

Core Java

By

Sateesh Gupta

Naresh Technology

SRI RAGHAVENDRA XEROX

Software Languages Material Available

Beside Bangalore Ayyangar Bakery, Opp. C DAC, Ameerpet, Hyderabad.

Cell: 9951596199

N
S.G.

Introduction

Date - 3/09/12

Rs:- 170

Core Java

Q) What is Java?

→ Java is a very simple, high-level, secured, concurrent, platform independent, object oriented programming language.

→ Java is a technology which provides —

a. Language

b. Platform

Q) What is a platform?

→ Platform is a hardware & software environment which provides runtime environment for applications or programs.

→ This runtime environment consists of —

1. Memory management

2. Process management

3. I/O management

4. Device management

→ Platform which acts as mediator or interface between software & hardware.

Q) What is the abbreviation of Java?

→ There is no abbreviation of Java. The Development Team of Java just chosen this name.

→ The name Java specifically does not have any meaning rather it refers to the hot, aromatic drink COFFEE. This is the reason Java programming language icon is coffee cup.

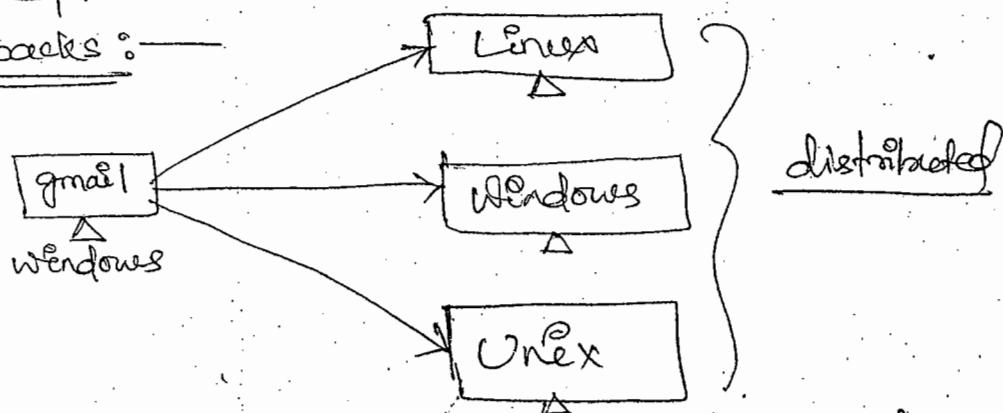
Q) What is meant by platform dependent & platform independent application?

Platform dependent: —

→ An application that is compiled on one OS & then unable to run in another OS, then that application is called platform dependent.

→ A program which is prepared (compile) on one OS can be execute on same OS but cannot be executed on other OS, becoz once we compiled this it generates machine code which is specific for a machine one which it was prepared.

Drawbacks: —



→ After compilation of the program, it generates machine code which is platform dependent code.

Platform Independent: —

4/9/12

→ If the application's compiled code is able to run in another Operating System then that application is called platform independent appl'.

→ The program which is compiled ~~on one OS~~, it should ^{not} generate machine code.

→ When this program is compiled it generates a intermediate language code (byte code) which does not have any instruction related to real machine OS.

→ Byte code generates becoz of platform independency.

→ Platform independency is required for distributed application.

→ It is required a software which is responsible to convert the intermediate language into machine code.

Q What is Bytecode?

- A compiled code of Java source program is called bytecode.
- It is an intermediate language code, which is a portable code means portable across multiple OS.
- Bytecode is a collection of mnemonics.
(mnemonics → ADD, MOV).
- The name is given as bytecode bcoz every byte code in computer memory occupy "one byte".

Q Diff. betⁿ bytecode & machine code?

| <u>bytecode</u> | <u>machine code</u> |
|--|--|
| (1) It is a portable code. | (1) It is a non-portable code. |
| (2) These codes are collection of mnemonics. | (2) These codes are collection of 1's & 0's. |
| (3) This code is platform independent code. | (3) This code is platform dependent code. |
| (*) | |

Q. What is the java file's extension?

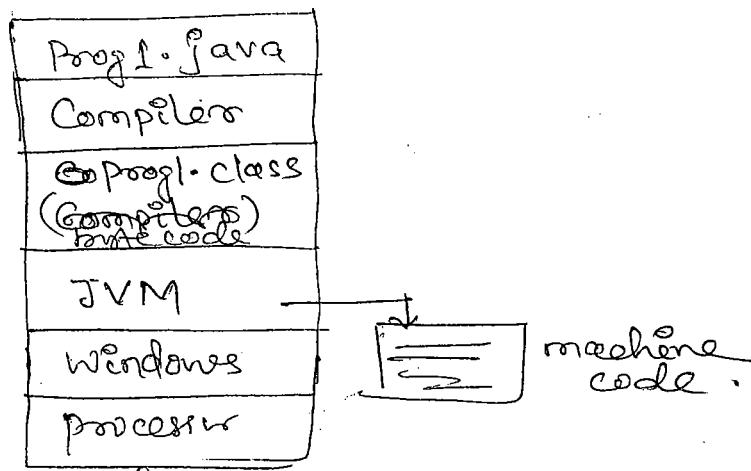
1. Source code → ".java" → developer written code
2. Compiled code → ".class" → the bytecode generated by compiler

→ Generally ".exe" contain machine code which is understand by machine contain JVM (Java virtual machine).

→ Computer is responsible for converting source code into bytecode and JVM is responsible for converting bytecode into machine code.

Java Virtual Machine (JVM)

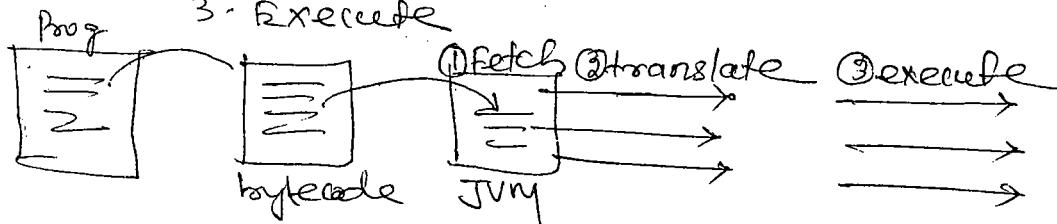
5/09/12



- JVM is developed by SUN separately for every OS, not by OS vendor.
- JVM is available for all OS separately. But JVM is not install automatically, we must install JVM in our computer for executing Java program bytecodes.
- Java is platform independent, Java Software is platform dependent. Java Software is known as JVM which is provided by Java.
- JVM provides runtime environment for Java applications/programs.
- JVM is a software in C, C++, bcoz of this C & C++ are platform dependent.
- JVM provides a translator for converting bytecode into machine code or executable code.
 1. Interpreter
 2. Hotspot compiler / JIT (just-in-time) Compiler

Interpreter → Interpreter translates byte code into machine code line by line.

