.equals("anu"))
            {
                lstr.set("Anushka");
            }
            if (s.equals("ratan"))
            {
                lstr.remove();
            }
        }
        System.out.println(lstr);
    }
}

```

- ✓ Sunnel object is added at first position because when we create the iterator cursor the cursor is pointing to before first record.

Ex- 7: Adding Book objects into ArrayList then removing the data using Iterator cursor.

Book.java

```
package com.dss;
public class Book {
    int id;
    String name;
    String author;
    public Book(int id, String name, String author) {
        this.id = id;
        this.name = name;
        this.author = author;
    }
}
```

Test.java

```
package com.dss;

import java.util.ArrayList;
import java.util.Iterator;

public class Test5 {

    public static void main(String[] args) {
        ArrayList<Book> books = new ArrayList<Book>();
        books.add(new Book(111, "java", "ratan"));
        books.add(new Book(222, "C", "anu"));
        books.add(new Book(333, "CPP", "durga"));

        Iterator<Book> itr = books.iterator();
        while(itr.hasNext())
        {
            Book b = itr.next();
            if(b.id==111)
                itr.remove();
            if(b.name.equals("CPP"))
                itr.remove();
        }

        //print the remaining data
        for(Book b:books)
        {
            System.out.println(b.id+" "+b.name+" "+b.author);
        }
    }
}
```

Collection Data Sorting

- ✓ It is possible to sort the collection data by using `sort()` method of `Collections` class and by default it perform ascending order.
- ✓ if we want to perform sorting data must satisfies the below conditions,
 - The data must be homogenous
 - Must implements Comparable interface.
- ✓ In java String, all wrapper classes are implementing comparable interface by default.
- ✓ To perform sorting internally JVM uses `compareTo()` method
 - if you want perform sorting of two diff object JVM will generate `ClassCastException`.
 - If you are perform sorting with null obj JVM will generate `NullPointerException`.

```
import java.util.*;  
class Test  
{    public static void main(String[] args)  
    {        ArrayList<String> al = new ArrayList<String>();  
        al.add("ratan");  
        al.add("anu");
```

```

        al.add("Sravya");
        System.out.println("ArrayList data before sorting="+al);
        Collections.sort(al);
        System.out.println("ArrayList data after sorting ascending order="+al);

        LinkedList<Integer> l = new LinkedList<Integer>();
        l.add(10);
        l.add(3);
        l.add(20);
        l.add(5);
        System.out.println("Before sorting: "+l);
        Collections.sort(l);
        System.out.println("After sorting: "+l);

    }
}

```

Case 1: if we are trying to perform sorting of heterogeneous data , while performing comparison(by using compareTo() method) JVM will generate **java.lang.ClassCastException**

```

ArrayList al = new ArrayList();
al.add("ratan");
al.add(10);
Collections.sort(al);

```

Case 2: when we perform sorting of data if the data contains null value while performing comparison (by using compareTo()) JVM will generate **java.lang.NullPointerException**.

```

ArrayList al = new ArrayList();
al.add("ratan");
al.add(null);
Collections.sort(al);

```

Ex:

If we want to perform descending order use `Collections.reverseOrder()` method along with `Collection.sort()` method.

```
Collections.sort(list , Collections.reverseOrder());
```

```
package com.dss;
```

```
import java.util.ArrayList;
import java.util.Collections;
```

```
public class Test1 {
    public static void main(String[] args) {
        ArrayList<String> al = new ArrayList<String>();
        al.add("ratan");
        al.add("anu");
        al.add("durga");
        al.add("sravya");
        System.out.println("Before sorting:"+al);
        Collections.sort(al,Collections.reverseOrder());
    }
}
```

```

        System.out.println("After sorting:"+al);
    }
}

```

Java.lang.Comparable :

- ✓ If we want to sort user defined class like Emp based on eid or ename with default natural sorting order then your class must implements Comparable interface.
- ✓ Comparable interface present in java.lang package it contains only one method compareTo(obj) then must override that method to write the sorting logics.
- ✓ If your class is implementing Comparable interface then that objects are sorted automatically by using **Collections.sort()**. And the objects are sorted by using compareTo() method of that class.

"ratan".compareTo("anu")	==> +ve	==> change the order
"ratan".compareTo("ratan")	==> 0	==> no change
"anu".compareTo("ratan")	==> -ve	==> no change

Ex :

Emp.java: Normal version of comparable performing sorting of eid

```

class Emp implements Comparable
{
    int eid;
    String ename;
    Emp(int eid,String ename)
    {
        this.eid=eid;
        this.ename=ename;
    }
    public int compareTo(Object o)
    {
        Emp e = (Emp)o;
        if(eid == e.eid)      return 0;
        else if(eid > e.eid) return 1;
        else                  return -1;
    }
}

```

Emp.java : Generic version of Comparable performing sorting of ename

```

class Emp implements Comparable<Emp>
{
    int eid;
    String ename;
    Emp(int eid, String ename)
    {
        this.eid=eid;
        this.ename=ename;
    }
    public int compareTo(Emp e)
    {
        return ename.compareTo(e.ename);
    }
}

```

Test.java: use any one Emp.java file to perform sorting

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<Emp> al = new ArrayList<Emp>();
        al.add(new Emp(111, "ratan"));
        al.add(new Emp(444, "anu"));
        al.add(new Emp(333, "durga"));
        al.add(new Emp(222, "sravya"));

        Collections.sort(al);

        for(Emp e : al)
        {
            System.out.println(e.eid+" "+e.ename);
        }
    }
}

```

Java.lang.Comparable vs. java.util.Comparator:

Property

1. Sorting logics

6. which type of sorting

2. Properties sorting

4. Method calling to
perform sorting

3. Sorting method

5. package

Java.lang.Comparable

Here we are mixing both business logics & sorting logics in single class.

Default natural sorting order & customized also.

Only one property sorting

Int compareTo(Object o1)

This method compares this object with o1 object and returns a integer. Its value has following meaning

positive – this object is greater than o1
zero this object equals to o1
negative this object is less than o1

Collections.sort(List)

Java.lang

Comparator sorting:

Emp.java:

```
class Emp
{
    int eid;
    String ename;
    Emp(int eid, String ename)
    {
        this.eid = eid;
        this.ename = ename;
    }
}
```

EidComp.java:- normal version of comparator : type casting required

```
import java.util.Comparator;
public class EidComp implements Comparator{
    public int compare(Object o1, Object o2) {
        Emp e1 = (Emp)o1;
        Emp e2 = (Emp)o2;
        if(e1.eid == e2.eid)      return 0;
```

integer. Its value has following meaning.

positive – o1 is greater than o2

zero – o1 equals to o2

negative – o1 is less than o1

Collections.sort(List, Compar)

Java.util

only For customized sorting order.

Java.util.Comparator

We can separate the sorting logics & normal logics in different classes.

Multiple properties sorting

int compare(Obj o1, Obj o2)

This method compares o1 and o2 objects. and returns a

```

        else if(e1.eid>e2.eid)    return 1;
        else                      return -1;
    }
}

```

EnameComp.java:- generic version of comparator : type casting not required

```

import java.util.Comparator;
class EnameComp implements Comparator<Emp>
{
    public int compare(Emp e1,Emp e2)
    {
        return (e1.ename).compareTo(e2.ename);
    }
}

```

Test.java:-

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<Emp> al = new ArrayList<Emp>();
        al.add(new Emp(333,"ratan"));
        al.add(new Emp(222,"anu"));
        al.add(new Emp(111,"Sravya"));
        Collections.sort(al,new EidComp());
        Iterator<Emp> itr = al.iterator();
        while (itr.hasNext())
        {
            Emp e = itr.next();
            System.out.println(e.eid+"---"+e.ename);
        }
    }
}

```

Sorting of emp name : `Collections.sort(al,new EnameComp());`

Sorting of emp id : `Collections.sort(al,new EidComp());`

Collection data Cloning & serialization process

In Collection frame work every class implements Cloneable & Serializable interfaces to support cloning process & serialization process.

ex1:

```

package com.dss;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.RandomAccess;

public class Test {
    public static void main(String[] args) {
        ArrayList<String> al = new ArrayList<String>();

```

```

        System.out.println(al instanceof List);
        System.out.println(al instanceof Cloneable);
        System.out.println(al instanceof Serializable);
        System.out.println(al instanceof RandomAccess);

        LinkedList<String> l = new LinkedList<String>();
        System.out.println(l instanceof List);
        System.out.println(l instanceof Cloneable);
        System.out.println(l instanceof Serializable);
        System.out.println(l instanceof RandomAccess);           //false
    }
}

```

ex2: Cloning process : The process of creating exactly duplicate objects is called cloning.

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        LinkedList<String> linked= new LinkedList<String>();
        linked.add("First");
        linked.add("Second");
        linked.add("Third");
        linked.add("Random");
        System.out.println("Actual LinkedList:"+linked);

        LinkedList<String> copy = (LinkedList) linked.clone();
        System.out.println("Cloned LinkedList:"+copy);
    }
}
E:\>java Test
Actual LinkedList:[First, Second, Third, Random]
Cloned LinkedList:[First, Second, Third, Random]

```

ex-3 : Serialization process : To perform serialization the class must implements Serializable interface.

```

package com.dss;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
public class Test {
    public static void main(String[] args) throws IOException, ClassNotFoundException {
        ArrayList<Emp> al = new ArrayList<Emp>();
        al.add(new Emp(111, "ratan"));
        al.add(new Emp(222, "anu"));

        //Serialization process
        FileOutputStream outputStream = new FileOutputStream("abc.txt");

```

```

ObjectOutputStream objectOutputStream = new ObjectOutputStream(outputStream);
objectOutputStream.writeObject(al);
outputStream.close();
objectOutputStream.close();
System.out.println("serialization process completed..... ");

//Deserialization process
FileInputStream inputStream = new FileInputStream("abc.txt");
ObjectInputStream objectInputStream = new ObjectInputStream(inputStream);
ArrayList<Emp> arraylist = (ArrayList<Emp>)objectInputStream.readObject();
outputStream.close();
objectInputStream.close();
System.out.println("Deserialization process completed..... ");
for(Emp e:arraylist)
{
    System.out.println(e.eid+"---"+e.ename);
}
}
}

```

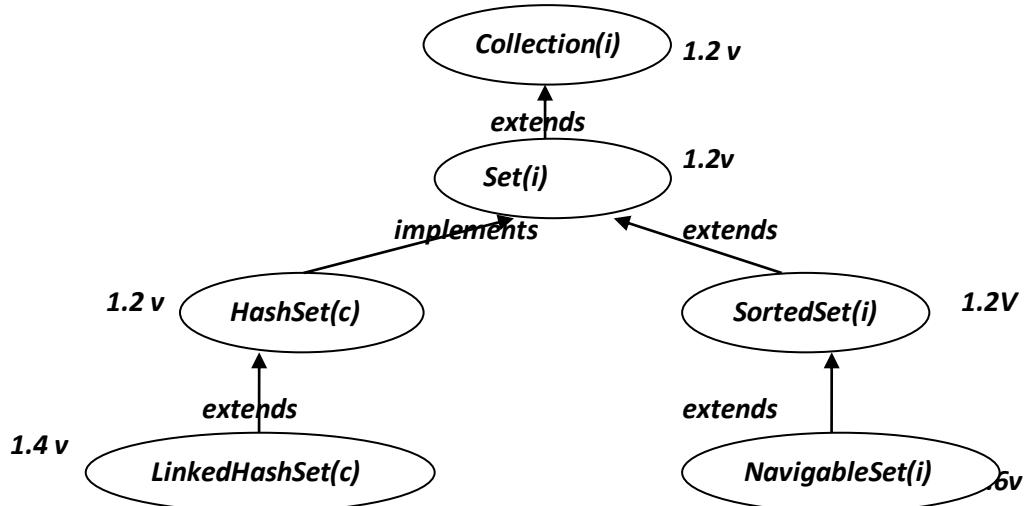
Observation :

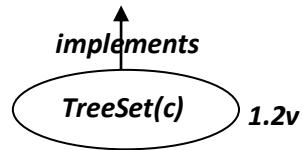
```

class Emp implements Serializable {      }
class Student {   }
class Test
{
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        al.add(new Emp());
        al.add(new Student());
    }
}

```

- ✓ To perform serialization of ArrayList inside the arraylist all objects must be serializable objects.
- ✓ In above example serialization of ArrayList data is not possible because the Student obj is not implements serializable interface.

Set interface implementation classes



List vs. Set : List allows duplicates & set duplicates not allowed.

Java.util.HashSet:

```
public class java.util.HashSet extends java.util.AbstractSet
    implements java.util.Set<E>, java.lang.Cloneable, java.io.Serializable
```

- 1) Introduced in 1.2 version.
- 2) Heterogeneous objects are allowed.
- 3) Duplicate objects are not allowed if we are trying to insert duplicate values then we won't get any compilation & Execution errors simply add method returns false .
- 4) Null insertion is possible but if we are inserting more than one null it return only one null value (because duplicates are not allowed).
- 5) The under laying data structure is HashTable.
- 6) Insertion order is not preserved it is based on the hash code of the object (hashing mechanism).
- 7) Methods are non-synchronized.
- 8) It supports only Iterator cursor to retrieve the data.

Constructors:

1. public HashSet(); it creates default HashSet.

```
new HashSet();
default capacity: 16 default fill ratio : 0.75
```

2. Public HashSet(int user-capacity);

```
New HashSet(10);
It create the HashSet by specified capacity. Default fill ratio 0.75
```

3. Public HashSet(int capacity, float fillRatio);

```
new HashSet(10,0.56);
It initialize both capacity and fillratio(also called as load factor).
```

4. Public HashSet(java/util/Collection); Adding one collection data into another collection data.

```
HashSet<String> h1 = new HashSet<String>();
h1.add("ratan");
```

```

HashSet<String> h2 = new HashSet<String>(h1);
h2.add("no1");
System.out.println(h2);

ex: import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        HashSet<String> h = new HashSet<String>();
        h.add("ratan");
        h.add("anu");
        h.add("durga");
        h.add("ratan");
        Iterator<String> itr = h.iterator();
        while (itr.hasNext())
        {
            String str = itr.next();
            System.out.println(str);
        }
    }
}

```

ex: when we insert duplicates it insert the first value & ignores the next occurrences.

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        HashSet<String> h = new HashSet<String>();
        System.out.println(h.add("ratan")); //true
        System.out.println(h.add("ratan")); //false
        System.out.println(h.add("ratan")); //false
        System.out.println(h);
    }
}

```

Java.util.LinkedHashSet:

```

public class java.util.LinkedHashSet      extends java.util.HashSet
                                         implements  java.util.Set<E>,
                                         java.lang.Cloneable,java.io.Serializable

```

1. *Introduced in 1.4 version and It is a child class of HashSet.*
2. *Heterogeneous objects are allowed.*
3. *Duplicate objects are not allowed if we are trying to insert duplicate values then we won't get any compilation & Execution errors simply add method return false.*
4. *Insertion order is preserved.*
5. *Null insertion is possible only once(because duplication is not possible).*
6. *The under laying data structure is LinkedList & hashTable.*
7. *Methods are non-synchronized.*
8. *It supports only Iterator cursor retrieve the data.*

Note : HashSet not preserved insertion order but LinkedHashSet preserved insertion order.

Constructors:-

```
Public LinkedHashSet();
Public LinkedHashSet(java.util.Collection<? extends E>);
Public LinkedHashSet(int capacity);
Public LinkedHashSet(int capacity, float fillRatio);
```

ex : Eliminating duplicates by using Set interface.

```
ArrayList<String> al = new ArrayList<String>();
al.add("anu");
al.add("anu");

LinkedHashSet<String> lh = new LinkedHashSet<String>(al);
lh.add("aaa");
System.out.println(lh);
```

ex: Set data sorting.