- It performs 3 operations
 1. fetch (Reading)
 2. translate
 3. Execute

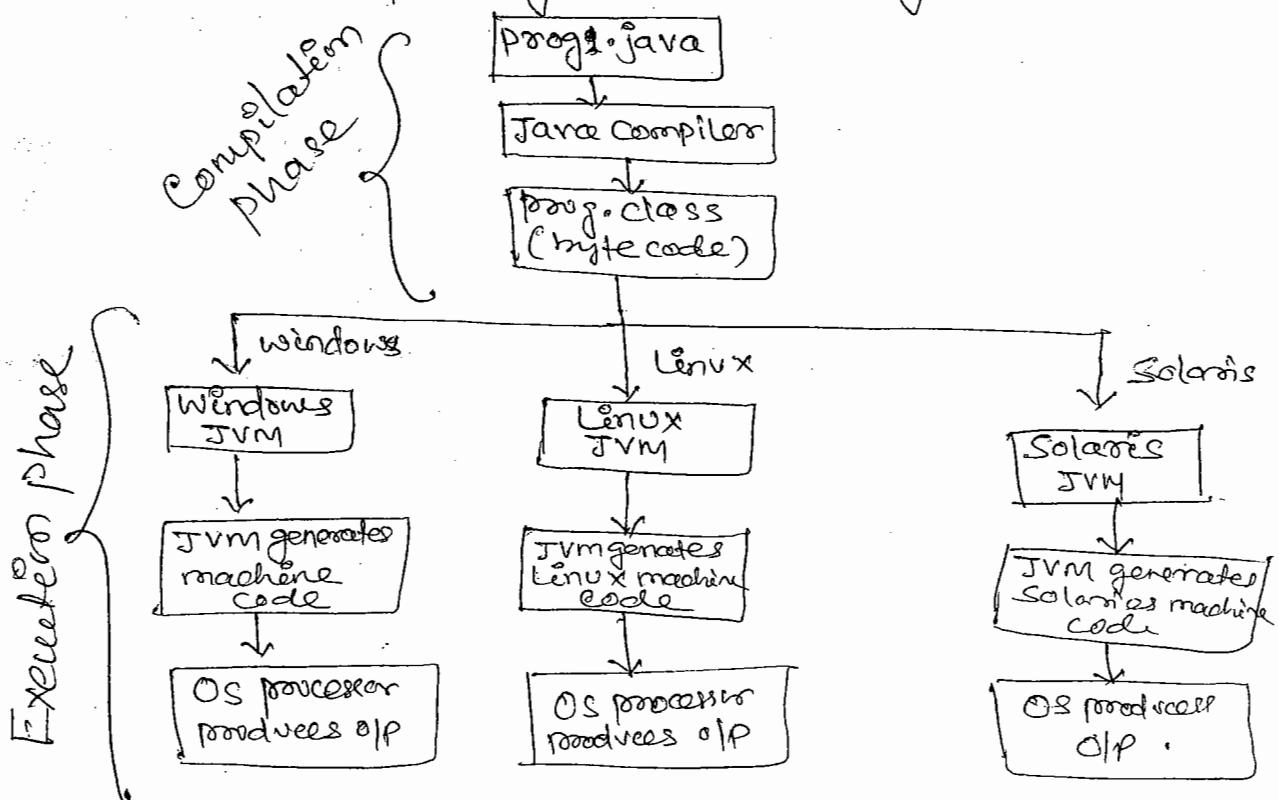


JIT: (Just in time compiler / HOT-Spot)

- It is provided by JVM, which translates bytecode into machine code.
- It performs 4 operations -
 1. fetch.
 2. Translate
 3. Store
 4. execute.

Q. Java is a platform dependent, justify?

- The compiled code of java is platform independent, but platform (JVM) provided by java is dependent.
- Java software is platform dependent and Java program becomes platform independent becoz it generates common bytecode which is run in every machine (OS) as JVM is separately installed in every machine.



Java Features:

- 1. Simple
- 2. Platform independent
- 3. portable
- 4. object oriented
- 5. Architecture
- 6. Dynamic
- 7. Robust
- 8. Secured
- 9. Multithreading
- 10. Distributed.

1. Simplicity:

- Java simplifies the programmer's job by avoiding explicit memory management.
- Java provides automatic memory management (garbage collector), which removes memory which is not in use.

ex:-

```
void main()
{
    fun1();
    fun2();
}
```

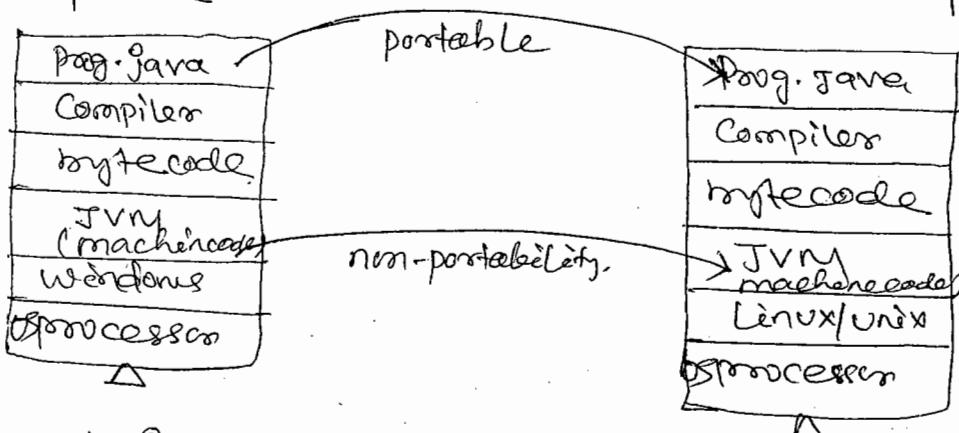
void fun1()
{
 int *p;
 p = malloc(10);
}
void fun2()
{
 int *q;
 q = malloc(20);
}

Q) What is memory leak?

- The memory reserved by program unable to reuse or use that memory is called leaked from program. This leads to wasteage of memory.
- Java simplifies by avoiding (most) pointers, there is no dereferencing operators in Java.

2. Portability:

- Moving instructions written in one language from one operating environment to another operating environment is called portability.
- Portability allows to develop program irrespective of hardware.



C, C++, Java → portable language.

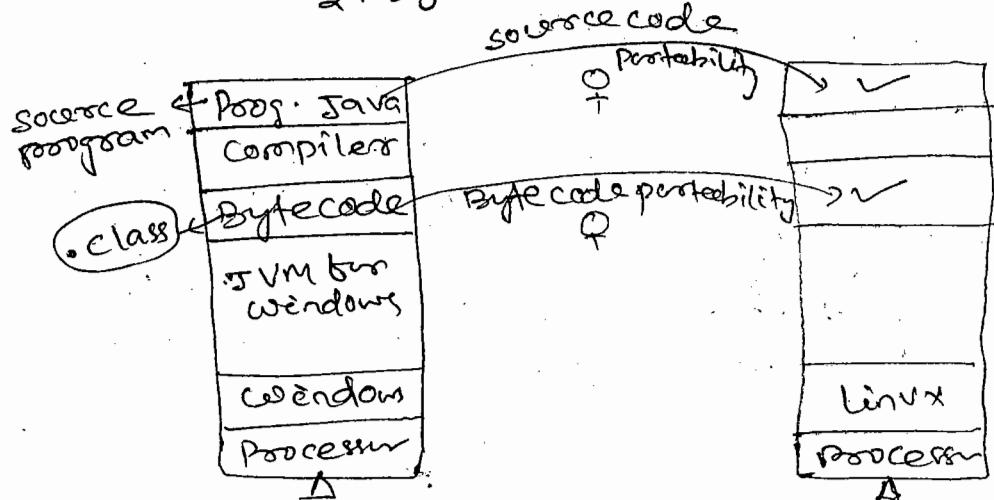
sourcecode, bytecode of Java → portable

machinecode of Java → non-portable.

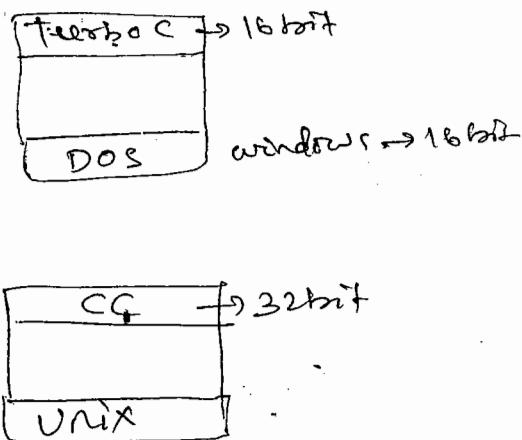
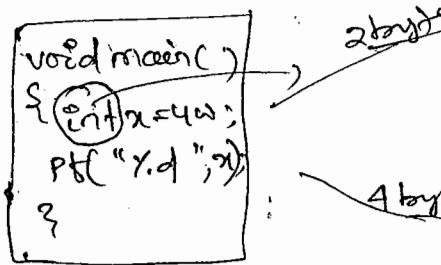
Portability :-

Java provides 2 type of portability -

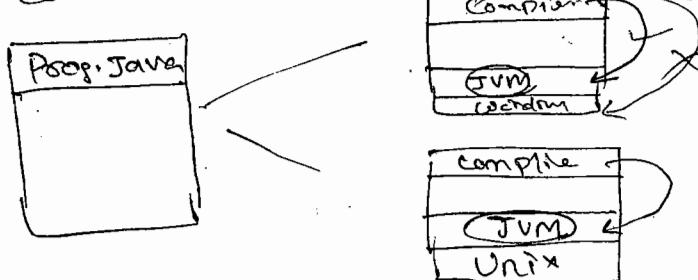
1. Source code portability
2. Bytecode portability.



Architecture :-



- The behaviour of java program does not change from one system to another system.
- JVM is a specification or set of rules which are common for all operating systems.

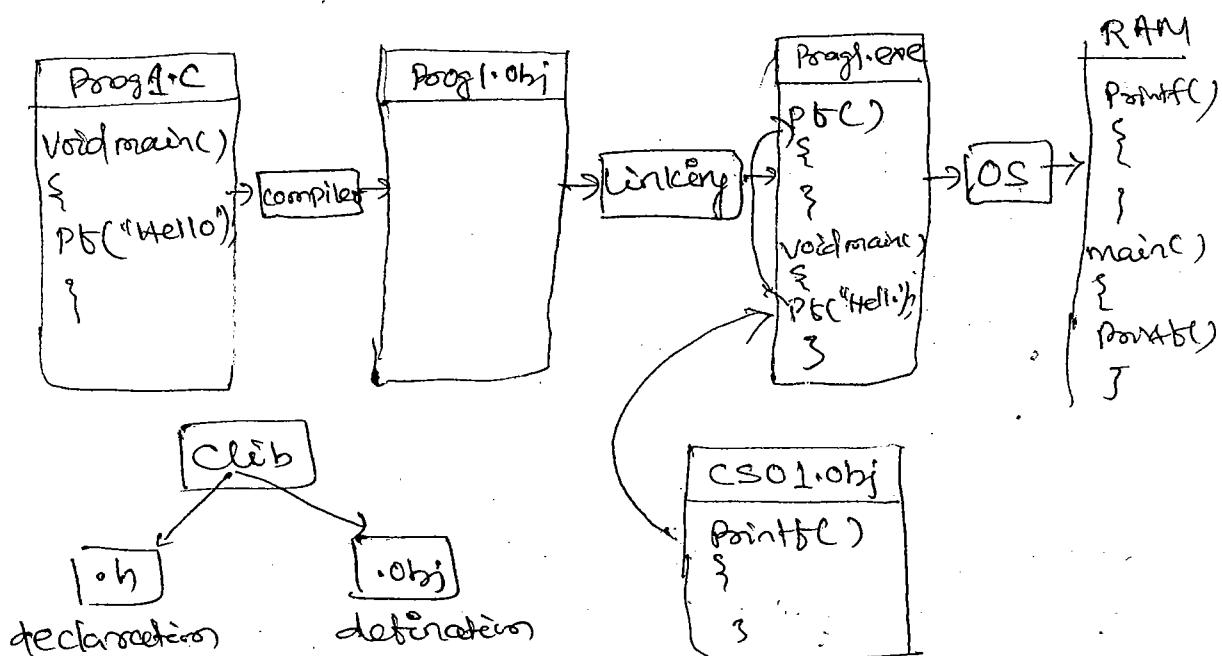


Dynamic :-

- Loading or Linking of programs are 2 types.
1. static loading
 2. Dynamic loading

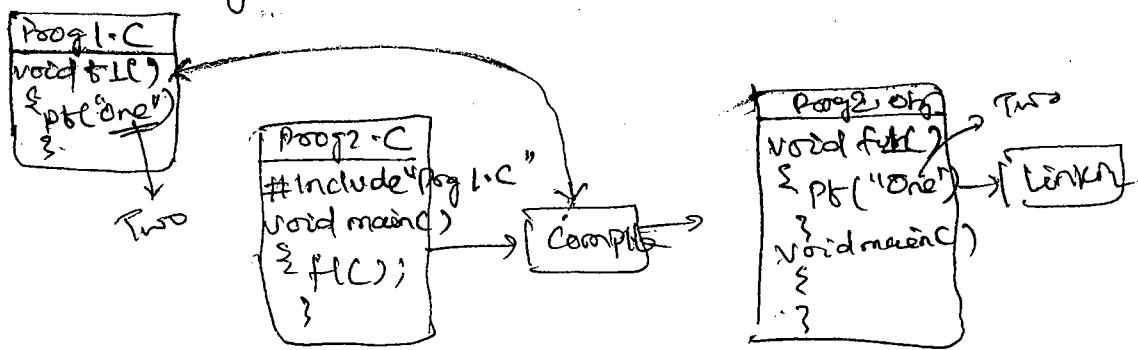
Static loading :-

- Loading of all executable blocks before executing program is called static loading.
- Disadvantage of it, that if a small change in a program then we compile the entire program.



Drawbacks:-

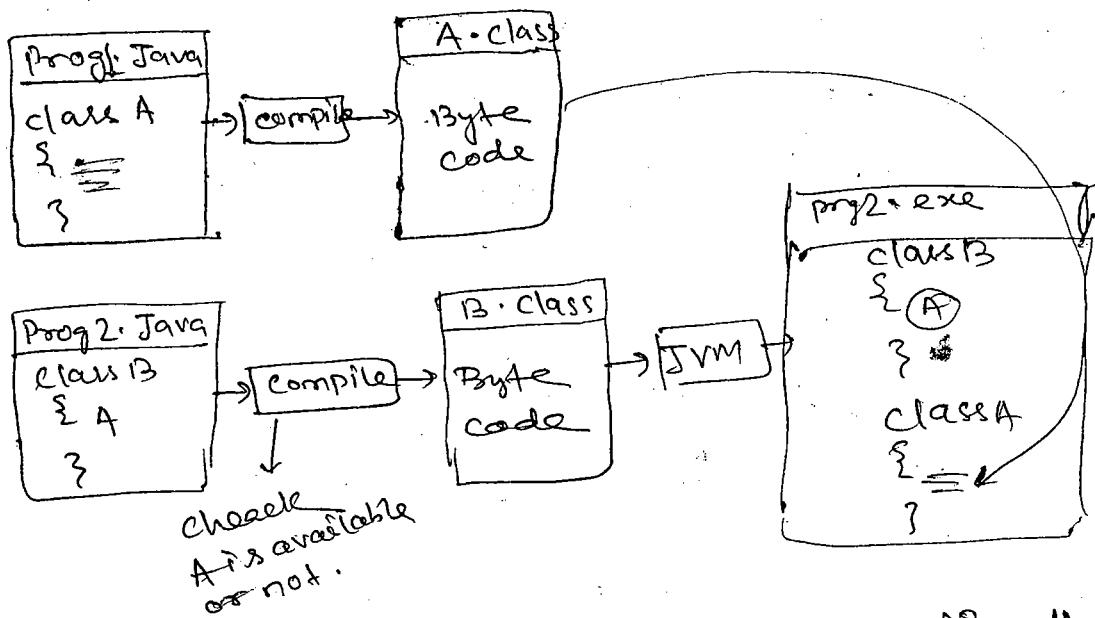
1. Any small change in one program required to recompile all the programs, which uses that executable block.
2. Required more space, which decrease efficiency of the program.