```
Set<Integer> integers = new LinkedHashSet<>();
integers.add(5);
integers.add(10);
integers.add(0);
integers.add(-1);
System.out.println("Original set: " + integers);
// Collections.sort(integers); This throws error since sort method accepts list not collection
List list = new ArrayList(integers);
Collections.sort(list);
System.out.println("Sorted set: " + list);
Collections.sort(list, Collections.reverseOrder());
System.out.println("Reversed set: " + list);
```

ex : inserting Emp data into LinkedHashSet & removing data using Iterator cursor.**Emp.java :**

```
package com.dss;
public class Emp {
    int eid;
    String ename;
    public Emp(int eid, String ename) {
        this.eid = eid;
        this.ename = ename;
    }
}
```

Test.java:

```
package com.dss;

import java.util.Iterator;
```

```

import java.util.LinkedHashSet;

public class Test {
    public static void main(String[] args) {
        LinkedHashSet<Emp> h = new LinkedHashSet<Emp>();
        h.add(new Emp(111, "ratan"));
        h.add(new Emp(222, "anu"));
        h.add(new Emp(333, "durga"));

        Iterator<Emp> itr = h.iterator();
        while(itr.hasNext())
        {
            Emp e = itr.next();
            if(e.eid==111)
                itr.remove();
            if(e.ename.equals("durga"))
                itr.remove();
        }

        for(Emp e : h)
        {
            System.out.println(e.eid+" "+e.ename);
        }
    }
}

```

Java.util.TreeSet:

<i>public class java.util.TreeSet</i>	extends	<i>java.util.AbstractSet<E></i>
	implements	<i>java.util.NavigableSet<E>, java.lang.Cloneable, java.io.Serializable</i>

1. *TreeSet introduced in 1.2 version.*
2. *Heterogeneous data is not allowed.*
3. *Insertion order is not preserved but it sorts the elements in some sorting order.*
4. *Duplicate objects are not allowed.*
5. *Null insertion is possible only once.*
6. *TreeSet Methods are non-synchronized.*
7. *The underlying data Structure is Balanced Tree.*
8. *It supports Iterator cursor to retrieve the data.*

TreeSet Constructors:

TreeSet() Constructs a new, empty tree set, sorted according to the natural ordering of its elements.

TreeSet(Collection<? extends E> c)

Constructs a new tree set containing the elements in the specified collection, sorted according to the natural ordering of its elements.

TreeSet(Comparator<? super E> comparator)

Constructs a new, empty tree set, sorted according to the specified comparator.

TreeSet(SortedSet<E> s)

Constructs a new tree set containing the same elements and using the same ordering as the specified sorted set.

Ex-1:

```
package com.dss;
import java.util.TreeSet;

public class Test1 {
    public static void main(String[] args) {
        TreeSet<String> t = new TreeSet<String>();
        t.add("ratan");
        t.add("anu");
        t.add("durga");
        System.out.println(t);

        TreeSet<Integer> t1 = new TreeSet<Integer>();
        t1.add(10);
        t1.add(2);
        t1.add(3);
        System.out.println(t1);
    }
}
```

- ✓ When we insert the data in TreeSet, by default it prints the data in sorting order(ascending or alphabetical order) because it is implementing SortedSet interface.
- ✓ To perform the sorting internally it uses compareTo() method and it compare the two objects it returns int value as a return value.

Case 1: `TreeSet t=new TreeSet();
t.add("anu");
t.add(10);
System.out.println(t);`

TreeSet allows homogeneous data, if we are trying to insert heterogeneous data while performing sorting by using compareTo() JVM will generate **java.lang.ClassCastException** (because it is not possible to compare integer data with String).

Case 2: `TreeSet t=new TreeSet();
t.add("ratan");
t.add(null);
System.out.println(t);`

- If the TreeSet contains data if we are trying to insert null value at the time of comparison JVM will generate `//java.lang.NullPointerException`.
- In java any object with comparison of null it will generate `java.lang.NullPointerException`.

Ex-2: constructor-4 example : `TreeSet(SortedSet<E> s)`

Constructs a new tree set containing the same elements and using the same ordering as the specified sorted set.

```
package com.dss;
import java.util.SortedSet;
import java.util.TreeSet;
public class Test4 {
    public static void main(String[] args) {
        TreeSet<Integer> t = new TreeSet<Integer>();
        for(int i=1;i<=10;i++)
        {
            t.add(i);
        }
        System.out.println(t);

        SortedSet<Integer> s1 = t.subSet(3, 7);
        TreeSet<Integer> t1 = new TreeSet<Integer>(s1);
        System.out.println(t1);

        SortedSet<Integer> s2 = t.tailSet(4);
        TreeSet<Integer> t2 = new TreeSet<Integer>(s2);
        System.out.println(t2);

        SortedSet<Integer> s3 = t.headSet(6);
        TreeSet<Integer> t3 = new TreeSet<Integer>(s3);
        System.out.println(t3);
    }
}
```

Ex-3: constructor-3 example

`TreeSet(Comparator<? super E> comparator)`

Constructs a new, empty tree set, sorted according to the specified comparator.

```
package com.dss;

import java.util.Comparator;
import java.util.TreeSet;

public class Test2 {
    public static void main(String[] args) {
        TreeSet<String> t = new TreeSet<String>(new MyComp());
        t.add("ratan");
        t.add("anu");
        t.add("durga");
        t.add("sravya");
        System.out.println(t);
    }
}
```

```

TreeSet<Integer> t1 = new TreeSet<Integer>(new MyComp1());
t1.add(10);
t1.add(3);
t1.add(7);
t1.add(20);
System.out.println(t1);
}

class MyComp implements Comparator<String>
{
    @Override
    public int compare(String s1, String s2) {
        return -s1.compareTo(s2);
    }
}

class MyComp1 implements Comparator<Integer>
{
    @Override
    public int compare(Integer i1, Integer i2) {
        return -i1.compareTo(i2);
    }
}

```

Check all possibilities :

```

return i1.compareTo(i2);
return -i1.compareTo(i2);
return i2.compareTo(i1);
return -i2.compareTo(i1);

```

Ex-4: constructor-3 example

TreeSet(Comparator<? super E> comparator)

Constructs a new, empty tree set, sorted according to the specified comparator.

Product.java

```

package com.dss;
public class Product {
    int id;
    String name;
    double cost;
    public Product(int id, String name, double cost) {
        this.id = id;
        this.name = name;
        this.cost = cost;
    }
}

```

Test.java

```

package com.dss;
import java.util.Comparator;
import java.util.TreeSet;

public class Test {
    public static void main(String[] args) {
        TreeSet<Product> t1 = new TreeSet<Product>(new MyCom2());
        t1.add(new Product(111, "chair", 1000.56));
        t1.add(new Product(333, "pen", 100.56));
        t1.add(new Product(222, "bottle", 10000.56));
        t1.add(new Product(444, "phone", 2000.56));

        for(Product p : t1)
        {
            System.out.println(p.id+" "+p.name+" "+p.cost);
        }
    }

    class MyCom2 implements Comparator<Product>
    {
        @Override
        public int compare(Product p1, Product p2) {
            return -p1.name.compareTo(p2.name);
        }
    }
}

```

ex -5 program to insert StringBuffer data into TreeSet to perform sorting in alphabetical order.

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        TreeSet<StringBuffer> t = new TreeSet<StringBuffer>(new MyComp());
        t.add(new StringBuffer("ccc"));
        t.add(new StringBuffer("aaa"));
        t.add(new StringBuffer("bbb"));
        System.out.println(t);
    }

    class MyComp implements Comparator<StringBuffer>
    {
        public int compare(StringBuffer sb1,StringBuffer sb2)
        {
            String s1 = sb1.toString();
            String s2 = sb2.toString();
            return -s1.compareTo(s2);
        }
    }
}

```

```

        }
    }
}
```

Ex -6 : write a program to insert String & StringBuffer object into TreeSet perform sorting.

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        TreeSet t = new TreeSet(new MyComp());
        t.add("ratan");
        t.add(new StringBuffer("sravya"));
        t.add("anu");
        t.add(new StringBuffer("suneelbabu"));
        t.add("sri");
        System.out.println(t);
    }
}
class MyComp implements Comparator
{
    public int compare(Object o1, Object o2)
    {
        String s1 = o1.toString();
        String s2 = o2.toString();
        return -s1.compareTo(s2);
    }
}
```

Ex-7 : Basic operations on TreeSet.

```

public E lower(E);
public E higher(E);
public java.util.SortedSet<E> subSet(E, E);
public java.util.SortedSet<E> headSet(E);
public java.util.SortedSet<E> tailSet(E);
public E pollFirst();
public E pollLast();
```

it print lower object of specified object
 it print higher object of specified object
 it print subset
 it print specified object above objects
 it print specified objects below values
 it print and remove first
 it print and remove last.

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        TreeSet<Integer> t = new TreeSet<Integer>();
        t.add(50);
        t.add(20);
```

```

t.add(40);
t.add(10);
t.add(30);
System.out.println(t);                                //10 20 30 40 50

System.out.println(t.headSet(30));                     //[10,20]
System.out.println(t.tailSet(30));                     //[30,40,50]
System.out.println(t.subSet(20,50));                  //[20,30,40]

System.out.println("last element="+t.last());          //50
System.out.println("first element="+t.first());         //10

System.out.println("lower element="+t.lower(50));       //40
System.out.println("higher element="+t.higher(20));    //30

System.out.println("print & remove first element="+t.pollFirst()); //10
System.out.println("print & remove last element="+t.pollLast());  //50

System.out.println("final elements="+t);                //20 30 40
System.out.println(t.remove(30));                      //20 40
System.out.println("final elements="+t);

}

}

```

Queue interface

Queue follows FIFO(first in first out)

A Queue is a collection for holding elements prior to processing.

Each Queue method exists in two forms:

- ✓ *one throws an exception if the operation fails, and*
- ✓ *The other returns a special value if the operation fails (either null or false, depending on the operation).*

Queue Interface Structure

Type of Operation	Throws exception	Returns special value
Insert	<code>add(e)</code>	<code>offer(e)</code>
Remove	<code>remove()</code>	<code>poll()</code>
Examine	<code>element()</code>	<code>peek()</code>

Example :- in priorityQueue insertion order is not preserved.

`public abstract E remove(); // it removes the data`
`public abstract E poll();`

It is used to retrieves and removes the head of this queue, or returns null if this queue is empty.

`public abstract E element(); // It is used to retrieves, but not remove, the head of this queue.`

`public abstract E peek();`

It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

```
import java.util.Iterator;
import java.util.PriorityQueue;
public class Test {
    public static void main(String[] args) {
        PriorityQueue<String> pq = new PriorityQueue<String>();
        pq.add("ratan");
        pq.add("anu");
        pq.add("durga");
        pq.add("sunny");
        pq.add("xxx");
        System.out.println(pq);

        System.out.println(pq.peek());
        System.out.println(pq.poll());
        System.out.println(pq);

        pq.remove();
        pq.remove("xxx");
        System.out.println(pq);

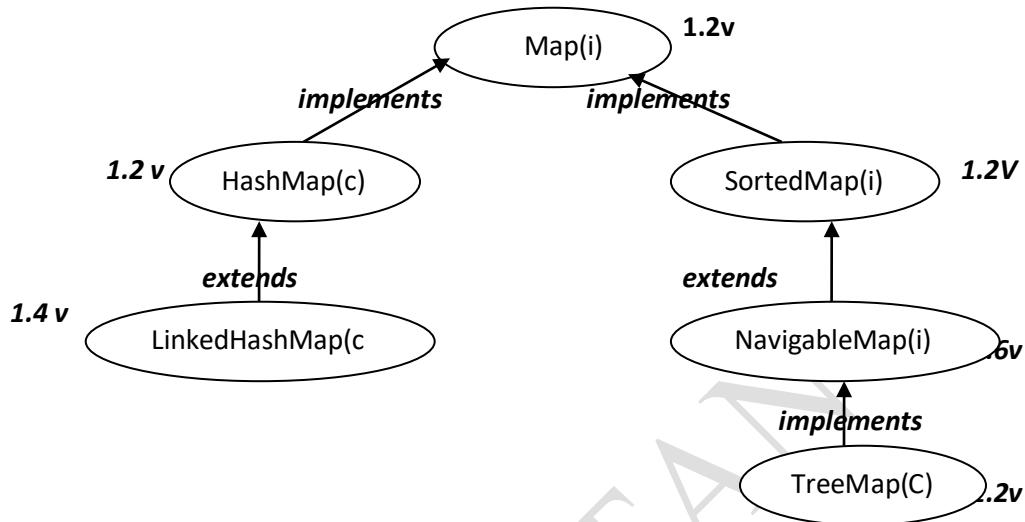
        Iterator<String> itr = pq.iterator();
        while(itr.hasNext())
        {
            String s = itr.next();
            System.out.println(s);
        }
    }
}
```

Assignment : create the class Book with fields : id, name , author, quantity

Create the PriorityQueue add 3-book objects print the data

Note : if we are adding the data in priorityQueue the data must be homogeneous & must implements comparable interface.

Map interface:-



- ✓ Map is used to store two objects at a time in the form of key value pairs. Here the key is object & value is object.
- ✓ The key value pair is known as entry, the map contains group of entries.
- ✓ In map the keys must be unique but values we can duplicate.

Java.util.HashMap:-

```

public class java.util.HashMap extends java.util.AbstractMap
    implements java.util.Map
        java.lang.Cloneable,java.io.Serializable
  
```

- 1) introduced in 1.2 version.
- 2) Heterogeneous data allowed.
- 3) Underlying data Structure is HashTable.
- 4) Duplicate keys are not allowed but values can be duplicated.
- 5) Insertion order is not preserved it is based on hashCode.
- 6) Null is allowed for key(only once) and allows for values any number of times.
- 7) Every method is non-synchronized.

Constructors:-

`HashMap();` it creates default HashMap.

`HashMap(java/util/Mapvar);` Adding one map data into another .

HashMap(int capacity); It creates the hashmap with specified capacity but the default capacity is 16.

HashMap(int capacity, float fillRatio);

It creates the hashMap with specified capacity & fillRatio.(default capacity is 16 & default fill ratio 0.75)

Entry:-

- ✓ The each and every key value pair is called **Entry**. The Map contains group of entries.
- ✓ Entry is sub interface of Map interface hence get the entry interface by using Map interface.

```
interface Map
{
    interface Entry
    {
        public abstract Object getKey();
        public abstract Object getValue();
        public abstract Object setValue();
    }
}
```

- ✓ To get all the keys use keyset() method.

```
public java.util.Set<K> keySet();
```

- ✓ To get all the values use values() method.

```
public java.util.Collection<V> values();
```

- ✓ To get all the entries use entrySet() method.

```
public java.util.Set<java.util.Map$Entry<K, V>> entrySet();
```

Example :

```
package com.dss;
import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map.Entry;
import java.util.Set;
public class Test {
    public static void main(String[] args) {
        HashMap<Integer, String> h = new HashMap<Integer, String>();
        h.put(111, "ratan");
        h.put(222, "anu");
        h.put(333, "durga");
        System.out.println(h);

        Set<Integer> s = h.keySet();
        System.out.println(s);

        Collection<String> s1 = h.values();
        System.out.println(s1);

        Set<Entry<Integer, String>> s2 = h.entrySet();
        Iterator<Entry<Integer, String>> iterator = s2.iterator();
```

```

        while(iterator.hasNext())
    {
        Entry<Integer, String> e = iterator.next();
        System.out.println(e.getKey()+"---"+e.getValue());
    }
}

```

Java.util.LinkedHashMap:-

```

public class java.util.LinkedHashMap extends java.util.HashMap
                                implements java.util.Map

```

- 1) interdicted in 1.4 version
- 2) Heterogeneous data allowed.
- 3) Underlying data Structure is HashTable & linkedlist.
- 4) Duplicate keys are not allowed but values can be duplicated.
- 5) Insertion order is preserved.
- 6) Null is allowed for key(only once)and allows for values any number of times.
- 7) Every method is non-synchronized.

Constructors:-

LinkedHashMap(); it creates default HashMap. Default capacity : 16 default fill ratio : 0.75

LinkedHashMap(java/util/Map); Used to add one map data into another map.

LinkedHashMap(int user-capacity); It creates the hashmap with specified capacity.

LinkedHashMap(int user-capacity, float fillRatio); creates hashMap with specified capacity & fillRatio.

Test.java:

```

package com.dss;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map.Entry;
import java.util.Set;
public class Test {
    public static void main(String[] args) {
        HashMap<Emp, Student> h = new HashMap<Emp, Student>();
        h.put(new Emp(111, "ratan"), new Student(1, "aaa"));
        h.put(new Emp(222, "anu"), new Student(2, "bbb"));

        Set<Entry<Emp, Student>> s2 = h.entrySet();
        Iterator<Entry<Emp, Student>> iterator = s2.iterator();
        while(iterator.hasNext())
        {
            Entry<Emp, Student> e = iterator.next();
            Emp ee = e.getKey();
            System.out.println(ee.eid+"---"+ee.ename);
            Student ss = e.getValue();
            System.out.println(ss.sid+"--"+ss.sname);
        }

        for (Entry<Emp, Student> m:h.entrySet())
        {
            Emp e = m.getKey();
        }
    }
}

```

```

        System.out.println(e.eid+"---"+e.ename);
        Student s = m.getValue();
        System.out.println(s.sid+"---"+s.sname);
    }
}
}

```

There are two approaches to add one map data into another map

1. constructor approach
2. by using putAll() method

constructor approach is used to add only one Map data into another Map.

putAll() method is used to add more than one map data into another Map.

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        //constructor approach
        LinkedHashMap<Integer,String> h1 = new LinkedHashMap<Integer,String>();
        h1.put(111,"ratan");
        LinkedHashMap<Integer,String> h2 = new LinkedHashMap<Integer,String>(h1);
        h2.put(222,"anu");

        for (Map.Entry m : h2.entrySet())
        {
            System.out.println(m.getKey()+"---"+m.getValue());
        }

        //by using putAll() to add the data
        LinkedHashMap<Integer,String> h11 = new LinkedHashMap<Integer,String>();
        h11.put(111,"ratan");

        LinkedHashMap<Integer,String> h22 = new LinkedHashMap<Integer,String>();
        h22.put(222,"anu");

        LinkedHashMap<Integer,String> h33 = new LinkedHashMap<Integer,String>();
        h33.putAll(h11);
        h33.putAll(h22);

        for(Entry<Integer, String> e :h3.entrySet())
        {
            System.out.println(e.getKey()+"---"+e.getValue());
        }
    }
}

```

Java.util.TreeMap:-

```
public class java.util.TreeMap extends java.util.AbstractMap
    implements java.util.NavigableMap,java.lang.Cloneable, java.io.Serializable
```

- 1) This class is introduced in 1.2 version.
- 2) It allows homogeneous data if we are trying to insert heterogeneous data at runtime while performing sorting JVM will generate ClassCastException.
- 3) Duplicate keys are not allowed but values can be duplicated.
- 4) Insertion order is not preserved it is based on some sorting order of keys.
- 5) The underlying data structure is red-black trees.
- 6) For empty TreeSet it is possible to insert null key once, but if the TreeSet contains data if we are inserting null keys at runtime we will get NullPointerException but for the values any number of null values insertion possible.

Constructors:-

TreeMap(); it will create empty TreeMap that will be sorted by using natural order of its keys.

TreeMap(java/util/Comparator); It creates TreeMap that will be sorted by using customized sorting order.

TreeMap(java/util/Map); adding one map data into another map

TreeMap(java/util/SortedMap); It creates the TreeMap by initializing SortedMap data.

Observations of TreeMap:

Case 1:-

```
TreeMap h = new TreeMap();
h.put(444,"ratan");
h.put(222,"anu");
h.put(111,"aaa");
System.out.println(h); // {111=aaa, 222=anu, 444=ratan}
```

In TreeMap when we insert the data that will be printed in sorting order based on key.

Case 2:-

```
TreeMap h = new TreeMap();
h.put(444,"ratan");
h.put("ratan","aaa"); // java.lang.ClassCastException
System.out.println(h);
```

TreeMap allows homogeneous data, if we are inserting heterogeneous data while performing sorting it will generate java.lang.ClassCastException.

Case 3:-

```
TreeMap h = new TreeMap();
h.put(444,"ratan");
h.put(null,"aaa"); // java.lang.NullPointerException
System.out.println(h);
```

If the treemap contains data then we are adding null value hence while performing sorting it will generate `java.lang.NullPointerException`(any object with comparision of null it will generate `NullPointerException`)

Example:- TreeMap data sorting (Constructor-2)

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        TreeMap h = new TreeMap(new MyComp());
        h.put("ratan",111);
        h.put("anu",222);
        h.put("zzz",333);
        System.out.println(h);           //{{zzz=333, ratan=111, anu=222}
    }
}

class MyComp implements Comparator <String>
{
    public int compare(String s1,String s2)
    {
        //place the comments check the output
        return s1.compareTo(s2);
        //return s2.compareTo(s1);
        //return -s1.compareTo(s2);
        //return s2.compareTo(s1);
    }
}
```

Example : adding one map data into another map (constructor-3)

```
TreeMap<Integer, String> t1 = new TreeMap<Integer, String>();
t.put(10, "ratan");
t.put(20, "anu");

TreeMap<Integer, String> t2 = new TreeMap<Integer, String>(t1);
t.put(10, "ratan");
t.put(20, "anu");
System.out.println(t2);
```

Example :- creating new TreeMap by passing sortedMap data (constructor-4)

```
import java.util.SortedMap;
import java.util.TreeMap;
public class Test {
    public static void main(String[] args) {
        TreeMap<Integer, String> t = new TreeMap<Integer, String>();
        t.put(10, "ratan");
        t.put(20, "anu");
        t.put(5, "durga");
        t.put(3, "sunny");
        System.out.println(t);
    }
}
```

```
SortedMap<Integer, String> sm = t.subMap(5, 20);
TreeMap<Integer, String> tt = new TreeMap<Integer, String>(sm);
System.out.println(tt);
}
}

Example:-
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        TreeMap<Integer, String> h = new TreeMap<Integer, String>();
        h.put(111,"ratan");
        h.put(222,"anu");
        h.put(333,"aaa");
        h.put(444,"aaa");
        h.put(555,"ccc");
        System.out.println(h);
        h.remove(555);
        System.out.println(h.firstEntry());
        System.out.println(h.lastEntry());
        System.out.println(h.firstKey());
        System.out.println(h.lastKey());
        System.out.println(h.lowerKey(222));
        System.out.println(h.higherKey(222));

        //creation of new TreeMap by passing SortedMap data
        SortedMap s1 = h.headMap(333);
        TreeMap t1 = new TreeMap(s1);
        System.out.println(t1);

        //creation of new TreeMap by passing SortedMap data
        SortedMap s2 = h.tailMap(333);
        TreeMap t2 = new TreeMap(s2);
        System.out.println(t2);
    }
}
```

Example :-

Ceiling() it return current provided value or greater value but if treemap does not contains same or greater value then it returns null .

floor():- it returns current value or less value but if treemap does notcontains same value or less then it return null.

pollFirstEntry:- it removes first entry & it prints that entry.

pollLastEntry():- it removes last entry and it prints that entry.

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        TreeMap h = new TreeMap();
        h.put(111,"ratan");
        h.put(222,"anu");
        h.put(444,"aaa");
        System.out.println(h);
        System.out.println(h.ceilingKey(222));
        System.out.println(h.ceilingEntry(333));
        System.out.println(h.floorKey(222));
        System.out.println(h.floorEntry(333));
        System.out.println(h.ceilingKey(666));

        Map.Entry m1 = h.pollFirstEntry();
        System.out.println(m1.getKey()+"---"+m1.getValue());

        Map.Entry m2 = h.pollLastEntry();
        System.out.println(m2.getKey()+"---"+m2.getValue());

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

Java.util.HashTable:-

```
public class java.util.Hashtable extends java.util.Dictionary
    implements java.util.Map,java.lang.Cloneable, java.io.Serializable
```

1. Introduced in the 1.0 version it's a legacy class.
2. Heterogeneous data allowed for both key & value.
3. Duplicate keys are not allowed but values can be duplicated.
4. Every method is synchronized hence only one thread is allowed to access it is a Thread safe but performance is decreased.
5. Null is not allowed for both key & Value , if we are trying to insert null values we will get NullPointerException.
6. The underlying datastructure is hashtable.

Constructors:-

HashTable(); it creates default HashMap.

HashTable (java/util/Map<? extends K, ? extends V> var);

it creates the HashMap by initializing the values specified in var.

HashTable (int capacity);

It creates the hashmap with specified capacity but the default capacity is 11.

HashTable (int capacity, float fillRatio);

It creates the hashMap with specified capacity & fillRatio.

Example:-

```
import java.util.Hashtable;
import java.util.Collection;
import java.util.Set;
class Test
{
    public static void main(String[] args)
    {
        Hashtable<Integer, String> h = new Hashtable<Integer, String>();
        h.put(1, "one");
        h.put(2, "two");
        h.put(3, "three");
        System.out.println(h);
        System.out.println(h.get("1")); //one
        System.out.println(h.isEmpty());
        h.remove(3);
        System.out.println(h.containsKey("1"));
        System.out.println(h.containsKey("3"));
        System.out.println(h.containsValue("one"));
        System.out.println(h.size());
        System.out.println(h.isEmpty());
        h.clear();
```

```

        System.out.println(h.isEmpty());
    }
}

```

Ex:

```

import java.util.Hashtable;
import java.util.Enumeration;
public class Test {
public static void main(String[] args) {
    Enumeration names;
    String key;

    Hashtable<String, String> hashtable = new Hashtable<String, String>();

    // Adding Key and Value pairs to Hashtable
    hashtable.put("Key1", "ratan");
    hashtable.put("Key2", "anu");
    hashtable.put("Key3", "durga");

    names = hashtable.keys();
    while(names.hasMoreElements()) {
        key = (String) names.nextElement();
        System.out.println("Key: " +key+ " & Value: " +
                           hashtable.get(key));
    }
}
}

```

Java.util.IdentityHashMap:-

```

public class java.util.IdentityHashMap extends           java.util.AbstractMap
                                         implements      java.util.Map,java.io.Serializable, java.lang.Cloneable

```

It is same as hashmap except one difference,

In case of Hashmap JVM will use equals() method to identify duplicate keys.(it performs content comparison)

In case of identityhashmap JVM will use ==operator to identify the duplicate keys.(it perform reference comparison)

Example:-

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        //equals() method to identify duplicate keys.
        HashMap<Integer, String> h = new HashMap<Integer, String>();
        h.put(new Integer(10), "ratan");
        h.put(new Integer(10), "anu");
    }
}

```

```

        System.out.println(h);

//== operator to identify duplicate keys.
IdentityHashMap<Integer, String> h1 = new IdentityHashMap<Integer, String>();
h1.put(new Integer(10), "ratan");
h1.put(new Integer(10), "anu");
System.out.println(h1);
}

E:\>java Test
{10=anu}
{10=anu, 10=ratan}

```

Java.util.WeakHashMap:-

<code>public class java.util.WeakHashMap</code>	extends	<code>java.util.AbstractMap</code>
	implements	<code>java.util.Map</code>

WeakHashMap is same as HashMap except following difference,

If an object is associated with hashmap that object is not destroyed even though it does not contains any reference type.

But in case of weakhashmap if the object does not contains reference type that object is eligible for garbage collector even though it associated with weakhashmap.

HashMap

```

import java.util.*;
class A
{
    public String toString()
    {
        return "A";
    }
    public void finalize()
    {System.out.println("object destroyed");}
}

```

```

    }

};

class Test
{
    public static void main(String[] args)
    {
        HashMap h = new HashMap();
        A a = new A();
        h.put(a, "ratan");
        System.out.println(h);
    }
}
```

```

        a=null;
        System.gc();
        System.out.println(h);
    }
}

E:\>java Test
{A=ratan}
{A=ratan}

WeakHashMap
import java.util.*;
class A
{
    public String toString()
    {
        return "A";
    }
    public void finalize()
    {System.out.println("object destroyed");
}
}
class Test
{
    public static void main(String[] args)
    {WeakHashMap h = new WeakHashMap();
        A a = new A();
        h.put(a,"ratan");
        System.out.println(h);
        a=null;
        System.gc();
        System.out.println(h);
    }
}

E:\>java Test
{A=ratan}
{}
object destroyed

```

Conversion of non-synchronized version to synchronized :

- ✓ Generally collection class methods are non- synchronized by default but it is possible to get synchronized version of Collection classes.

To get synchronized version of List interface use fallowing Collections class static method

public static List synchronizedList(List l)

```
ArrayList al = new ArrayList();
List l = Collections.synchronizedList(al);
```

To get synchronized version of Set interface use fallowing Collections class static method

public static Set synchronizedSet(Set s)

```
HasSet h = new HashSet();
Set h1 = Collections.synchronizedSet(h);
```

To get synchronized version of Map interface use fallowing Collections class static method

public static Map synchronized Map(Map m)

```
HashMap h = new HashMap();
Map m = Collections.synchronizedMap(h);
```

To get synchronized version of TreeSet use following Collections class static method

Collections.synchronizedSortedSet(SortedSet<T> s)

TreeSet t = new TreeSet();

SortedSet s = Collections.synchronizedSortedSet(t);

To get synchronized version of TreeMap use following Collections class static method

Collections.synchronizedSortedMap(SortedMap<K,V> m)

TreeMap t = new TreeMap();

SortedMap s = Collections.synchronizedSortedMap(t);

Java.util.Properties:

- ✓ To get the flexibility of modifications use properties file.
- ✓ In standalone applications(JDBC) or web-applications(web sites) the data is frequently changing like, Database username, Database password, url, driver ...etc
- ✓ in above scenario for every change must perform modifications in all .java files but it is complex. to overcome this problem use properties file.
- ✓ Properties file is a normal text file with .properties extension & it contains key=value formatted data but both key and value is string format.
- ✓ Once we done modifications on .properties file that modifications are reflected all the .java files.

abc.properties :

```
username = system  
password = manager
```

Test.java:-

```
import java.util.*;  
import java.io.*;  
class Test  
{    public static void main(String[] args) throws FileNotFoundException, IOException  
    {        //locate properties file  
        FileInputStream fis=new FileInputStream("abc.properties");  
        //load the properties file by using load() method of Properties class  
        Properties p = new Properties();  
        p.load(fis);  
  
        //get the data from properties class by using getProperty()  
        String username = p.getProperty("username");  
        String password = p.getProperty("password");  
  
        //use the properties file data  
        System.out.println("DataBase username="+username);  
        System.out.println("DataBase password =" +password);  
    }  
}
```

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) By using collection framework classes is it possible to store primitive data?
- 5) What is the root interface of collection framework?
- 6) What is the root interface of Map?
- 7) List out implementation classes of List interface?
- 8) What is the parent interface of Collection interface?
- 9) What are the collection classes not allowed heterogeneous data?
- 10) Can you please tell me some of the legacy classes present in collection framework?
- 11) What do you mean by auto-boxing?
- 12) Arrays are type safe or not?
- 13) How to provide type safety to the collection?
- 14) What is the purpose of generic version of collection classes?
- 15) What is the difference between general version of ArrayList and generic version of ArrayList?
- 16) What is purpose of generic version of ArrayList & arrays?
- 17) What is the difference between normal version of collections & generic version of collection?
- 18) What are the ways to add one Collections data into another collection?
- 19) What is the difference between removeAll() & retainsAll()?
- 20) When we get the IndexOutOfBoundsException?
- 21) When we get the ArrayIndexOutOfBoundsException?
- 22) When we will get StringIndexOutOfBoundsException?
- 23) How to convert Collection data to arrays & Arrays data to collection?
- 24) How to convert arrays to collections & collections to arrays?
- 25) What is the difference between ArrayList and LinkedList?
- 26) How to decide when to use ArrayList and when to use LinkedList?
- 27) What is the difference between ArrayList & vector?
- 28) Arrays used to hold homogeneous data but what is the purpose of generic version of Collection?
- 29) What is the purpose of RandomAccess interface and it is marker interface or not?
- 30) All collection classes are commonly implemented Serializable & Cloneable what is the purpose?
- 31) What do you mean by cursor and how many cursors present in java?
- 32) How many ways are there to retrieve objects from collections classes what are those?
- 33) What is the purpose of Enumeration cursor and how to get that cursor object?
- 34) What are the cursors used to retrieve the objects forward & backward direction?
- 35) What is the purpose of Iterator and how to get Iterator Object?
- 36) What is the purpose of ListIterator and how to get that object?
- 37) What is the difference between Enumeration vs Iterator Vs ListIterator?
- 38) What are the applicable cursors on ArrayList, Vector, LinkedList?
- 39) List out implementation classes of set interface?
- 40) What is the difference between HashSet & linkedHashSet?
- 41) What is the purpose of TreeSet class?

- 42) What is the difference between Set & List interface?
- 43) How to perform the sorting of collection data?
- 44) To perform the sorting what are the conditions?
- 45) To perform the default sorting it internally uses which method?
- 46) What is the difference between collection & collections?
- 47) What is the difference between comparable & comparator?
- 48) When we will get ClassCastException?
- 49) When we will get the NullPointerException?
- 50) In Comparable by using which method we are writing sorting logics?
- 51) Is it possible to compare two different objects?
- 52) What is the difference between compareTo() & equals() method?
- 53) In Comparator by using which method we are writing sorting logics?
- 54) What is the purpose of Map interface?
- 55) List out implementation classes of map interface?
- 56) What do you mean by entry.
- 57) How to get all values objects , key objets,entry objects?
- 58) What is the difference between HashMap & LinkedHashMap?
- 59) How many ways are there to add one map data into another Map?
- 60) What is the difference between TreeSet and TreeMap?
- 61) What is the difference between IdentityHashMap & WeakHashMap?
- 62) What is the difference between HashTable and Properties file key=value pairs?
- 63) HashTable null values are allowed or not?
- 64) What do you mean by properties file and what are the advantages of properties file?
- 65) By using properties file we can declare different type of data or not?
- 66) What is the difference between HashTable & HasMap?
- 67) How to convert non-synchronized version of method into synchronized version method?

***** Thank you *****

INTERNATIONALIZATION (i18N)

i18N enables the application to support in different languages.

- Internationalization is also called as i18n because in between I & n 18 characters are present.
- By using Locale class and ResourceBundle class we are enable i18n on the application.
- Local is nothing but language + country.
- For making your application to support i18n we need to prepare local specific properties file it means for English one properties file & hindi one properties file ...etc.
- The property file format is key = value
- The properties file name followed pattern bundlename with language code and country code.
 - ApplicationMessages_en_US.properties.

- In single web application contains different properties file all the properties files key must be same and values are changed local to Locale.

Java.util.Locale:-

- Locale Object is decide properties file based on argument you passed and then it display locale specific details based on Properties file entry.

```
Locale l = new Locale(args[0],args[1]);
Locale l = new Locale(en,US);
```

D:\5batch>javap java.util.Locale

Compiled from "Locale.java"

```
public final class java.util.Locale extends java.lang.Object {
    public static final java.util.Locale ENGLISH;
    public static final java.util.Locale FRENCH;
    public static final java.util.Locale GERMAN;
    public static final java.util.Locale ITALIAN;
    public static final java.util.Locale JAPANESE;
    public static final java.util.Locale KOREAN;
    public static final java.util.Locale CHINESE;
    public static final java.util.Locale SIMPLIFIED_CHINESE;
    public static final java.util.Locale TRADITIONAL_CHINESE;
    public static final java.util.Locale FRANCE;
    public static final java.util.Locale GERMANY;
    public static final java.util.Locale ITALY;
    public static final java.util.Locale JAPAN;
    public static final java.util.Locale KOREA;
    public static final java.util.Locale CHINA;
    public static final java.util.Locale PRC;
    public static final java.util.Locale TAIWAN;
    public static final java.util.Locale UK;
    public static final java.util.Locale US;
    public static final java.util.Locale CANADA;
    public static final java.util.Locale CANADA_FRENCH;
```

Sample Language Codes

Language Code	Description
de	German
en	English
fr	French
ru	Russian
ja	Japanese
sv	Javanese
ko	Korean
zh	Chinese

To get particular language and country code use following example:-

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Locale l = Locale.FRANCE;
        System.out.println(l.getLanguage());
        System.out.println(l.getCountry());
    }
}
```

D:\5batch>java Test

fr
FR

Java.util.ResourceBundle:-

Creates ResourceBundle object by passing Local object then by using ResourceBundle we are able to get data form properties file that is decide by Locale.

- It is possible to create ResourceBundle Object without specifying Locale it will take default properties file with default language.
ResourceBundle bundle1 = ResourceBundle.getBundle("Application");
- It is possible to create ResourceBundle Object by specifying default Locale object.
ResourceBundle bundle2 = ResourceBundle.getBundle("Application",Locale.FRANCE);
- It is possible to create ResourceBundle object by creating new user Locale Object
ResourceBundle bundle3 = ResourceBundle.getBundle("Application",new Locale("ratan","RATAN"));

Application 1:-**Steps to design application:-**

Step-1:- prepare properties files to support different languages and countries.