Dynamic Loading:

→ Loading & linking of executable blocks during execution of program is called dynamic loading.

Ex:-



→ The main advantage of dynamic loading that any small change in one program doesn't required to recompile all the programs.

Robust:

- Java is a strongly typed language.
- It is having strict type checking during compile time & runtime.
- The errors which are recognized by compiler are called syntax errors.
- The errors which are recognized by run-time (JVM) are called logical errors.

Ex:-

```

void main()
{
    int x
    y
    z
}
    
```

$z = x + y;$ → logical error ~~soft~~

printf("%d", z);

}

Security:-

- Pointers are not scattered below it manipulate address.
- Java restricts pointer operations.
- There is no pointer arithmetic in java.
- The pointers available in java are called references.
- These references can't manipulate address.

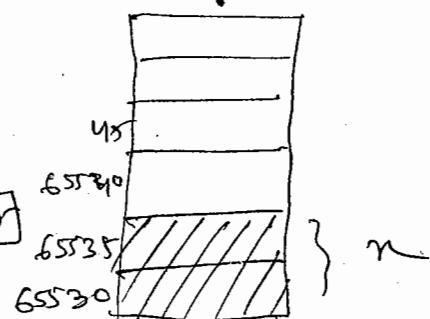
RN:

int *P;

int x = 10;

P = &x; 65531

P = P - 1;



→ in java this $(P = P - 1)$ not done.

Date - 27/09/2012

Multithreading:-

Application

single tasking

multitasking
(To utilize CPU utilization)

web app → multithreaded.

→ Multithreading allows to develop a multi-tasking applications.

→ A process is an instance of a program and simultaneous execution multiple processes is called process based multitasking.

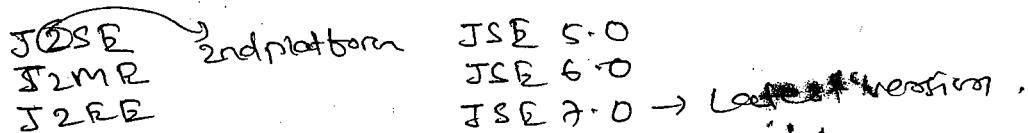
→ A thread is an instance of a process or a process under execution simultaneous execution is called thread based multi-tasking.

Distributed:

- It allows to develop networking application.
- It provides set of pre-defined protocols which allows to develop distributed applications.

Java Editions:

1. JSE → Java Standard Edition. (web-supportive appn)
2. JME → Java micro Edition. (device appn)
3. JEE → Java Enterprise Edition. (web appn)



JSE:

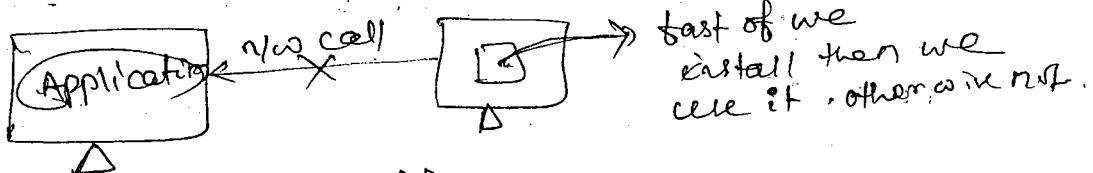
- It allocates us 3 types of applications -

- (a) Standalone application.
- (b) Client - server application.
- (c) web - supporting application.

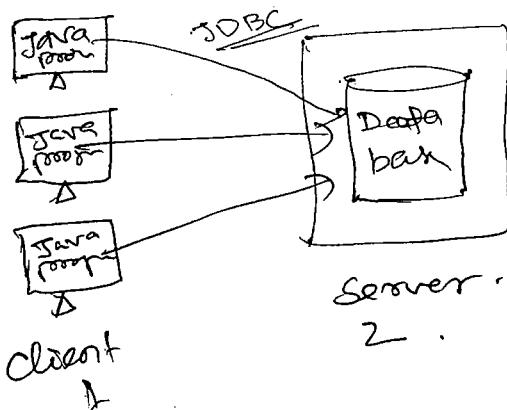
Standalone Application:

- An application whose resources cannot be shared by more than one user is called "Standalone application".

- These applications can be desktop applications or console based applications.



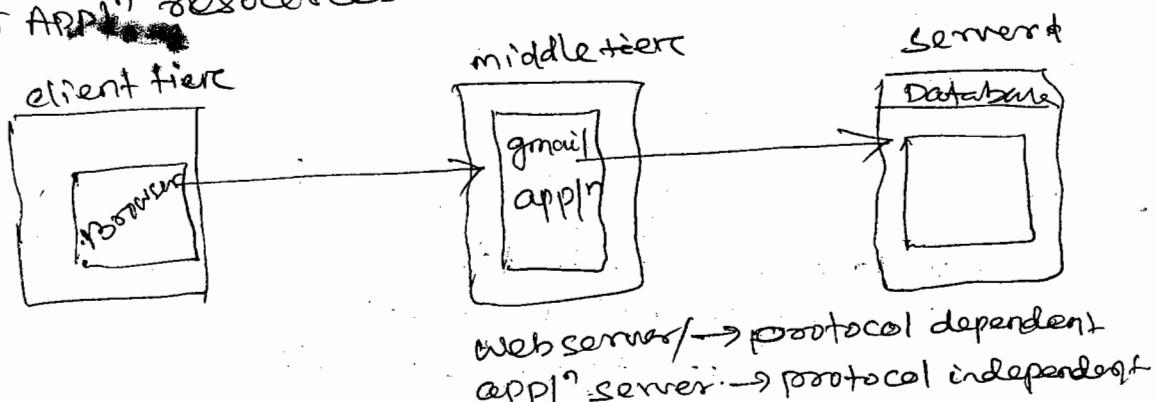
Client - server application:



- A client is a java program which communicate with server (database).
- Applet is an embedded program which is embedded within HTML page (webpage).
- It is downloaded along with webpage & executed in client browser.