<i>Application.properties</i>	default properties file(base properties file)
<i>Application_fr_FR.properties</i>	French properties file
<i>Allication_ratan_RATAN.properties</i>	Ratan country properties file

Step 2:- create locale object it identified particular language and country and it decides execution of properties file.

```
Locale l = new Locale("en","US");
```

The above statement specify language is English and country united states

```
Locale l = new Locale("fr","CA");
```

```
Locale x = new Locale("fr","FR");
```

The above two locales specifies France language in Canada & France

Instead of hard coding language name and country name get the values from command prompt at runtime.

```
Public static void main(String[ ] args)
{
    Locale l = new Locale(args[0],args[1]);
}
D:\5batch>java Test fr FR
```

Step 3:-create ResourceBundle by passing Locale object.

//if no local is Matched this property file is executed [default property file]

```
ResourceBundle bundle1 = ResourceBundle.getBundle("Application");
```

//it create ResourceBundle with local that is already defined [France properties file]

```
ResourceBundle bundle2 = ResourceBundle.getBundle("Application",Locale.FRANCE);
```

Step 4:- fetch the text form ResourceBundle

```
String msg = Bundle.getString("wish");
System.out.println(msg);
```

Application.properties:-

countryname = USA

lang = eng

Application_fr_FR.properties:-

countryname = canada

lang = france

Allication_ratan_RATAN.properties:-

countryname=Ratan

lang= ratan

Test.java:-

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        //if no local is Matched this property file is executed
        ResourceBundle bundle1 = ResourceBundle.getBundle("Application");
        //it create ResourceBundle with local that is already defined
        Locale l1 = Locale.FRANCE;
        ResourceBundle bundle2 = ResourceBundle.getBundle("Application",l1);
        //it creates ResourceBundle with new user created Locale
        Locale l2 = new Locale("ratan","RATAN")
        ResourceBundle bundle3 = ResourceBundle.getBundle("Application",l2);
        System.out.println(bundle1.getString("countryname")+"--"+bundle1.getString("lang"));
        System.out.println(bundle2.getString("countryname")+"--"+bundle2.getString("lang"));
        System.out.println(bundle3.getString("countryname")+"--"+bundle3.getString("lang"));
    }
}
```

Output:-

D:\5batch>java Test

USA--eng

Canada--france

Ratan--Ratan

APPLICATION 2:-

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        //creates local object with the help of arguments
        Locale l = new Locale(args[0],args[1]);
        //it creates resource bundle with local passed from as command line arguments
        ResourceBundle bundle = ResourceBundle.getBundle("Application",l);
        System.out.println(bundle.getString("countryname"));
        System.out.println(bundle.getString("lang"));
    }
}
```

D:\5batch>java Test x y

USA

eng

D:\5batch>java Test fr FR

canada

france

D:\5batch>java Test ratan RATAN

Ratan

ratan

Application before internationalization:-

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        System.out.println("hello");
        System.out.println("i like you");
        System.out.println("i hate you");
    }
}

```

We are decide to print this messages in different languages like Germany, French.....etc then we must translate the code in different languages by moving the message out of source code to text file it looks the program need to be internationalized(supporting different languages).

Application.properties:-

wish = hello
 lovely = i love you
 angry = i hate you

Application_fr_FR.properties:-

wish = hlloe
 lovely = i evol you
 angry = i etah you

Application_hi_IN.properties:-

wish=\u0c39\u0c46\u0c32\u0c4d\u0c32\u0c4a
 lovely=\u0c07 \u0c32\u0c4a\u0c35\u0c46 \u0c2f\u0c4a\u0c09
 angry=\u0c07 \u0c39\u0c24\u0c46 \u0c09

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Locale l = new Locale(args[0],args[1]);
        ResourceBundle rb = ResourceBundle.getBundle("Application",l);
        System.out.println(rb.getString("wish"));
        System.out.println(rb.getString("lovely"));
        System.out.println(rb.getString("angry"));
    }
}

```

D:\5batch>java Test fr FR
 hello
 i evol you
 i etah you

D:\5batch>java Test fr FR

hlloe

i evol you

i etah you

D:\5batch>java Test hi IN

?????

? ???? ???

? ??? ?

Conversion of any language to Unicode values:-

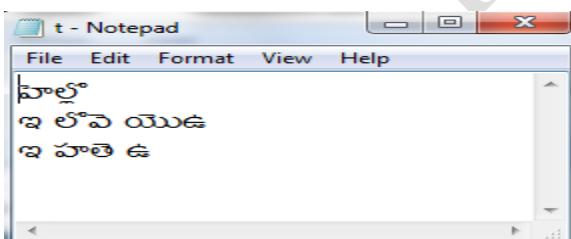
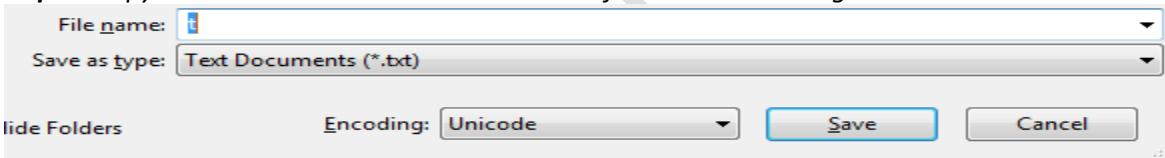
Step 1:- download Unicode editor from internet www.higopi.com

Converters Link

Above converter can also be downloaded and used offline from [here](#)

Step 2:- unzip the file and click on index.html page select language and type the words.

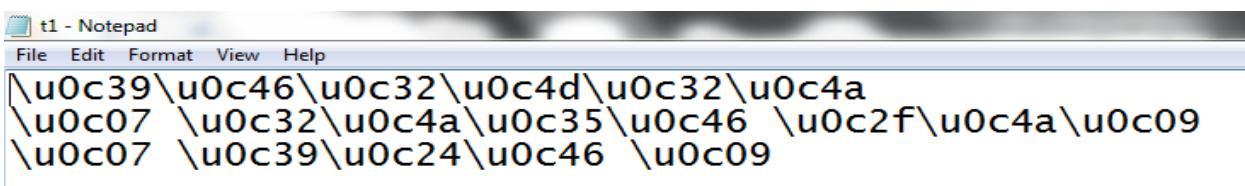
Step 3:- copy the content and save the data in text file and while saving select **Unicode**.



Step 4:- convert the above language to Unicode character format.

Syntax:- **native2ascii -encoding encoding-name source-file destination-file**

D:\>native2ascii -encoding unicode t.txt output.txt



Application :-**Application.properties:-**

wish = hello
 lovely = i love you
 angry = i hate you

Application_fr_FR.properties:-

wish = hlloe
 lovely = i evol you
 angry = i etah you

Application_tl_IN.properties:-

wish=\u0c39\u0c46\u0c32\u0c4d\u0c32\u0c4a
 lovely=\u0c07 \u0c32\u0c4a\u0c35\u0c46 \u0c2f\u0c4a\u0c09
 angry=\u0c07 \u0c39\u0c24\u0c46 \u0c09

Test.java:-

```
import java.util.*;
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Locale l = new Locale(args[0],args[1]);
        ResourceBundle b = ResourceBundle.getBundle("Application",l);
        Frame f = new Frame();           //to create frame
        f.setVisible(true);             //to provide visibility to frame
        f.setSize(300,75); //to align the frame set bounds
        f.setLayout(new FlowLayout()); //to set the frame proper format
        //creation of buttons with labels
        Button b1 = new Button(b.getString("wish"));
        Button b2 = new Button(b.getString("lovely"));
        Button b3 = new Button(b.getString("angry"));
        //adding buttons into frame
        f.add(b1);
        f.add(b2);
        f.add(b3);
    }
}
```

D:\5batch>java Test tl IN





Test.java:- example

```
import java.util.*;
public class Test {
    static public void main(String[] args) {
        String language;
        String country;
        Locale currentLocale;
        ResourceBundle messages;
        if (args.length != 2)
        {
            language = new String("en");
            country = new String("US");
        }
        else
        {
            language = new String(args[0]);
            country = new String(args[1]);
        }
        currentLocale = new Locale(language, country);
        messages = ResourceBundle.getBundle("Application", currentLocale);
        System.out.println(messages.getString("wish"));
        System.out.println(messages.getString("lovely"));
        System.out.println(messages.getString("angry"));
    }
}
```

D:\5batch>java Test

hello

i love you

i hate you

D:\5batch>java Test x y

hello

i love you

```
i hate you
D:\5batch>java Test tl IN
??????
? ???? ???
? ??? ?
D:\5batch>java Test fr FR
hllo
i evol you
i etah you
```

Example :- display Date in different Locale.

`DateFormat.DEFAULT,`
`DateFormat.SHORT,`
`DateFormat.MEDIUM,`
`DateFormat.LONG,`
`DateFormat.FULL`

Sample Date Formats

Style	U.S. Locale	French Locale
DEFAULT	Jun 30, 2009	30 juin 2009
SHORT	6/30/09	30/06/09
MEDIUM	Jun 30, 2009	30 juin 2009
LONG	June 30, 2009	30 juin 2009
FULL	Tuesday, June 30, 2009	mardi 30 juin 2009

Test.java:-

```
import java.util.*;
import java.text.DateFormat;
class Test
{
    public static void main(String[] args)
    {
        Date d = new Date();
        //default locale en US
        DateFormat df1 = DateFormat.getDateInstance(DateFormat.DEFAULT, Locale.getDefault());
        System.out.println(df1.format(d));
        //date of fresh
        DateFormat df2 = DateFormat.getDateInstance(DateFormat.MEDIUM, Locale.FRENCH);
        System.out.println(df2.format(d));
        //date of Italy
        DateFormat df3 = DateFormat.getDateInstance(DateFormat.SHORT, Locale.ITALY);
        System.out.println(df3.format(d));
    }
};
```

Example on time format:-

Sample Time Formats

Style	U.S. Locale	German Locale
DEFAULT	7:03:47 AM	7:03:47
SHORT	7:03 AM	07:03
MEDIUM	7:03:47 AM	07:03:07
LONG	7:03:47 AM PDT	07:03:45 PDT
FULL	7:03:47 AM PDT	7.03 Uhr PDT

```
import java.util.*;
import java.text.*;
class Test
{
    public static void main(String[] args)
    {
        Date d = new Date();
        DateFormat df1 = DateFormat.getTimeInstance(DateFormat.DEFAULT, Locale.getDefault());
        System.out.println(df1.format(d));
        DateFormat df2 = DateFormat.getTimeInstance(DateFormat.MEDIUM, Locale.FRENCH);
        System.out.println(df2.format(d));
        DateFormat df3 = DateFormat.getTimeInstance(DateFormat.SHORT, Locale.ITALY);
        System.out.println(df3.format(d));
    }
};
```

Example on both data and Time format:-

Sample Date and Time Formats

Style	U.S. Locale	French Locale
DEFAULT	Jun 30, 2009 7:03:47 AM	30 juin 2009 07:03:47
SHORT	6/30/09 7:03 AM	30/06/09 07:03
MEDIUM	Jun 30, 2009 7:03:47 AM	30 juin 2009 07:03:47
LONG	June 30, 2009 7:03:47 AM PDT	30 juin 2009 07:03:47 PDT
FULL	Tuesday, June 30, 2009 7:03:47 AM PDT	mardi 30 juin 2009 07 h 03 PDT

```
import java.util.*;
import java.text.*;
class Test
{
    public static void main(String[] args)
    {
        Date d = new Date();
        DateFormat df1 = DateFormat.getDateInstance(DateFormat.FULL, DateFormat.FULL, Locale.getDefault());
        System.out.println(df1.format(d));
        DateFormat df2 = DateFormat.getDateInstance(DateFormat.FULL, DateFormat.FULL, Locale.FRENCH);
        System.out.println(df2.format(d));
    }
};
```

Internationalization

- 1) *What is the main importance of I18n?*
- 2) *What is the purpose of locale class?*
- 3) *What is the format of the properties file?*
- 4) *Local class present in which package?*
- 5) *What do you mean by properties file and what it contains?*
- 6) *What is the purpose of ResourceBundle class and how to create object?*
- 7) *How to convert different languages characters into Unicode characters?*
- 8) *What is the command used to convert different language characters into Unicode characters?*
- 9) *Who decides properties file executions?*
- 10) *What is the method used to get values from properties file?*
- 11) *By using which classes we are achieving i18n?*
- 12) *What is the default Locale and how to get it?*
- 13) *Is it possible to create your own locale?*
- 14) *What is purpose of DateFormat class and it is preset in which package?*
- 15) *What are the DateFormat Constantans' to print Date & time?*
- 16) *How to print date in different Locales?*
- 17) *How to print time in different locales?*
- 18) *How to print both date & time by using single method?*
- 19) *What do you mean by factory method? getBundle() is factory method or not?*
- 20) *How to get particular locale language & country?*

***** ***Thank you*** *****

JVM Architecture

- ✓ JVM is a software it is a specification that provide the runtime environment in which the java byte code executed.
- ✓ JVM is a platform dependent software(installing along with JDK software).
- ✓ Java is developed with the concept of WORA which runs on a VM(virtual machine).
- ✓ The JVM understandable file format is .class file it contains byte code.

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

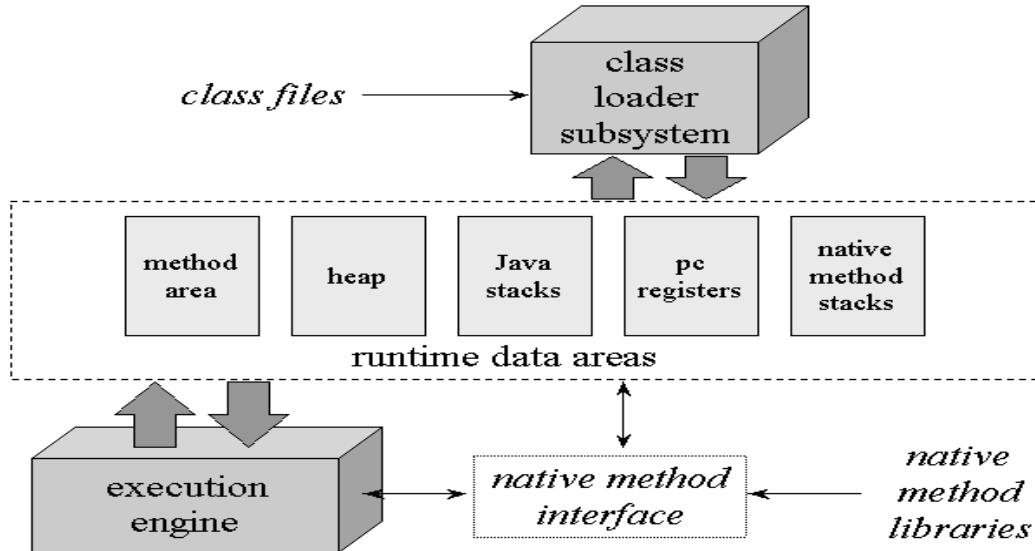
Operations of JVM:-

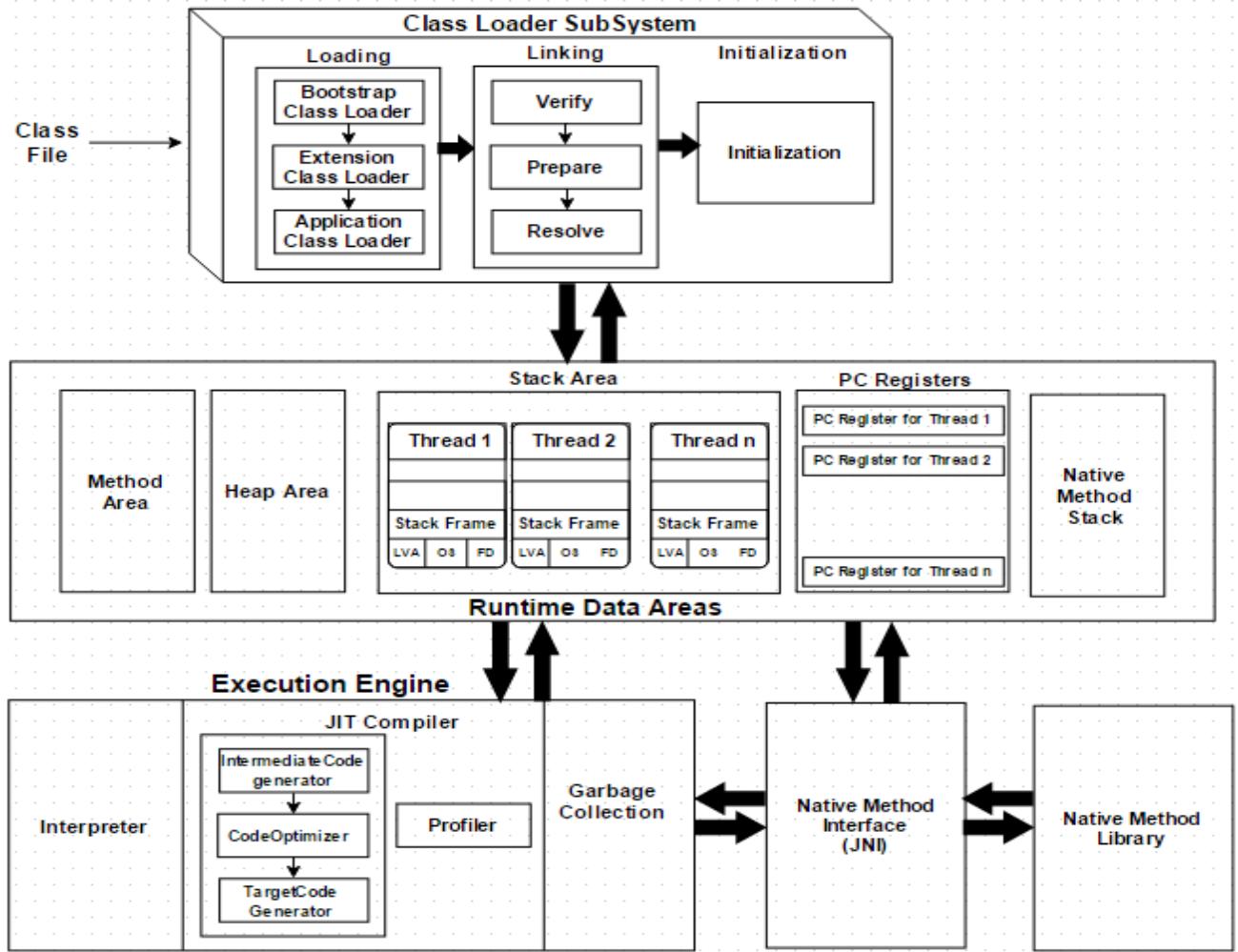
- ✓ Loading the byte code into memory. Converting byte code instructions into machine instructions.
- ✓ Allocating sufficient memory space for class properties.
- ✓ Providing runtime environment in which java byte code is executed.
- ✓ JVM implementation is known as JRE(java runtime environment)
- ✓ JVM is interpreter to execute java program line by line.