JEE:

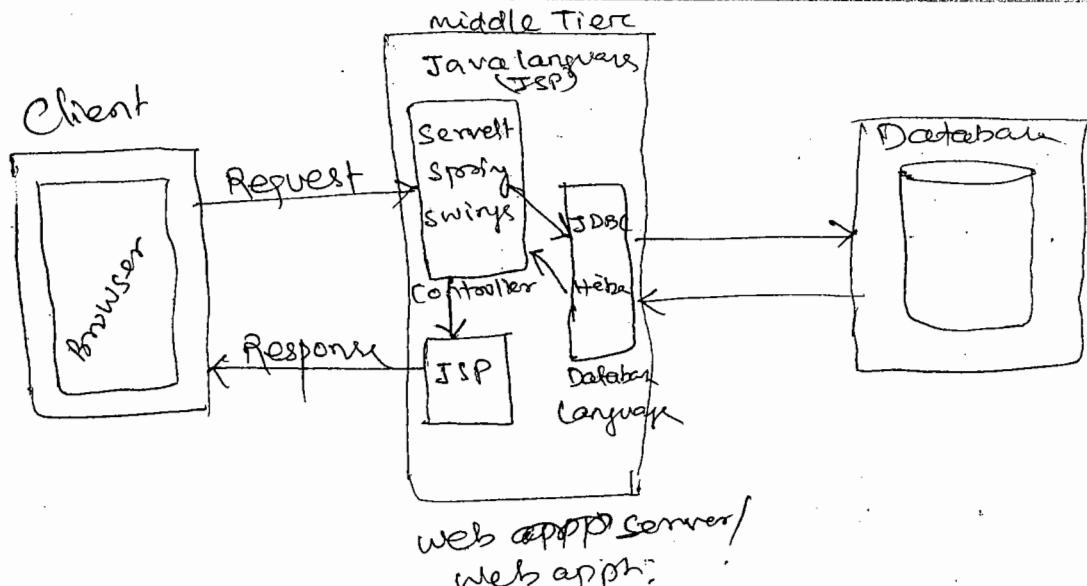
- All "resources" are shared by 'n' no. of clients.
- ↪ Application resources are shared by 'n' no. of clients.



(3 tier architecture)

- ↪ JEE consists of JDBC, JSP, servlets, JSP, EJB, Spring, Hibernate, Streets, XML, webservices etc all are called API (application programming interface).

- ↪ An application whose resources shared by more than one client is called enterprise application.
- All enterprise appl' are called enterprise applications.
- Every enterprise appl' is called web application.



JME :

- This is used for developing device app's.
- The applications which are written using C & C++ can be developed with JME.

Java Versions :

- X → A small change within the edition is called version.

Code Name

| | | |
|---------------------------------|------|------------|
| Java 1.0 | 1995 | Oak |
| Java 1.1 | 1997 | Oak |
| Java 1.2 | 1999 | Playground |
| also called as Java 2 platform. | | |
| Java 1.3 | 2000 | Rosewood |
| Java 1.4 | 2002 | Merlion |
| Java 5 | 2004 | Tiger |
| Java 6 | 2006 | Mustang |
| Java 7 | 2009 | Dolphin |

- J2SE 1.2] → no change in (languages, <sup>collection
of
Keywords</sup>)
 J2SE 1.3] i.e. change library,
 J2SE 1.4]
- JSE 5.0] → There is a change in language.
 JSE 6.0
 JSE 7.0

Java 5 Features:

→ The extra features of Java are —

1. Auto boxing & unboxing.
2. Enhanced for loop.
3. Generics.
4. Varargs.
5. Annotations.
6. static import.
7. Enums.

Date - 8/09/2012

Java History:

→ Java was created in 1991 by James Gosling of Sun micro system. Initially called Oak. Its name was changed to Java becoz there was already a language called Oak.

Note:-

Now Java is a product of ORACLE corporation.

→ The original motivation for Java is they need for a platform independent language that could be embedded in various consumer electronic products like toasters and refrigerators.

Java Software:

→ Java software is a platform dependent.
→ In order to develop Java applications, a system should have a software called JSDK (Java Software Development Kit).

Person want to
develop a
Java project
(development)
(JDK)

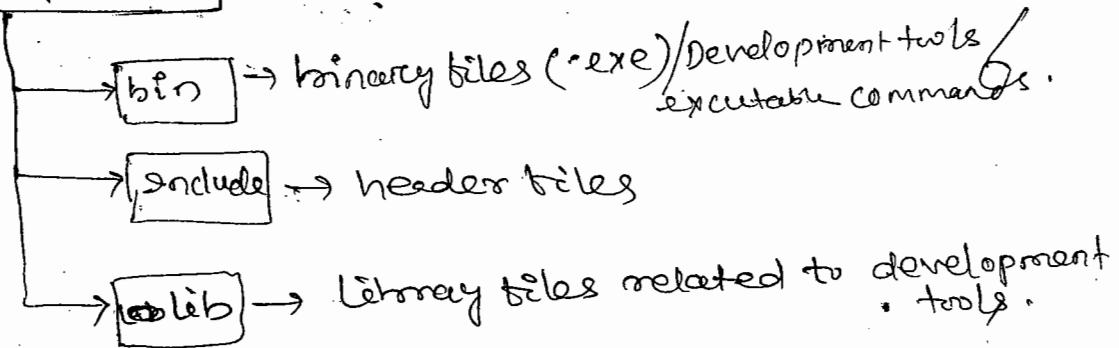
JDK
(Java development
Kit)

↓ → (Developers
of end user)
JRE (Java runtime
environment)
Person wants to execute
the program.

JDK:

→ JDK is the Java Development Kit used by java developer/programmer in order to develop java projects/applications.

JDK 1.6.0



javac.exe → java compiler

javadoc.exe → java documentation generator

java.exe → java interpreter

javah.exe → java header file generator

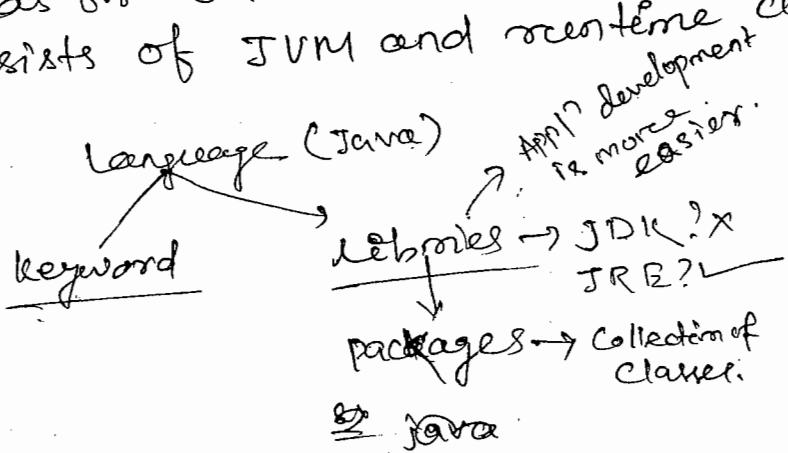
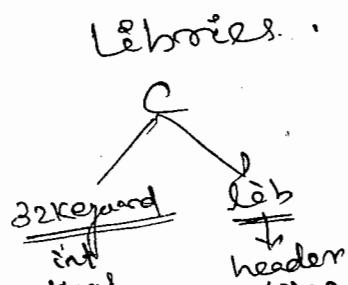
jdb.exe → java debugger