JVM perform mainly four actions :-

- 1) Loading .class file byte code into memory [class loader subsystem]
- 2) Memory areas
- 3) Running application [Execution engine]
- 4) Native method interface.

JVM Architecture:-





Class loader subsystem:-

1. It is used to load the .class file into memory. Read the data from .class file hard disk loading into
2. It verifies the byte code instructions.
3. It allots the memory required for the program.

It performs three activities,

a. Loading

Reading .class file information and storing .class file binary information in method area.

The JVM will create the class Class object to access the class binary information.

b. Linking

a. Verification

Byte code verifier will verify whether the generated byte code proper or not. If the verification fails we will get Verification error.

b. Preparation

JVM will allocate memory for static variables & assign default values but not original values & original values are assigned in initialization.

c. Resolution

c. Initialization

Load the student class--> create the class obj --> stored in heap (form of class obj not student object)

Emp.java

```
class Emp
{
    private int eid;
    private String ename;
    public void setEname(String ename)
    {
        this.ename=ename;
    }
    public void setEid(int eid)
    {
        this.eid=eid;
    }
    public int getEid()
    {
        return eid;
    }
    public String getEname()
    {
        return ename;
    }
}
```

Test.java

```
import java.lang.reflect.*;
class Test
{
    public static void main(String[] args) throws ClassNotFoundException
    {
        Class c = Class.forName("Emp");
        //To get declared methods
        Method[] m = c.getDeclaredMethods();
        for (Method mm :m)
        {
            System.out.println(mm);
        }

        //To get declared variables
        Field[] f = c.getDeclaredFields();
        for (Field ff :f)
        {
            System.out.println(ff);
        }
    }
}
G:\>java Test
public void Emp.setEname(java.lang.String)
public java.lang.String Emp.getEname()
public void Emp.setEid(int)
public int Emp.getEid()
int Emp.eid
java.lang.String Emp.ename
```

Class class object :-

For every loaded .class file only one Class obj is created by JVM even though we are using that class multiple times.

```
class Test
{
    public static void main(String[] args)
    {
        Emp e1 = new Emp();
        Class c1 = e1.getClass();

        Emp e2 = new Emp();
        Class c2 = e2.getClass();

        System.out.println(c1.hashCode());
        System.out.println(c2.hashCode());
        System.out.println(c1==c2);
    }
}
```

G:\>java Test

705927765705927765true

Different types of Class loader:-

- 1) Bootstrap class loader.
 - ✓ It loads the predefined class present in **rt.jar** file. rt.jar contains all predefined classes.(c:/program files/java/jdk/jre/rt.jar)
 - ✓ It is developed in non-java code.
- 2) Extension class loader.
 - ✓ It loads the .class files present in **ext** folder. This folder present C:\Program Files\Java\jdk1.8.0_65\jre\lib
 - ✓ It is implemented in java code. The class file name is **sun.misc.Launcher\$AppClassLoader@15db9742**
- 3) Application class loader.
 - ✓ It loads the .class files present application level path (Test.class, Emp.class)
 - ✓ It is implemented in java. The class file name is **sun.misc.Launcher\$ExtClassLoader@5c647e05**

Jar file creation :- G:\>jar -cvf Emp.jar Emp.class [It will create Emp.jar file]

To load this jar file place this jar file ext folder.

```
class Test
{
    public static void main(String[] args)
    {
        //String is predefined class present in rt.jar
        System.out.println(String.class.getClassLoader());
        // Application level class present in current directory
        System.out.println(Test.class.getClassLoader());
        //present in ext folder (in the form of jar)
        System.out.println(Emp123.class.getClassLoader());
    }
}
```

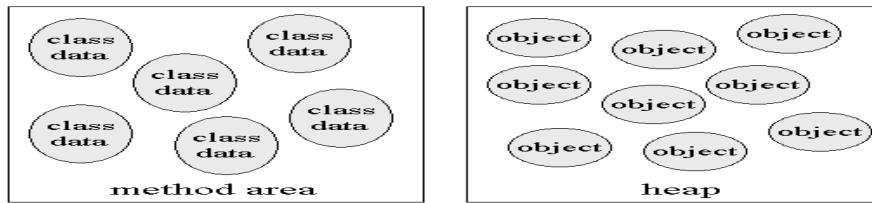
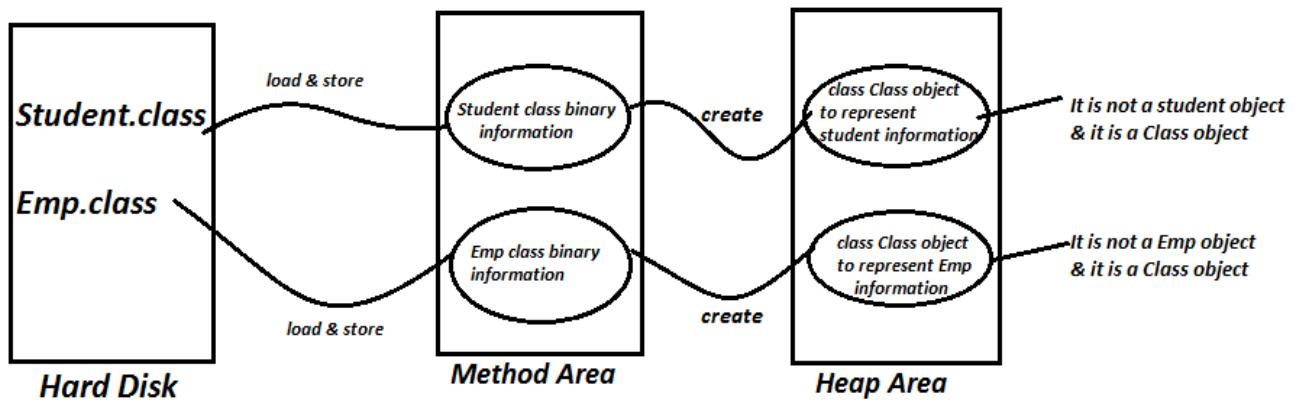
G:\>java Test

null

sun.misc.Launcher\$AppClassLoader@15db9742

sun.misc.Launcher\$ExtClassLoader@5c647e05

Heap area:-It is used to store the Objects. Class object is stored in heap memory.

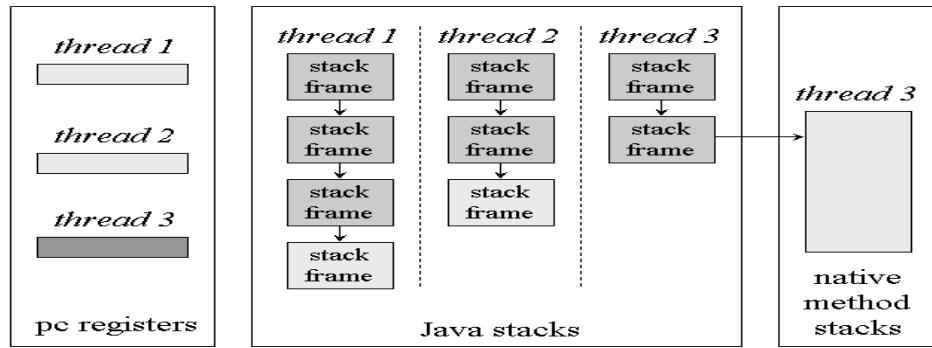


Runtime data area:-this is the memory resource used by the JVM and it is 5 types

Method Area:-It is used to store the class data and method data.

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.

PcRegister : it store the next instruction to be executed.

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.

Networking(java.net package)

Introduction to networking:-

- 1) The process of connecting the resources (computers) together to share the data is called networking.
- 2) Java.net is package it contains number of classes by using that classes we are able to connection between the devices (computers) to share the information.
- 3) Java socket programming provides the facility to share the data between different computing devices.
- 4) In the network we are having to components
 - a. Sender(source) :The person who is sending the data is called sender.
 - b. Receiver(destination) : The person who is receiving the data is called receiver

In the network one system can acts as a sender as well as receiver.
- 5) In the networking terminology we have client and server.
Client:- who takes the request & who takes the response is called client.
Server:- The server contains the project
 1. It takes the request from the client
 2. It identifies the requested resource
 3. It process the request
 4. It will generate the response to client

Categories of network:-

We are having two types of networks

- 1) Per-to-peer network.
- 2) Client-server network.

Client-server:- In the client server architecture always client system acts as a client and server system acts as a server.

Peer-to-peer:- In the peer to peer client system sometimes behaves as a server, server system sometimes behaves like a client the roles are not fixed.

Types of networks:-

Intranet:- It is also known as a private network. To share the information in limited area range(within the organization) then we should go for intranet.

Extranet:- This is extension to the private network means other than the organization , authorized persons able to access.

Internet:- It is also known as public networks. Where the data maintained in a centralized server hence we are having more sharability. And we can access the data from anywhere else.

The frequently used terms in the networking:-

- 1) IP Address
- 2) URL(Uniform Resource Locator)
- 3) Protocol
- 4) Port Number
- 5) MAC address.
- 6) Connection oriented and connection less protocol

7) Socket.

IP Address:-

- 1) IP Address is a unique identification number given to the computer to identify the computer uniquely in the network.
- 2) The IP Address is uniquely assigned to the computer it is should not duplicated.
- 3) The IP Address range is 0-255.
 125.0.4.255 ----> Valid 124.654.5.6 ----> Invalid
- 4) Each and every website contains its own IP Address we can access the sites through the names otherwise IP Address.

Site Name : www.google.com IP Address : 74.125.224.72

Example:-

```
import java.net.*;
import java.util.*;
class Test
{
    public static void main(String[] args) throws Exception
    {
        Scanner s = new Scanner(System.in);
        System.out.println("please enter site name");
        String sitename=s.nextLine();
        InetAddress in=InetAddress.getByName(sitename);
        System.out.println("the ip address is:"+in);
    }
}
```

G:\ratan>java Test

please enter site name

www.google.com

the ip address is:www.google.com/216.58.199.196

G:\ratan>java Test

please enter site name

aaa

Exception in thread "main" java.net.UnknownHostException: aaa

Protocol:-

The protocol is a set of rules followed in communication.

- ✓ TCP(Transmission Control Protocol)(connection oriented protocol)
- ✓ UDP (User Data Gram Protocol)(connection less protocol)
- ✓ Telnet
- ✓ SMTP(Simple Mail Transfer Protocol)
- ✓ IP (Internet Protocol)

MAC (media access control):-

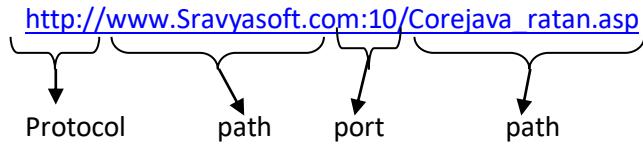
MAC address is a unique identifier for NIC(Network interface protocol). A network protocol can have multiple NIC but one unique MAC.

Port number:

The port number is used to identify the different applications uniquely .And it is associated with the ip address for communication between two applications.

URL(Uniform Resource Locator):-

- 1) URL is a class present in the java.net package.
- 2) By using the URL we are accessing some information present in the world wide web.



The URL contains information like

- a. Protocol : `http://`
- b. Server name IP address : `www.Sravyasoft.com`
- c. Port number of the particular application and it is optional(:10)
- d. File name or directory name `Corejava_ratan.asp`

Example:-

```
import java.net.*;
class Test
{
    public static void main(String[] args) throws Exception
    {
        URL url=new URL("http://www.Sravyasoft.com:10/index.html");
        System.out.println("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 communication acknowledgement sent by receiver so it is reliable but slow.
- To achieve the following communication the java peoples are provided the following classes.
- a. Socket
 - b. ServerSocket

Connection Less :- (UDP)

- 1) UDP is a protocol by using this protocol we are able to send data without using Physical Connection.
 - 2) In this communication acknowledgement not sent by receiver so it is not reliable but fast.
 - 3) This is a light weight protocol because no need of the connection between the client and server .
- To achieve the UDP communication the java peoples are provided the following classes.
1. DatagramPacket.
 2. DatagramSocket.

Socket:-

- 1) *Socket is used to create the connection between the client and server.*
- 2) *Socket is nothing but a combination of IP Address and port number.*
- 3) *The socket is created at client side.*

```
Socket s=new Socket(int IPAddress, int portNumber);
Socket s=new Socket("125.125.0.5",123);           Server IP Address. Server port number.
Socket s=new Socket(String HostName, int PortNumber);  Socket s=new Socket(Sravyasoft,123);
```

Client.java:-

```
import java.net.*;
import java.io.*;
class Client
{
    public static void main(String[] args) throws Exception
    {
        //to write the data to server
        Socket s=new Socket("localhost",5555);
        String str="ratan from client";
        OutputStream os=s.getOutputStream();
        PrintStream ps=new PrintStream(os);
        ps.println(str); //write the data to server
        //read the data from server
        InputStream is=s.getInputStream();
        BufferedReader br=new BufferedReader(new InputStreamReader(is));
        String str1=br.readLine();
        System.out.println(str1);
    }
}
```

Server.java:-

```
import java.io.*;
import java.net.*;
class Server
{
    public static void main(String[] args) throws Exception
    {
        //To read the data from client
        ServerSocket ss=new ServerSocket(5555);
        Socket s=ss.accept();
        System.out.println("connection is created ");
        Thread.sleep(2000);
        //Read the data from client
        InputStream is=s.getInputStream();
        BufferedReader br=new BufferedReader(new InputStreamReader(is));
        String data=br.readLine();
        System.out.println(data);
        Thread.sleep(2000);
        //write the data to the client
        data=data+"this is from server";
        OutputStream os=s.getOutputStream();
        PrintStream ps=new PrintStream(os);
        ps.println(data);
    }
}
```

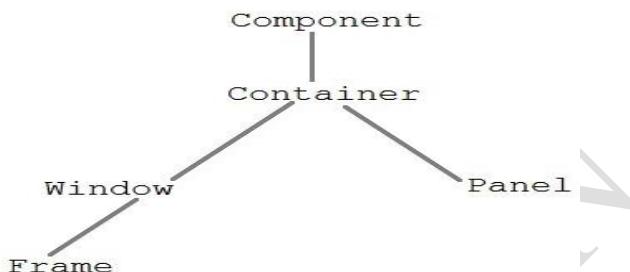
Java.awt package

- ✓ AWT(Abstract Window Tool kit) is an **API** to develop *GUI*, window based applications in java.
- ✓ AWT components are platform dependent it displays the application according to the view of operating system.
- ✓ AWT is a package it will provide very good predefined support to design *GUI* applications.
- ✓ Awt is heavy weight because these components are using operating system resources.
- ✓ By using `java.awt` package we are able to prepare static components to provide the dynamic nature to the component use `java.awt.event` package.(it is a sub package of `java.awt`).

component :-The root class of `java.awt` package is *Component* class.

Container:-It is a component in awt that contains another components like *Button*,*TextField*...etc

The classes that extends container classes those classes are containers such as *Frame*, *Dialog* and *Panel*.

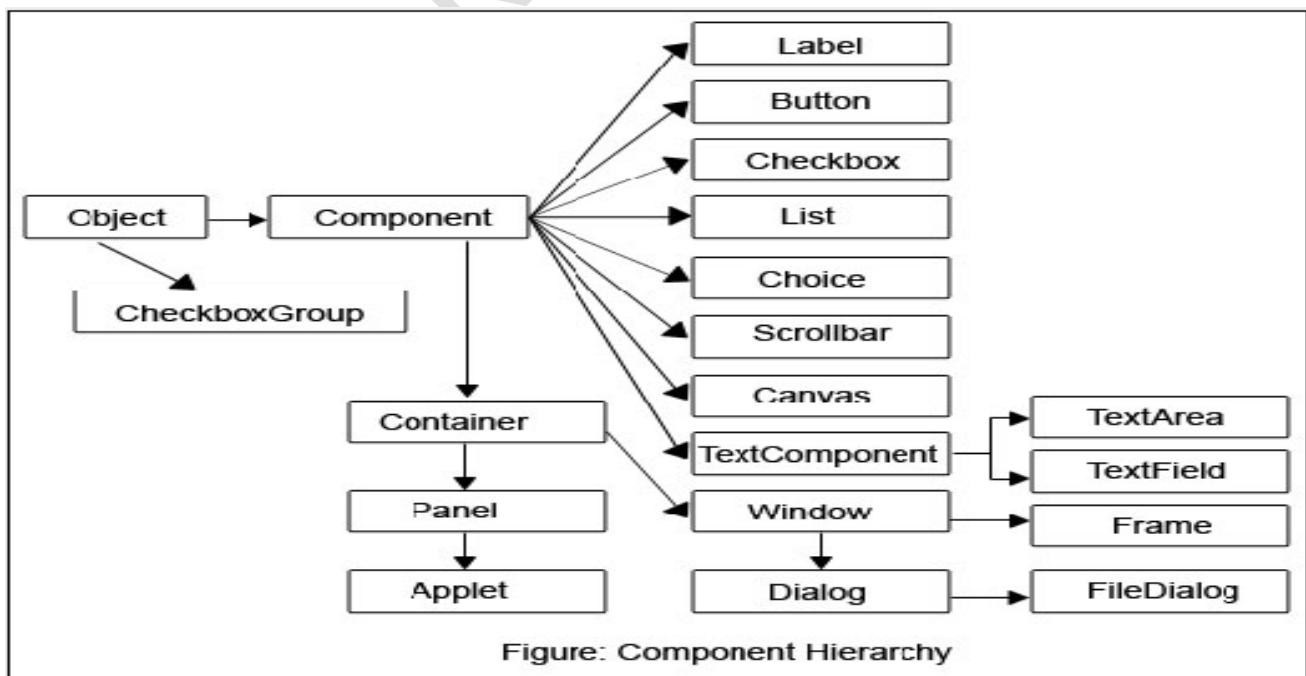


Window :- Window class creates a top level window. Window does not have borders and menu bar.

Panel : it is a sub class of Container. Panel does not contain title bar, menu bar or border.

Frame : The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

AWT Component Hierarchy:-



Java.awt.Frame:-

- ✓ Frame is a container it contains other components like title bar, Button, Text Field...etc.
- ✓ When we create a Frame class object. Frame will be created automatically with invisible mode, so to provide the visible nature to the frame use setVisible() method of Frame class.
public void setVisible(boolean b) where b=true visible mode b=false invisible mode.
- ✓ When we created a frame, the frame is created with initial size 0 pixel heights & 0 pixel width hence it is not visible. To provide particular size to the Frame use setSize() method.
public void setSize(int width,int height)
- ✓ To provide title to the frame use, **public void setTitle(String Title)**
- ✓ When we create a frame, the default background color of the Frame is white. If you want to provide particular color to the Frame we have to use the following method.
public void setBackground(color c)

Example-1:- There are two approaches to create Frame in java

1. By creating object of Frame class.
2. By extending the Frame class.

Approach 1:- Creation of Frame by creating Object of Frame class.

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

Approach 2:-Taking user defined class by extending Frame class.

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

Example : Displaying text on the screen

- ✓ To display some textual message on the frame override `paint()` method.
- public void `paint(Graphics g)`
- ✓ To set a particular font to the text use `Font` class present in `java.awt` package