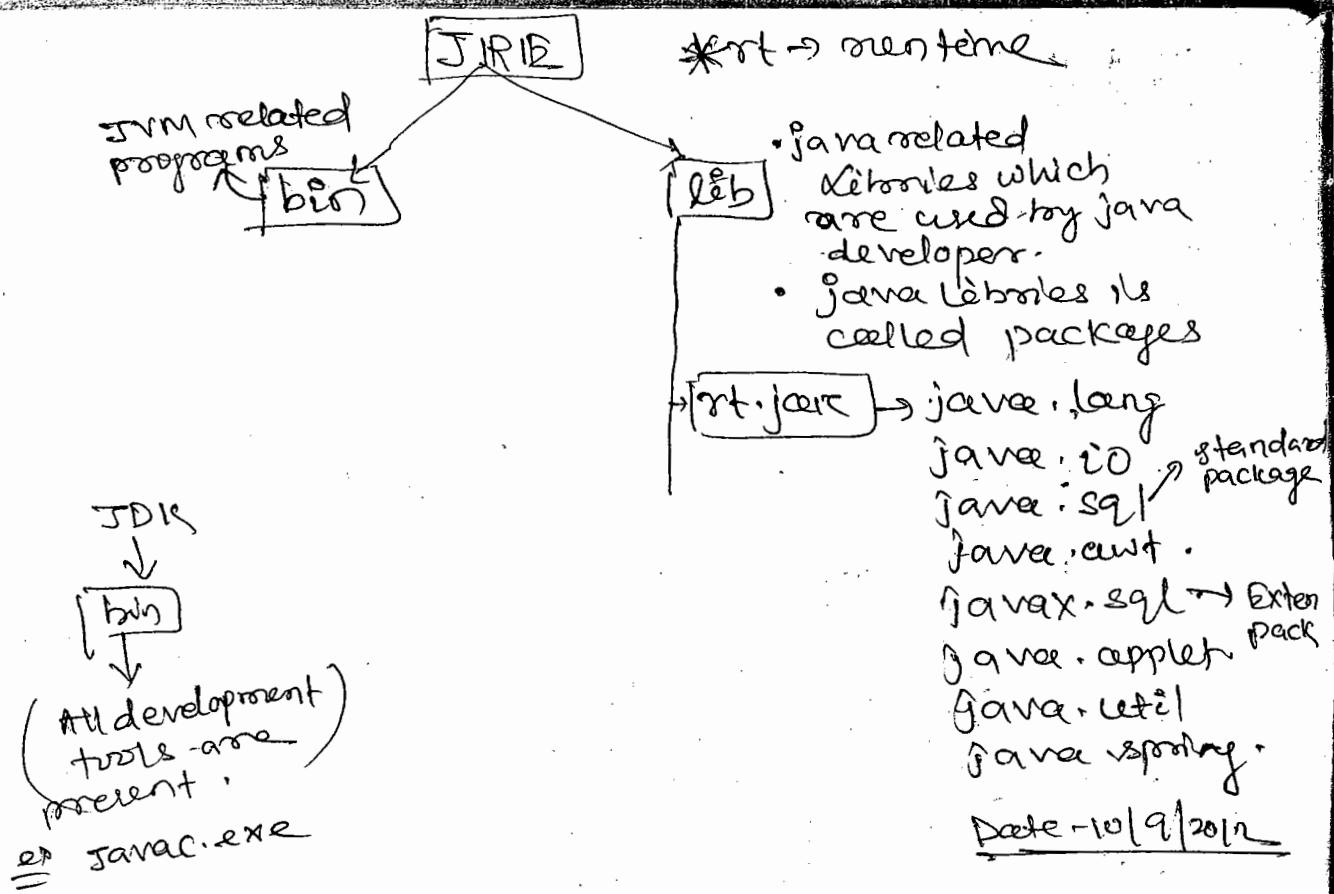
jar.exe → Java archive utility

javadoc.exe → java document generator

JRE:

→ JRE stands for Java Runtime environment, which consists of JVM and runtime class libraries.





Character Set of Java:

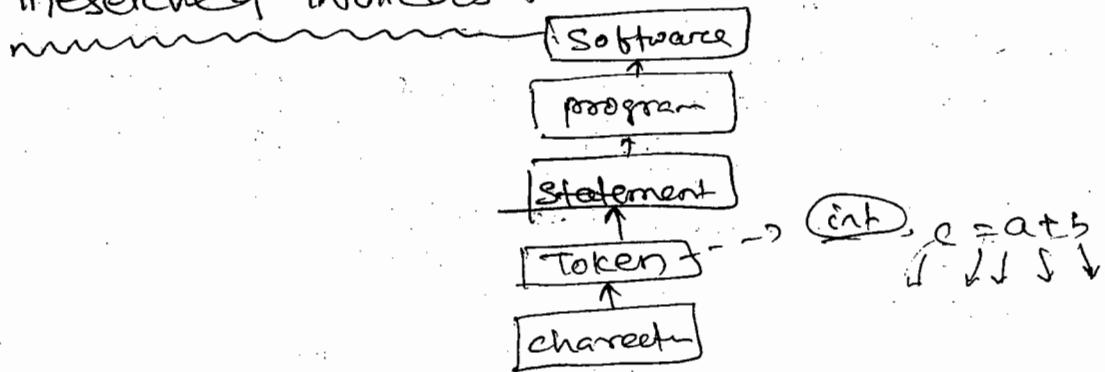
- Java is a universal language.
- It not only supports characters available in English. It also supports characters available in other languages.
- Java uses a UNICODE character set.
- According to Java there is 65536 characters available in Java.
- * → In Java character takes 2 bytes.

Characters

UNICODE

| | | |
|---|---|-----|
| A | → | 65 |
| B | → | 66 |
| C | → | 67 |
| : | | |
| Q | → | 97 |
| b | → | 98 |
| c | → | 99 |
| : | | |
| Z | → | 300 |
| z | → | 301 |

Reserved Words:



Indirect
Storage

External
package

→ Java language related words are called "reserved words".

→ These words are recognized by Java compiler & Java runtime (JVM).

→ Reserved words are 2 types -

→ Reserved words (52)

Total (52)

Reserved words (50)

(50)

Keywords

Used (48)
Keywords

Unused (2)
Keywords

③
Reserved (3)
(literals)

constant.

True true
false false
null null.

goto → consequence of unstructured programs.

Const (final is replaced by const in Java).

Used keywords

Data types ①

- ① int
- ② short
- ③ long
- ④ float
- ⑤ char
- ⑥ double
- ⑦ boolean
- ⑧ void
- ⑨ byte

Control statements ⑫

- ⑩ if
- ⑪ else
- ⑫ if else
- ⑬ switch
- ⑭ case
- ⑮ default
- ⑯ for
- ⑰ while
- ⑲ do
- ⑳ break
- ㉑ continue
- ㉒ return

modifiers ⑪

- ① private
- ② public
- ③ protected
- ④ abstract
- ⑤ final
- ⑥ static
- ⑦ transient
- ⑧ synchronized
- ⑨ volatile
- ⑩ strictfp
- ⑪ native

Exception

- ⑫ try
- ⑬ catch
- ⑭ finally
- ⑮ throw
- ⑯ throws
- ⑰ assert (1-4)

| <u>Packages</u> | <u>class related type/</u> | <u>Inheritance</u> | <u>References</u> |
|-----------------|----------------------------|--------------------|--------------------------|
| ① import | ① class | ① extends | ① This |
| ② package | ② interface | ② implements | ② Super ③ instance of |
| | ③ enum(1.5) | | |

* Null is a reserved literals.

Identifiers:

- Identifier is a user-defined word.
- Identifier is a collection of alphabets, digits,
- & allows a special characters (i.e. \$, -).
- The identifiers contain certain rules called "Hungarian notation".