```
Font f=new Font("String type,int style,int size");
Font f= new Font("arial",Font.Bold,30);

import java.awt.*;
class MyFrame extends Frame
{
    MyFrame()
    {
        setVisible(true);
        setSize(500,500);
        setTitle("myframe");
        setBackground(Color.red);
    }
    public static void main(String[] args)
    {
        MyFrame t = new MyFrame();
    }
    public void paint(Graphics g)
    {
        Font f=new Font("arial",Font.ITALIC,25);
        g.setFont(f);
        g.drawString("hi ratan how r u",100,100);
    }
}
```

- ✓ In above example when we create the `MyFrame` class object ,jvm will executes `MyFrame` class constructor just before this JVM will execute `Frame` class zero argument constructor.
- ✓ The `Frame` class zero argument constructor calling `repaint()` method & this method will access predefined `Frame` class `paint()` method. But as per the requirement overriding `paint()` method will be executed.

Layout Managers:-

When we are trying to add the components into container without using layout manager the components are overriding hence the last added component is visible on the container instead of all.

To overcome above problem to arrange the components into container in specific manner use layout manager.

Definitions:-

The layout managers are used to arrange the components in a Frame in particular manner. or A layout manager is an object that controls the size and the position of components in a container

Different layouts in java,

- 1) `java.awtFlowLayout`
- 2) `java.awt.BorderLayout`
- 3) `java.awt.GridLayout`
- 4) `java.awt.CardLayout`
- 5) `java.awt.GridBagLayout`

java.awt.FlowLayout

The `FlowLayout` is used to arrange the components into row by row format. Once the first row is filled with components then it is inserted into second row. And it is the default layout of the applet.

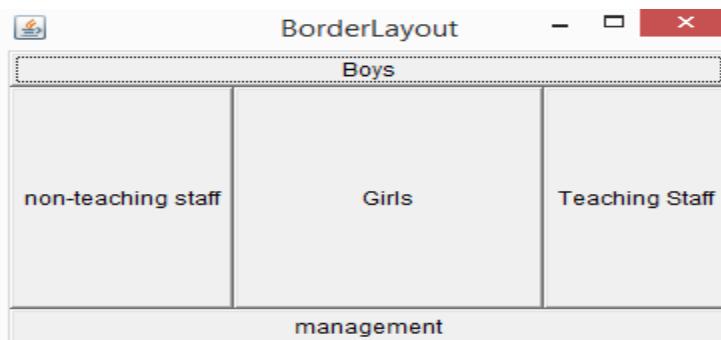
Java.awt.BorderLayout:-

The BoderLayout is dividing the frame into five areas north,south,east,west,center so we can arrange the components in these five areas.

To represent these five areas borderlayout is providing the fallowing 5-consts

```
public static final java.lang.String NORTH;
public static final java.lang.String SOUTH;
public static final java.lang.String EAST;
public static final java.lang.String WEST;
public static final java.lang.String CENTER;

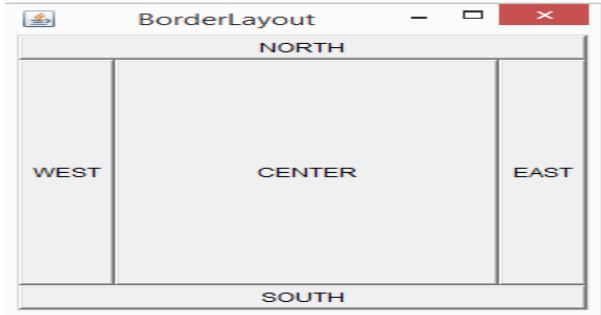
import java.awt.*;
class MyFrame extends Frame
{
    Button b1,b2,b3,b4,b5;
    MyFrame()
    {
        //this keyword is optional because all methods are current class methods only
        this.setSize(400,400);
        this.setVisible(true);
        this.setTitle("BorderLayout");
        this.setLayout(new BorderLayout());
        b1=new Button("Boys");
        b2=new Button("Girls");
        b3=new Button("management");
        b4=new Button("Teaching Staff");
        b5=new Button("non-teaching staff");
        this.add("North",b1);
        this.add("Center",b2);
        this.add("South",b3);
        this.add("East",b4);
        this.add("West",b5);
    }
}
class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};
```



Example 2: project level reduce the length of the code.

```
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f = new Frame("BorderLayout");
        f.setVisible(true);
        f.setSize(300,300);
        f.setLayout(new BorderLayout());

        f.add(new Button("NORTH"),BorderLayout.NORTH);
        f.add(new Button("SOUTH"),BorderLayout.SOUTH);
        f.add(new Button("EAST"),BorderLayout.EAST);
        f.add(new Button("WEST"),BorderLayout.WEST);
        f.add(new Button("CENTER"),BorderLayout.CENTER);
    }
}
```



Preparation of components:-

Label :- Label is a constant text which is displayed along with a TextField or TextArea.

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

TextField :- TextField is an editable area & it is possible to provide single line of text. Enter Button doesn't work on TextField.

To set Text to the textarea : t.setText("Sravya");
To get the text from TextArea : String s=t.getText();

Constructors: TextFiled tx=new TextFiled();
 TextField tx=new TextField("ratan");

TextArea :- TextArea is a Editable Area & enter button will work on TextArea.

Constructors: TextArea t=new TextArea();
 TextArea t=new TextArea(int rows,int columns);
 TextArea t=new TextArea(String text, int rows,int columns);
✓ To set Text to the textarea : ta.setText("Sravya");
✓ To get the text from TextArea : String s=ta.getText();

Button :- Used to perform operations by clicking.

Example :-

```
import java.awt.*;
class MyFrame
{
    MyFrame()
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);
        f.setLayout(new FlowLayout()); //information about layout check next page
        Label l1=new Label("user name:");
        Label l2=new Label("user password:");

        TextField tx1 = new TextField(30);
        TextField tx2 = new TextField(30);

        Button b = new Button("login");

        f.add(l1);
        f.add(tx1);
        f.add(l2);
        f.add(tx2);
        f.add(b);
    }
    public static void main(String[] args)
    {
        MyFrame f = new MyFrame();
    }
}
```

1. **Choice**:-List is allows to select multiple items but choice is allow to select single Item.

Choice ch=new Choice();

- | | |
|-----------------------------------------------------------|-----------------------------------------|
| ✓ To add items to the choice | : add() |
| ✓ To remove item from the choice based on String | : choice.remove("HYD"); |
| ✓ To remove the item based on the index position | : choice.remove(2); |
| ✓ To remove the all elements | : ch.removeAll(); |
| ✓ To inset the data into the choice based on the position | : choice.insert(2,"ratan"); |
| ✓ To get selected item from the choice | : String s=ch.getSelectedItem(); |
| ✓ To get the selected item index number | : int a=ch.getSelectedIndex(); |

2. **List**: List is providinglist of options to select.

CONSTRUCTOR:-

List l=new List(); It will creates the list by default size is four elements.

List l=new List(3); It will display the three items size and it is allow selecting the only one.

List l=new List(5,true); It will display five items and it is allow selecting the multiple items.

- | | |
|------------------------------------------------------|------------------------------------------|
| ✓ To add the elements to the List | : list.add("c"); |
| ✓ To add the elements to the List at specified index | : list.add("ratan",0); |
| ✓ To remove element from the List | : list.remove("c"); |
| ✓ To get selected item from the List | : String x=l.getSelectedItem(); |
| ✓ To get selected items from the List | : String[] x=s.getselectedItems() |

Example :-

```
import java.awt.*;
class Test
{
    public static void main(String[] args)
    {
        Frame f=new Frame();
        f.setVisible(true);
        f.setTitle("ratan");
        f.setBackground(Color.red);
        f.setSize(400,500);
        f.setLayout(new FlowLayout());
        //Choice information
        Choice ch=new Choice();
        ch.add("c");
        ch.add("cpp");
        ch.add("java");
        ch.add(".net");
        ch.remove(".net");
        ch.remove(0);
        ch.insert("ratan",2);
        f.add(ch);
        System.out.println(ch.getItem(0));
        System.out.println(ch.getItemCount());
        //List information
        List l=new List(3,true);
        l.add("c");
        l.add("cpp");
        l.add("java");
        l.add(".net");
        l.add("ratan");
        l.add("arun",0);
        l.remove(0);
        f.add(l);
        System.out.println(l.getItem(0));
        System.out.println(l.getItemCount());
    }
}
```



3. Checkbox: - The user can select more than one checkbox at a time.

```
Checkbox cb2=new Checkbox("MCA");
Checkbox cb3=new Checkbox("BSC",true);
    ✓ To set a label to the CheckBox explicitly : cb.setLabel("BSC");
    ✓ To get the label of the checkbox :String str=cb.getLabel();
    ✓ To get state of the CheckBox : Boolean b=ch.getState();
```

4. RADIO BUTTON:-

- ✓ AWT does not provide any predefined support to create Radio Buttons directly.
- ✓ It is possible to create RadioButton by using two classes.
 - CheckboxGroup
 - Checkbox

step 1:- Create Checkbox group object. **CheckboxGroup cg=new CheckboxGroup();**

step 2:- pass Checkboxgroup object to the Checkbox class argument.

```
Checkbox cb1=new Checkbox("male",cg,true);
Checkbox cb2=new Checkbox("female",cg,false);
    ✓ To get the status of the RadioButton :String str=Cb.getState();
    ✓ To get Label of the RadioButton :String str=getLabel();
```

Example:-

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

        Label l1 = new Label("Qualifications:");
        Label l2 = new Label("Gender:");
        Checkbox cb1=new Checkbox("SSC",true);
        Checkbox cb2=new Checkbox("DEGREE");
        Checkbox cb3=new Checkbox("MCA");
        Checkbox cb4=new Checkbox("BTECH");
        f.add(l1);       f.add(cb1);       f.add(cb2);       f.add(cb3);f.add(cb4);
        System.out.println(cb1.getLabel());
        System.out.println(cb2.getState());

        CheckboxGroup cg=new CheckboxGroup();
        Checkbox r1=new Checkbox("male",cg,true);
        Checkbox r2=new Checkbox("female",cg,false);
        f.add(l2);           f.add(r1);           f.add(r2);
        System.out.println(cb1.getLabel());
        System.out.println(cb1.getState());
    }
}
```



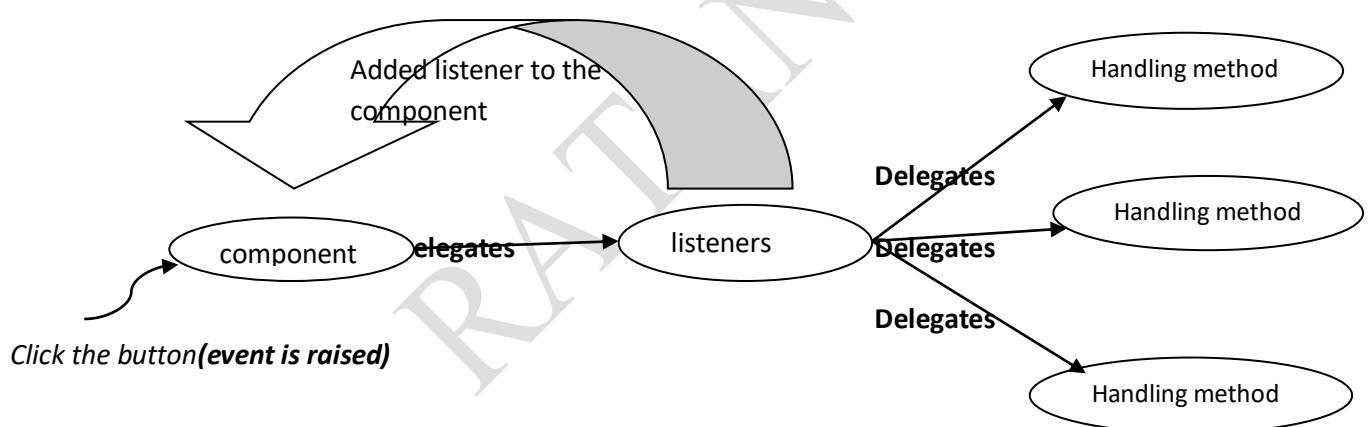
Event delegation model:-

1. When we create a component the components visible on the screen but it is not possible to perform any action on that component because those components are by default static.
- Example:** Whenever we create a Frame it can be minimized and maximized and resized but it is not possible to close the Frame even if we click on Frame close Button. Because frame is a static component so it is not possible to perform actions on the Frame.
2. To make static component into dynamic component we have to add some actions to the Frame. To attach these actions to the Frame component we need event delegation model.

Event: Event is nothing but a particular action generated on the particular component.

1. When an event generates on the component the component is unable to respond because component can't listen the event.
2. To make the component listen the event we have to add listeners to the component. Wherever we are adding listeners to the component the component is able to respond based on the generated event.
3. **java.awt.event** package contains listeners and event classes for event handling.
4. The listeners are different from component to component.

A component delegate event to the listener and listener is designates the event to appropriate method by executing that method only the event is handled. This is called Event Delegation Model.



To attach a particular listener to the Frame we have to use following method

Public void AddxxxListener(xxxListener e)

Where xxx may be ActionListener,windowListener

The Appropriate Listener for the Frame is "windowListener"

ScrollBar:-

1. By using ScrollBar we can move the Frame up and down.
- ScrollBar s=new ScrollBar(int type)

Type of scrollbar

1. VERTICAL ScrollBar
2. HORIZONTAL ScrollBar

To create a HORIZONTAL ScrollBar:-

```
ScrollBar sb=new ScrollBar(ScrollBar.HORIZONTAL);
```

To get the current position of the scrollbar we have to use the following method.

```
public int getValue()
```

To create a VERTICAL ScrollBar:-

```
ScrollBar sb=new ScrollBar(ScrollBar.VERTICAL);
```

Appropriate Listeners for Components:-

GUI Component	Event Name	Listner Name	Lisener Methods
1.Frame	Window Event	Window Listener	1. Public Void WindowOpened(WindowEvent e) 2. Public Void WindowActivated(WindowEvent e) 3. Public Void WindowDeactivated(WindowEvent e) 4. Public Void WindowClosing(WindowEvent e) 5. Public Void WindowClosed(WindowEvent e) 6. Public Void WindowIconified(WindowEvent e) 7. Public Void WindowDeiconified(WindowEvent e)
2.Textfield	ActionEvent	ActionListener	Public Void Actionperformed(ActionEvent ae)
3.TextArea	ActionEvent	ActionListener	Public Void Actionperformed(ActionEvent ae)
4.Menu	ActionEvent	ActionListener	Public Void Actionperformed(ActionEvent ae)
5.Button	ActionEvent	ActionListener	Public Void Actionperformed(ActionEvent ae)
6.Checkbox	ItemEvent	ItemListener	Public Void ItemStatechanged(ItemEvent e)
7.Radio	ItemEvent	ItemListener	Public Void ItemStatechanged(ItemEvent e)
8.List	ItemEvent	ItemListener	Public Void ItemStatechanged(ItemEvent e)
9.Choice	ItemEvent	ItemListener	Public Void ItemStatechanged(ItemEvent e)
10.Scrollbar	AdjustmentEvent	AdjustmentListener	Public Void AdjustmentValueChanged (AdjustementEvent e)
11.Mouse	MouseEvent	MouseListener	1. Public Void MouseEntered(MouseEvent e) 2. Public Void MouseExited(MouseEvent e) 3. Public Void MousePressed(MouseEvent e) 4. Public Void MouseReleased(MouseEvent e) 5. Public Void MouseClicked(MouseEvent e)
12.Keyboard	KeyEvent	KeyListener	1. Public Void KeyTyped(KeyEvent e) 2. Public Void KeyPressed(KeyEvent e) 3. Public Void KeyReleased(KeyEvent e)

Event handling code:-

It is possible to write the event handling code in following ways

In different class.

zSame class

By using anonymous inner classes.

Steps to perform event handling:-

- 1) Prepare the required components.
- 2) Implements the listener interface override the methods to write the even handling code.
- 3) Add the listener to the component.

Applying WindowListener on the Frame :-

The Frame contains WindowLister it contains 7-methods listed below.

windowActivated(WindowEvent e)

Invoked when the Window is set to be the active Window.

windowClosed(WindowEvent e)

Invoked when a window has been closed as the result of calling dispose on the window.

windowClosing(WindowEvent e)

Invoked when the user attempts to close the window from the window's system menu.

windowDeactivated(WindowEvent e)

Invoked when a Window is no longer the active Window.

windowDeiconified(WindowEvent e)

Invoked when a window is changed from a minimized to a normal state.

windowIconified(WindowEvent e)

Invoked when a window is changed from a normal to a minimized state.

windowOpened(WindowEvent e)

Invoked the first time a window is made visible.

In below example we are providing even handling code in separate class :-

PROVIDING CLOSING OPTION TO THE FRAME

```
import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame
{
    MyFrame()
    {
        this.setSize(400,500); //here this keyword is optional because it is a current class
        this.setVisible(true);
        this.setTitle("myframe");
        this.setBackground(Color.green);
        this.addWindowListener(new myclassimpl());
    }
}
class myclassimpl implements WindowListener
{
    public void windowActivated(WindowEvent e)
    {
        System.out.println("window activated");
    }
    public void windowDeactivated(WindowEvent e)
    {
        System.out.println("window deactivated");
    }
    public void windowIconified(WindowEvent e)
    {
        System.out.println("window iconified");
    }
    public void windowDeiconified(WindowEvent e)
    {
        System.out.println("window deiconified");
    }
    public void windowClosed(WindowEvent e)
    {
        System.out.println("window closed");
    }
    public void windowClosing(WindowEvent e)
    {
        System.out.println("window closing");
        System.exit(0);
    }
    public void windowOpened(WindowEvent e)
    {
        System.out.println("window Opened");
    }
};
class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};
```

****PROVIDING CLOSING OPTION TO THE FRAME BY USING WINDOWADAPTOR CLASS ****

In above example when our implements the windowListener interface then we must override all (7)the methods of WindowListener interface .but to close the frame we required only one method that is windowCloseing() method.

To overcome above limitation use WindowAdaptor class it contains all empty implementations of interface methods.

Note : if our class implements WindowListener interface we must override all the methods of windowListener interface . But if our class extends windowAdaptor class it is possible to override application required methods.

```
import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame
{
    MyFrame()
    {
        this.setVisible(true);
        this.setSize(500,500);
        this.addWindowListener(new Listenerimpl());
    }
};
class Listenerimpl extends WindowAdapter
{
    public void windowClosing(WindowEvent we)
    {
        System.exit(0);
    }
};
class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};
```

Writing event handling code in anonymous inner class:-

```
import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame
{
    MyFrame()
    {
        this.setVisible(true);
        this.setSize(500,500);
        this.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {
                System.exit(0);
            }
        });
    }
};
class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};
```

Example :- 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("ratan");
    }
    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);
    }
}
class FrameEx
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};

```

Example : 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();
    }
};

```

Example :- GRIDLAYOUT

```

import java.awt.*;
class MyFrame extends Frame
{
    MyFrame()
    {
        this.setVisible(true);
        this.setSize(500,500);
        this.setTitle("rattaiah");
        this.setBackground(Color.red);
        this.setLayout(new GridLayout(4,4));
        for (int i=0;i<10 ;i++ )
        {
            Button b=new Button(""+i);
            this.add(b);
        }
    }
};

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

```

Example :-ACTIONLISTENER

The below example we are performing addition and multiplications when we click add & mul buttons. So whenever we clicking button the button is able to listen the even to do this add the Listener to button.

The appropriate listener for button is **ActionListener**.

```

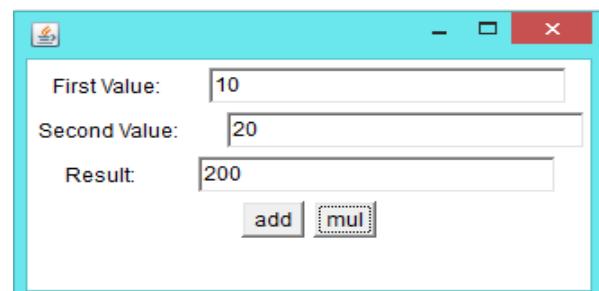
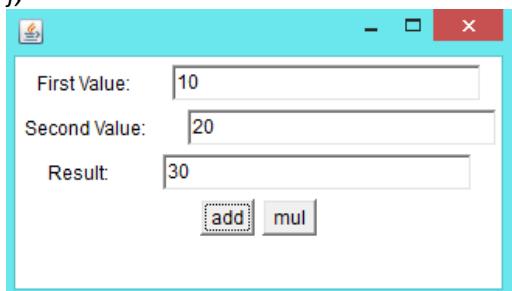
import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame implements ActionListener
{
    TextField tx1,tx2,tx3;
    Label l1,l2,l3;
    Button b1,b2;
    int result;
    MyFrame()
    {
        this.setSize(250,400);
        this.setVisible(true);
        this.setLayout(new FlowLayout());
        l1=new Label("First Value      :");
        l2=new Label("Second Value     :");
        l3=new Label("Result          :");

        tx1=new TextField(25);
        tx2=new TextField(25);
        tx3=new TextField(25);

        b1=new Button("add");
        b2=new Button("mul");
    }
};

```

```
b1.addActionListener(this); // this represent current class object  
b2.addActionListener(this);  
this.add(l1);  
this.add(tx1);  
this.add(l2);  
this.add(tx2);  
this.add(l3);  
this.add(tx3);  
this.add(b1);  
this.add(b2);  
}  
public void actionPerformed(ActionEvent e)  
{  
    try{  
        int fval=Integer.parseInt(tx1.getText());  
        int sval=Integer.parseInt(tx2.getText());  
        String label=e.getActionCommand();  
        if (label.equals("add"))  
        {  
            result=fval+sval;  
        }  
        if (label.equals("mul"))  
        {  
            result=fval*sval;  
        }  
        tx3.setText(""+result);  
    }  
    catch(Exception ee)  
    {  
        ee.printStackTrace();  
    }  
};  
class Demo  
{  
    public static void main(String[] args)  
    {  
        MyFrame f=new MyFrame();  
    }  
};
```



Example :LOGIN STATUS

```
import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame implements ActionListener
{
    Label l1,l2;
    TextField tx1,tx2;
    Button b;
    String status="";
    MyFrame()
    {
        setVisible(true);
        setSize(400,400);
        setTitle("girls");
        setBackground(Color.red);
        l1=new Label("user name:");
        l2=new Label("password:");
        tx1=new TextField(25);
        tx2=new TextField(25);

        b=new Button("login");
        b.addActionListener(this);
        tx2.setEchoChar('*');

        this.setLayout(new FlowLayout());

        this.add(l1);    this.add(tx1);   this.add(l2);
        this.add(tx2);  this.add(b);
    }
    public void actionPerformed(ActionEvent ae)
    {
        String uname=tx1.getText();
        String upwd=tx2.getText();
        if(uname.equals("Sravya")&&upwd.equals("dss"))
        {
            status="login success";
        }
        else
        {
            status="login failure";
        }
        repaint();
    }
    public void paint(Graphics g)
    {
        Font f=new Font("arial",Font.BOLD,30);
        g.setFont(f);
        this.setForeground(Color.green);
        g.drawString("Status:----"+status,50,300);
    }
}
class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};
```

Example :

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

Example :- 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();
    }
};
```

Example : 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 Demo
{
    public static void main(String[] args)
    {
        myframe f=new myframe();
    }
};

Example :-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();
    }
};

```

Example : 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();
    }
};

```

Example :- *AdjustmentListener*

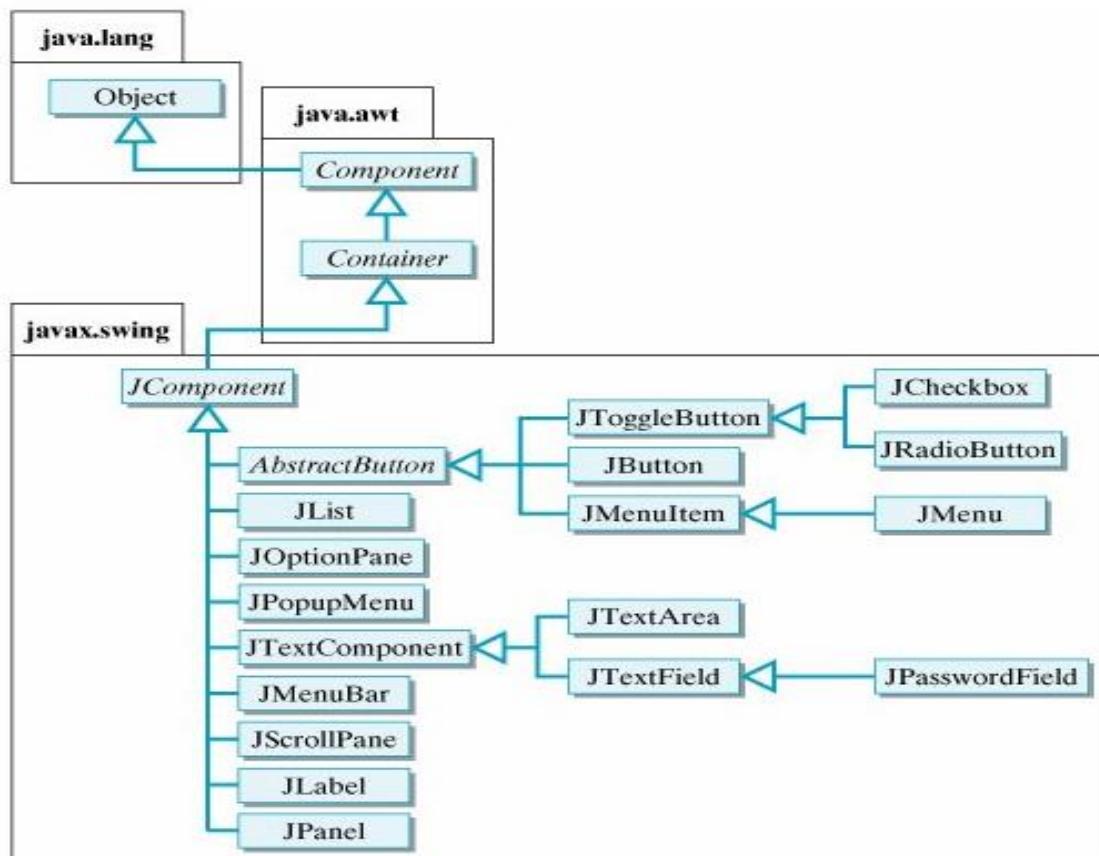
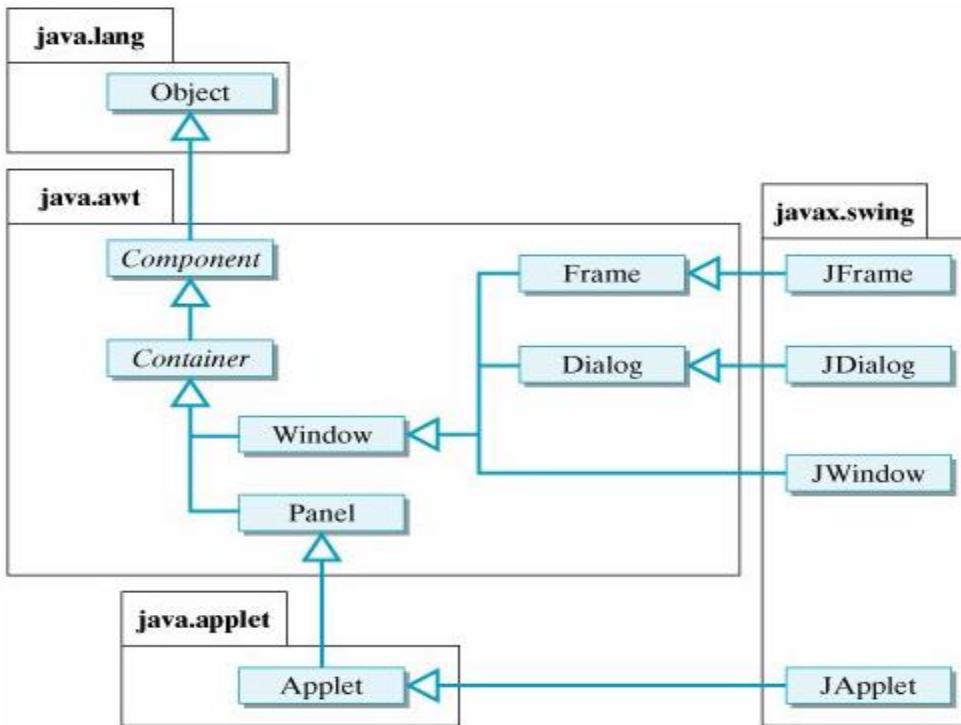
```
import java.awt.*;
import java.awt.event.*;
class myframe extends Frame implements AdjustmentListener
{
    Scrollbar sb;
    int position;
    myframe()
    {
        this.setSize(400,400);
        this.setVisible(true);
        this.setLayout(new BorderLayout());
        sb=new Scrollbar(Scrollbar.VERTICAL);
        this.add("East",sb);
        sb.addAdjustmentListener(this);
    }
    public void adjustmentValueChanged(AdjustmentEvent e)
    {
        position=sb.getValue();
    }
    public void paint(Graphics g)
    {
        g.drawString("position:"+position,100,200);
        repaint();
    }
}
class scrollbarex
{
    public static void main(String[] args)
    {
        myframe f=new myframe();
    }
}
```

SWINGS

1. Sun Micro Systems introduced AWT to prepare GUI applications but awt components not satisfy the client requirement.
2. An alternative to AWT Netscape Communication Corporation has provided set of GUI components in the form of IFC(Internet Foundation Class) but IFC also provide less performance and it is not satisfy the client requirement.
3. In the above context[sun &Netscape] combine and introduced common product to design GUI applications is called JFS(java foundation classes) then it is renamed as swings.

Differences between awt and Swings:

- ✓ AWT components are platform dependent but Swings components are platform independent because these components are completely written in java .
- ✓ AWT components are heavyweight component but swing components are light weight component.
- ✓ AWT components consume more number of system resources Swings consume less number of system resources.
- ✓ AWT is provided less number of components whereas swing provides more number of components.
- ✓ AWT doesn't provide Tooltip Test support but swing components have provided Tooltip test support.
- ✓ In awt to close the window : windowListenerwindowAdaptor
In case of swing use small piece of code.
`f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`
- ✓ In case of AWT we will add the GUI components in the Frame directly but Swing we will add all GUI components to panes to accommodate GUI components.
- ✓ The awt classes & interfaces are present in java.awt packages & swing classes are present in javax.swing package.



Example :-

```
package swingss;

import java.awt.Color;
import java.awt.FlowLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
public class MyFrame extends JFrame{
    public MyFrame() {
        setVisible(true);
        setSize(300, 300);
        setTitle("MyFrame");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        JLabel label = new JLabel("Hello World:");
        JButton button = new JButton("click me");
        add(label);
        add(button);
    }
    public static void main(String[] args) {
        MyFrame f = new MyFrame();
    }
}
```

Example 2:-

```
package swingss;
import java.awt.*;
import javax.swing.*;
class Test2 extends JFrame
{
    JLabel l1,l2,l3,l4,l5,l6,l7;
    JTextField tf;
    JPasswordField pf;
    JCheckBox cb1,cb2,cb3;
    JRadioButton rb1,rb2;
    JList l;
    JComboBox cb;
    JTextArea ta;
    JButton b;
    Container c;
    Test2()
    {
        setVisible(true);
        setSize(150,500);
        setTitle("SWING GUI COMPONENTS EXAMPLE");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        getContentPane().setBackground(Color.red);
```

```
setBackground(Color.green);
l1=new JLabel("User Name");
l2= new JLabel("password");
l3= new JLabel("Qualification");
l4= new JLabel("User Gender");
l5= new JLabel("Technologies");
l6= new JLabel("UserAddress");
l7= new JLabel("comments");
tf=new JTextField(15);
tf.setToolTipText("TextField");
pf=new JPasswordField(15);
pf.setToolTipText("PasswordField");
cb1=new JCheckBox("BSC",false);
cb2=new JCheckBox("MCA",false);
cb3=new JCheckBox("PHD",false);
rb1=new JRadioButton("Male",false);
rb2=new JRadioButton("Female",false);
ButtonGroup bg=new ButtonGroup();
bg.add(rb1);           bg.add(rb2);
String[] listitems={"cpp","c","java"};
l=new JList(listitems);
String[] cbitems={"hyd","pune","bangalore"};
cb=new JComboBox(cbitems);
ta=new JTextArea(5,20);
b=new JButton("submit");
add(l1);
add(tf);      add(l2);      add(pf);
add(l3);      add(cb1);      add(cb2);      add(cb3);
add(l4);      add(rb1);      add(rb2);      add(l5);
add(l);       add(l6);       add(cb);       add(l7);
add(ta);      add(b);
}
public static void main(String[] args)
{
    Test2 f=new Test2();
}
};
```



Example 3:

```
package swingss;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;

import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MyFrame extends JFrame{
    public MyFrame() {
        setVisible(true);
        setSize(200, 200);
        setTitle("MyFrame");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new BorderLayout());//setting layout to frame

        JLabel label = new JLabel(new ImageIcon("F:\\corejava images\\download.jpg"));
        add(label);
        label.setLayout(new FlowLayout());// setting layout to back ground image

        JLabel label2 = new JLabel("Hello World:");
        JButton button = new JButton("click me");
        label.add(label2);
        label.add(button);
    }
    public static void main(String[] args) {
        MyFrame f = new MyFrame();
    }
}
```



Example 4:-

```

package swingss;

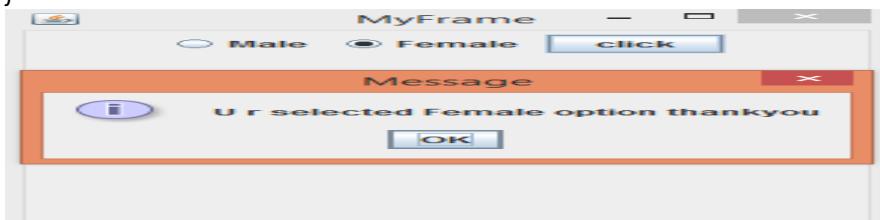
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JRadioButton;
public class MyFrame2 extends JFrame implements ActionListener {
    JRadioButton button1, button2;
    JButton button;
    public MyFrame2() {
        setVisible(true);
        setTitle("MyFrame");
        setSize(300, 300);
        setLayout(new FlowLayout());

        button1 = new JRadioButton("Male");
        button2 = new JRadioButton("Female");
        button = new JButton("click");

        ButtonGroup group = new ButtonGroup();
        group.add(button1);
        group.add(button2);
        add(button1);           add(button2);           add(button);
        button.addActionListener(this);
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        if(button1.isSelected())
        {   JOptionPane.showMessageDialog(this, "U r selected Male option thankyou");}
        if(button2.isSelected())
        {   JOptionPane.showMessageDialog(this, "U r selected Female option thankyou");}
    }
    public static void main(String[] args) {
        MyFrame2 frame2 = new MyFrame2();
    }
}

```



Example 5:

```
package swingss;

import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
public class MyFrame3 extends JFrame {
    public MyFrame3() {
        setVisible(true);
        setSize(400, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        String[][] data={{"101","ratan","10000"}, {"102","anu","20000"}, {"101","durga","30000"}};
        String[] col={"Eid","Ename","Esal"};
        JTable jTable = new JTable(data,col);
        add(jTable);

        JScrollPane jScrollPane = new JScrollPane(jTable);
        add(jScrollPane);
    }
    public static void main(String[] args)
    {
        MyFrame3 frame3 = new MyFrame3();
    }
}
```



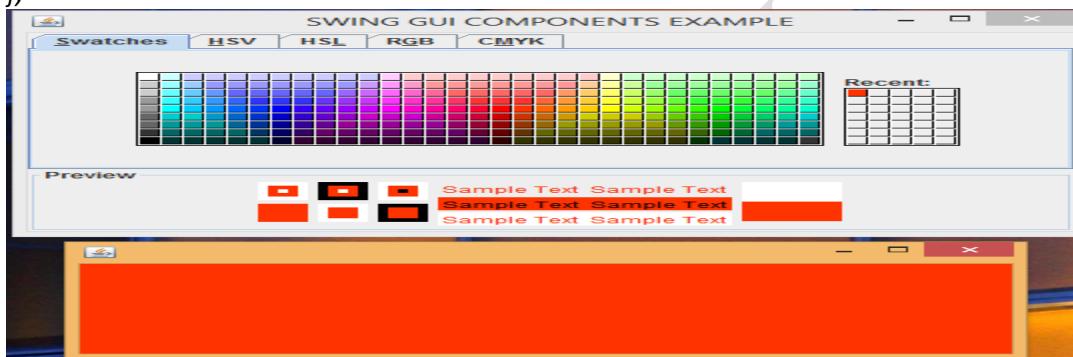
Application 6: - JCOLORCHOOSEN

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

```

        c=getContentPane();
        cc=new JColorChooser();
        cc.getSelectionModel().addChangeListener(this);
        c.add(cc);
    }
    public void stateChanged(ChangeEvent c)
    {
        Color color=cc.getColor();
        JFrame f=new JFrame();
        f.setSize(400,400);
        f.setVisible(true);
        f.getContentPane().setBackground(color);
    }
}
class Demo
{
    public static void main(String[] args)
    {
        MyFrame f=new MyFrame();
    }
};

```



Application 7: JFILECHOOSER

```

import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
class MyFrame extends JFrame implements ActionListener
{
    JFileChooser fc;
    Container c;
    JLabel l;
    JTextField tf;
    JButton b;
    MyFrame()
    {
        this.setVisible(true);
        this.setSize(500,500);
        this.setTitle("SWING GUI COMPONENTS EXAMPLE");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        c=getContentPane();
        l=new JLabel("Select File:");

```

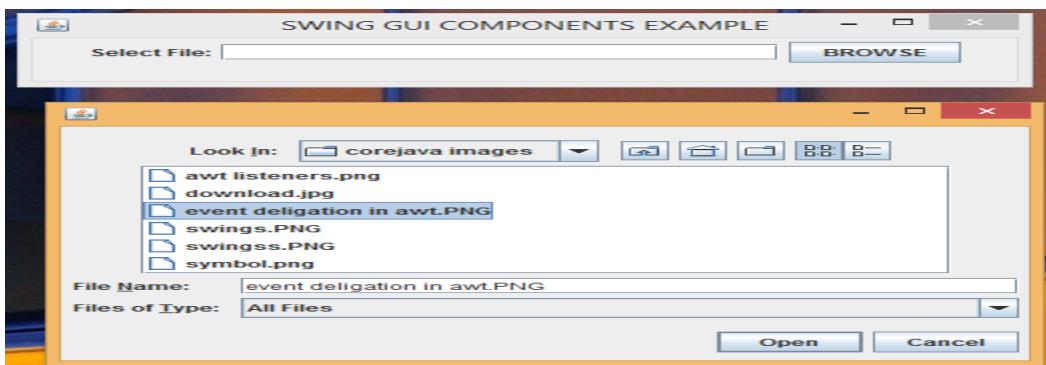
```
tf=new JTextField(25);
b=new JButton("BROWSE");
this.setLayout(new FlowLayout());
b.addActionListener(this);
c.add(l);           c.add(tf);           c.add(b);
}

public void actionPerformed(ActionEvent ae)
{
    class FileChooserDemo extends JFrame implements ActionListener
    {
        FileChooserDemo()
        {
            Container c=getContentPane();
            this.setVisible(true);
            this.setSize(500,500);
            fc=new JFileChooser();
            fc.addActionListener(this);
            fc.setLayout(new FlowLayout());
            c.add(fc);
        }

        public void actionPerformed(ActionEvent ae)
        {
            File f=fc.getSelectedFile();
            String path=f.getAbsolutePath();
            tf.setText(path);
            this.setVisible(false);
        }
    }
    new FileChooserDemo();
}

}

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



Java.awt.Applet

- ✓ The applet is runs on browser window to display the dynamic content on browser window.
- ✓ The applet does not contains main methods to start the execution but it contains life cycle methods these methods are automatically called by web browser.
- ✓ To run the applet in browser window we need to install plugin in.

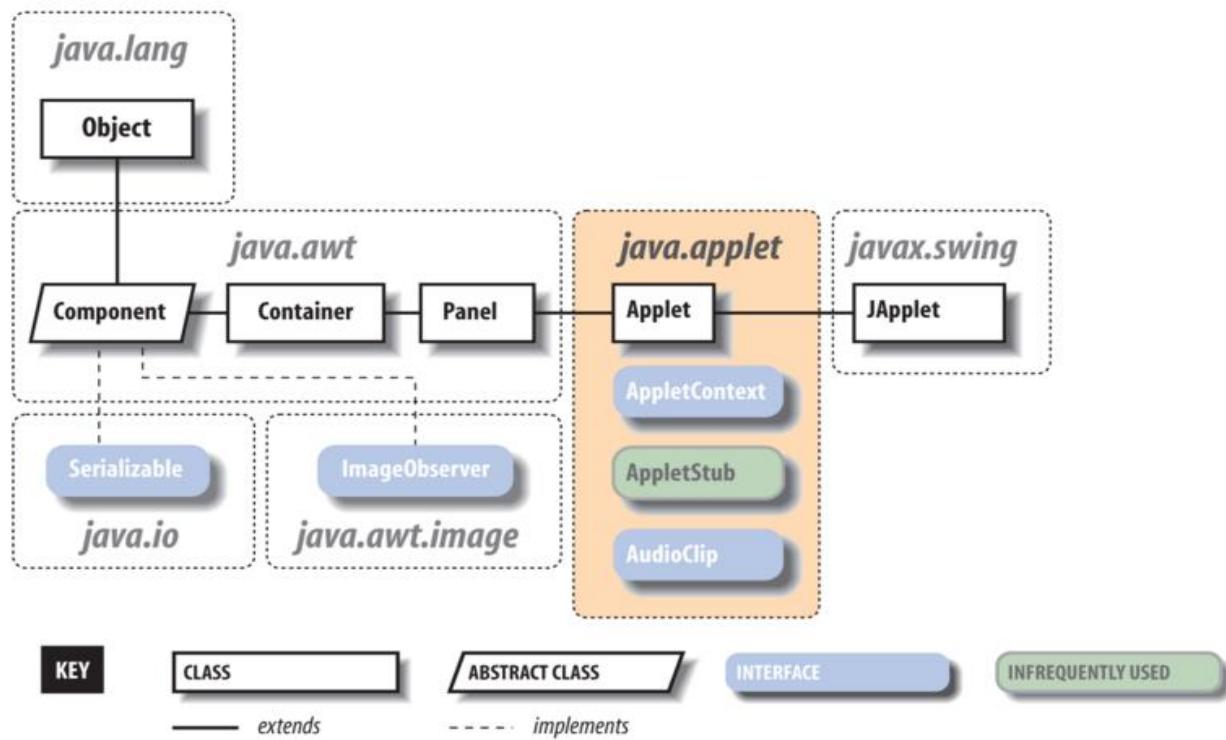


Figure 23-1. The `java.applet` package

The applet contains four life cycle methods

- a. `Init()`
- b. `Start()`
- c. `Stop()`
- d. `Destroy()`

These methods are called by applet viewer or web browser

`Init()` : used to initialize the applet and it called only once.

`Start()` : it invoked after `init()` method to start the applet then the applet become visible.

`Stop()` : it is used to stop the applet then the applet become invisible.

`Destroy()`: it called after `stop` method it gives to applet last chance to cleanup.

`Java.awt.Component` class provide one life cycle method of applet

`Public void paint(Graphics g)`

The above method used to paint the applet to print the data in applet.

Steps to design the application:-

1. Create the applet by extending **Applet** class.
2. Then configure the applet in html code.

Applet First Application:-**Test.java:-**

```
import java.applet.Applet;
import java.awt.*;
public class Test extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello, world this is applet first example!", 150, 150);
    }
}
```

Firstapplet.html

```
<html>
<body>
<applet code="Test.class" height="300" width="300"/>
</body>
</html>
```

Execution : appletviewer Firstapplet.html

**Running applet:-**

It is possible to run the applet in two ways

- 1) By using html file

Configure the applet in html file then open the html file in browser window.
Click on **Firstapplet.html** then the applet is displayed on browser window.

- 2) By using applet viewer tool

Run the application by using below command

Appletviewer Firstapplet.html

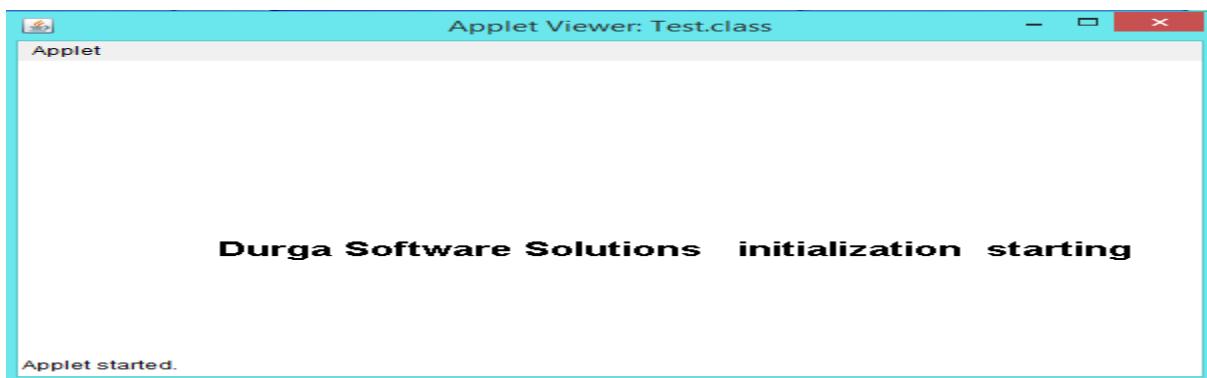
Application 2:*Test.java:-*

```
import java.awt.*;
import java.applet.*;
public class Test extends Applet
{
    String msg="";
    public void paint(Graphics g)
    {
        Font f=new Font("arial",Font.BOLD,20);
        g.setFont(f);
        g.drawString("Durga Software Solutions "+msg,100,200);
    }
    public void init()
    {
        msg=msg+"initialization"+ " ";
        System.out.println("init()");
    }
    public void start()
    {
        msg=msg+"starting"+ " ";
        System.out.println("start()");
    }
    public void stop()
    {
        msg=msg+"stoping";
        System.out.println("stop()");
    }
    public void destroyed()
    {
        msg=msg+"destroyed";
        System.out.println("destroy()");
    }
};
```

Configuration of Applet:-

```
<html>
<applet code="Test.class" width="500" height="500">
</applet>
</html>
```

Execution : appletviewer Firstapplet.html

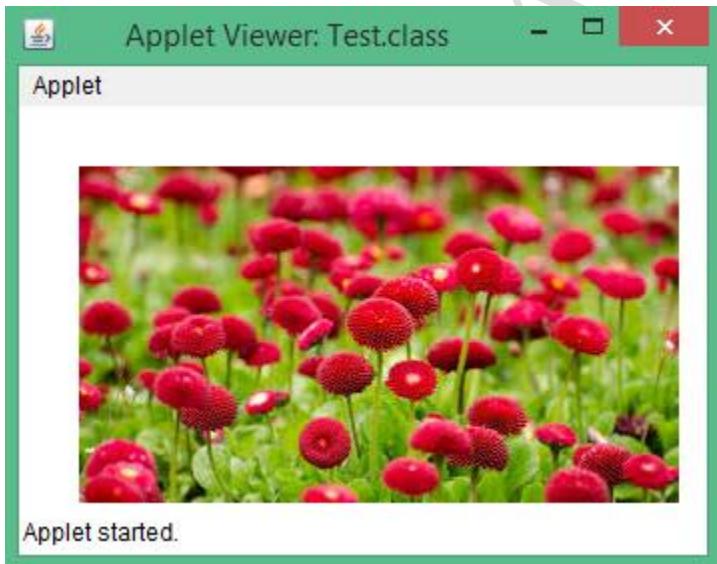


Application 3: displaying image on applet.**Test.java:**

```
import java.awt.*;
import java.applet.*;
public class Test extends Applet {
    Image picture;
    public void init() {
        picture = getImage(getDocumentBase(), "flower.jpg");
    }
    public void paint(Graphics g) {
        g.drawImage(picture, 30, 30, this);
    }
}
```

Firstapplet.html

```
<html>
<body>
<applet code="Test.class" height="300" width="300"></applet>
</body>
</html>
```

Execution : appletviewer Firstapplet.html

Different types of methods in java (must know information about all methods)

- | | |
|------------------------------|-----------------------------|
| 1) Instance method | 12) Overridden method |
| 2) Static method | 13) Instance Factory method |
| 3) Normal method | 14) Static factory method |
| 4) Abstract method | 15) Pattern factory method |
| 5) Accessor methods | 16) Template method |
| 6) Mutator methods | 17) Default method |
| 7) Inline methods | 18) Public method |
| 8) Call back methods | 19) Private method |
| 9) Synchronized methods | 20) Protected method |
| 10) Non-synchronized methods | 21) Final method |
| 11) Overriding method | 22) Strictfp method |
| | 23) Native method |

Different types of classes in java (must know information about all classes)

- | |
|--------------------------------------------------|
| 1) Normal class /concrete class /component class |
| 2) Abstract class /helper class |
| 3) Tightly encapsulated class |
| 4) Public class |
| 5) Default class |
| 6) Adaptor class |
| 7) Final class |
| 8) Strictfp class |

- 9) JavaBean class /DTO(Data Transfer Object)
/VO (value Object)/BO(Business Object)
- 10) Singleton class
- 11) Child class
- 12) Parent class
- 13) Implementation class

Different types of variables in java

(must know information about all variables)

- | | |
|--------------------------|---------------------------|
| 1) Local
variables | 6) Protected
variables |
| 2) Instance
variables | 7) Volatile
variables |
| 3) Static
variables | 8) Transient
variables |
| 4) Final
variables | 9) Public
variables |
| 5) Private
variables | |



Material by Mr. Ratan.....



Write a Review



About your RATAN sir.....!

THE PAIN YOU FEEL TODAY
WILL BE THE
STRENGTH
YOU FEEL TOMORROW