1. Class name should start with Capital letters.
2. If class name having multiple words separate with capital letters.

Ex:- Account { String name
Saving Account { System.out.println("Hello");
CURRENT Account {
Deposite

2. Method (function) name should start with small letter, if method having multiple words separated by Capital letters.

Ex:- point()
printReport()

3. Variable name should start with small letter. If variable name having multiple words separated with capitals.

Ex:- point, accno, CustomerName.

4. Constant name should be given in Capital letters. If constant name having multiple words separated by underscore (-).

Ex:- PI, MAX-RANGE,
MAX-VALUE

Q.

Q

Q

150

70

Q Identify the following identifiers?

Student → variable

Rollno → class

getRollno() → function/method

MAX_RNO → Constant

Q Identify valid identifier?

goto → invalid

transient → invalid

true → invalid

TRUE → valid (coz java is case sensitive)
so tend the difference.)

Q Identify reserved keywords?

this → no

instanceof → no

true → Yes

null → Yes

Data types in Java

→ Java data types are classified into 2 categories.

1. Primitive datatypes. (value type)

2. Reference datatypes. (address type)

→ The main purpose of Data types is that what kind of value we can stored in to the variable.

Primitive datatypes:

→ A variable of type primitive datatype hold values.

→ The primitive datatypes are → "signed".

→ Every datatype in Java is by default "signed".

① int → 4 bytes

② short → 2 bytes

③ long → 8 bytes

④ byte → 1 byte

⑤ double → 8 bytes
6 digits

7 digits
⑥ float → 4 bytes

{
⑦ Boolean → 1 bit
⑧ void → empty datatype
⑨ char → 2 bytes

Reference Datatypes :-

- A variable of type hold address
- Reference variable is restricted pointer like variable.
- The reference datatypes are —

1. class
2. interface
3. array
4. enum

Date-11/09/2012

Q. Why char in Java is 2 byte?
A. The character in Java is 2 byte bcoz in Java UNICODE char is used so to allocate memory for UNICODE character we required 2 byte memory.
In ASCII code 65 to 90 is 1 byte bcoz it uses ASCII code.

Structure of Java Program :-

Java Program is divided into 4 sections.

- 1) Document Section
- 2) Package Statement
- 3) import statement
- 4) class definition section

Document Section :-

In document Section content information about the program.

This section is ignored by compiler.
Documentation is provided using comments.

Java supports 3 types of comments —

- 1) Single-line comment (//)
- 2) Multi-line comment ((/* */))
- 3) Documented comment ((/** */))

Ex. /**

company :
Title :-
Project :-
Date :-
*/

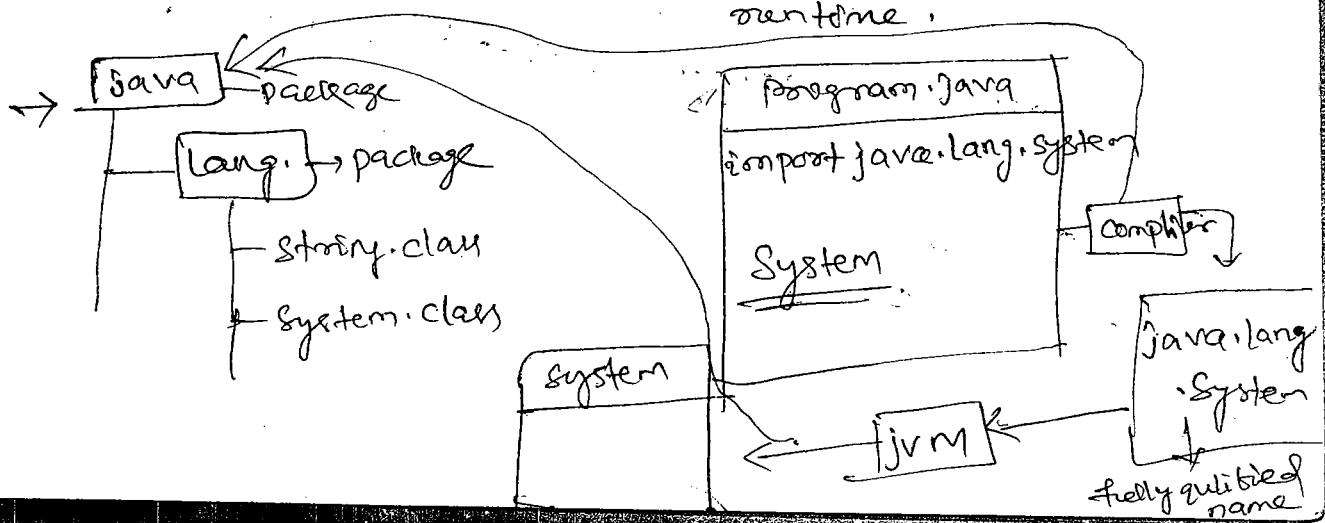
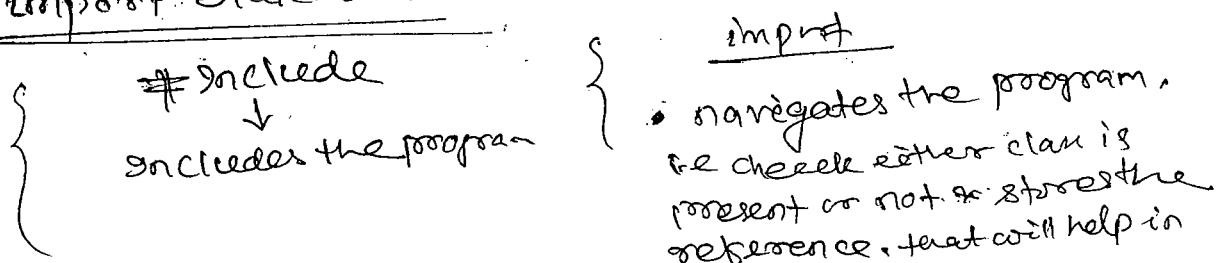
Advantage
(we can separate it from
others.)

- Documented comments can be separated from source program to build document file (.html).
- This building of document file is done using utility called "javadoc".
- Documented comments are optional.

Package Statement:

- Package is a collection of types. These types include classes, interfaces or enum.
- Package is a directory which contains "class" files.
- Package provides @ namespace management & (B) access protection (i.e. Security).
- One source program allows ^{only} one package statement.
- For a beginner's package may or may not compulsory but for a developer it is compulsory.

Import Statement:



- Import is used for using the contents of existing package.
- Import does not include class, but add package reference. It adds name of the package to it belongs.
- A Java program can have multiple import statements.
- The default package import by any Java program is "java.lang".
- Ex:
 import package-name.* // more than one;
 import package-name.class-name; // for a single class;

Q) Can we use content of package without using package statement?

Ans Yes, using fully qualified name (classname along with the package name).

Class definition Section:

- As the term in Java encapsulated inside the class so it is called "method".
- Every Java executable program must have a class which contains "main()" method.
- Class have 2 type,
 - ① Executable
 - ② Non-executable
- Q Class is a datatype & that is user defined datatype.

First program in Java:

Steps:-

- ① Open any text editor (notepad)
- ② Write java program
- ③ Save with an extension ".java"
- ④ Compiling java program

Steps for it -

- ① Open command prompt
start → run → cmd → OK

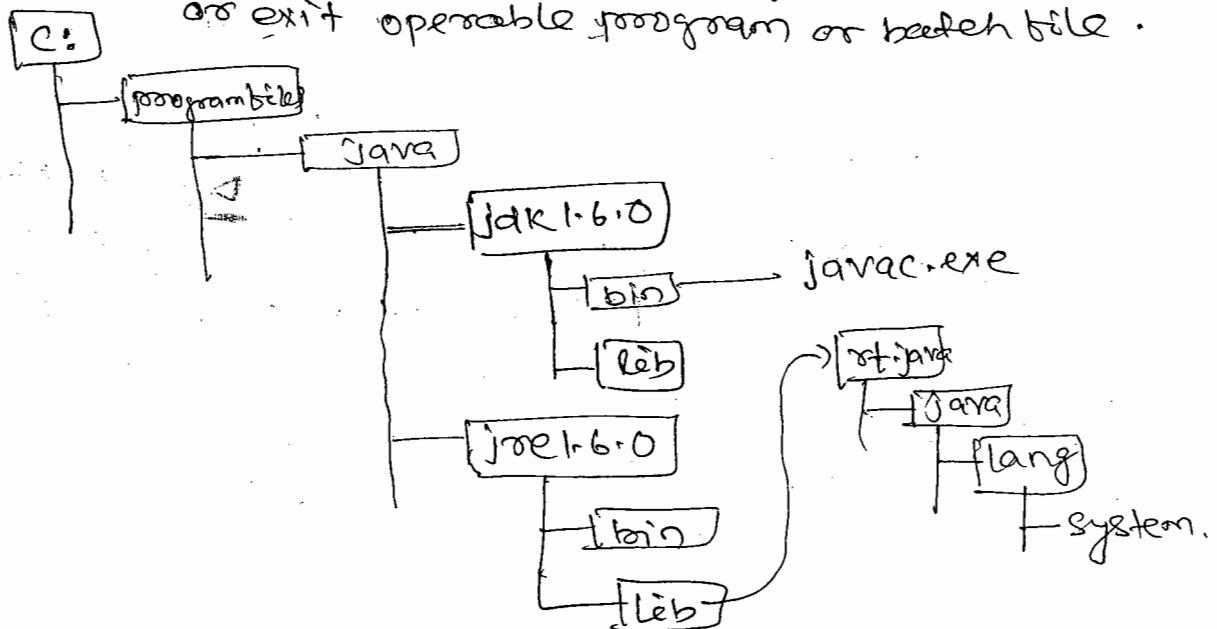
~~cd \~~

cd batch9am

~~batch9am > javac prog1.java~~

error.. javac is not recognized as an internal

or exit operable program or batch file.



So we can set the path.

Java

Java Environment

→ Java uses 3 environment variables

- i) path
- ii) classpath
- iii) JAVA_HOME

Path :-

- Path is used for locating executable files / binary files (i.e bin).
- In order to access binary files outside java software, we need to set path.

C:\batch 9am> Path = C:\program files\java\jre1.6.0\bin ; ↵
> javae program\java ↵

Classpath :- (location)

- Classpath is java environment variable used by java compiler & JVM for locating class files.

Set classpath = C:\program files\java\jre1.6.0\lib\ext.jar ; ↵ . // windows

export classpath = location ; // UNIX, LINUX

Date - 12/9/2012

JAVA_HOME :-

- This is a java environment variable used by java server's for locating java softwares.

Ex :- Tomcat, weblogic, glassfish

JAVA_HOME = C:\program files\java

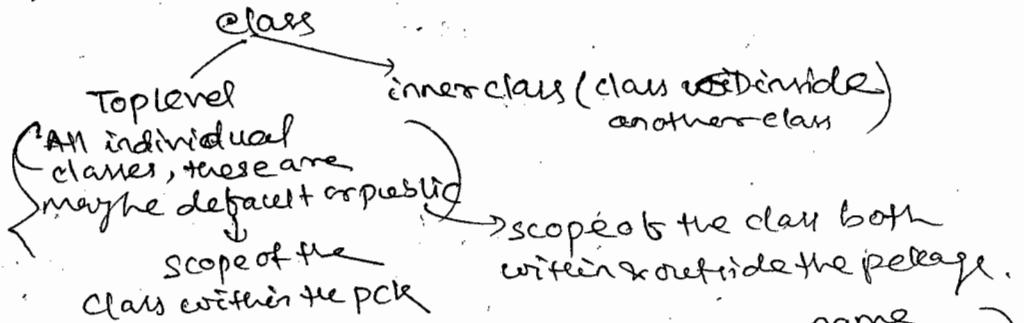
Program :-

```
program2.java
void main()
{
}
```

javac program2.java

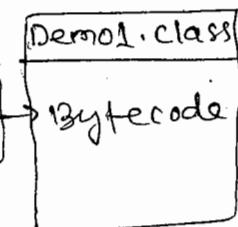
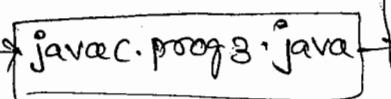
class, interface,
enum expected.

→ The above program display compile time error becoz java is a pure object oriented language. Everything in java must encapsulated within the class.



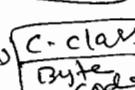
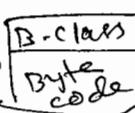
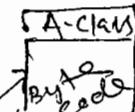
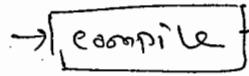
Program3.java

```
class Demo1
{
    void main()
    {
    }
}
```



Prog4.java

```
class A
{
    class B
    {
        class C
    }
}
```



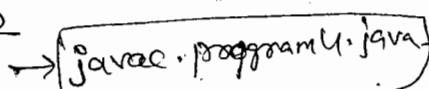
→ On compiling of java program, java compiler generates .class files.

→ For every class in java source program, compiler generates .class file, becoz class is a reusable program or executable program.

→ This class file contain bytecode

Program4.java

```
public class Demo2
{
    void main()
    {
    }
}
```



Error

Class Demo2 is
public, should be
saved with a
filename
Demo2.java,

✓ Every public class should be saved with class name.

→ Java supports 2 types of classes —

1. Top-level classes

2. Inner classes.

— Top-level classes : —

→ An individual class is called top-level class.

→ It allows only 2 access specifiers.

a) public

b) default.

ex: ~~private~~ class Abc ; } ~~protected~~ class Port {
 { }
 { }

— Inner classes : —

→ Class resides inside another class is called inner class or nested class.

ex: class A
 { class B]
 { }
 { }

→ When we type
C:\batch9am> java Demo1 ↵

→ Java invoke JVM which loaded Demo1 class from hardisk to RAM.

→ After loading it verify the bytecode is generating ~~properly~~ properly or not.

→ Then look for main method.

→ On execution of ~~the~~ Demo1 class JVM throws

an exception/error, No such method Error: main.

Syntax:-

① public static void main (String args[])
 {
 }

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

{

}

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

{

}

(d) public static void main (String... args) (Java 5.0)

{

}

① ② ③ ④ ⑤ ⑥
Public static void main (String args[])

{

}

1 → Access specifier

2 → modifier

3 → Return type

4 → method name (user-defined class)

5 → Class name (pre-defined class & available in java.lang package)
→ st. jar

6 → ~~array~~ name of the array (identifier)

Q) Why main method is public?

→ Public is an access specifier.

→ main() is public becoz it is accessible
the environment or package.

Static: —

→ It is an access modifier.

→ This method is called without creating object
of class.

→ Static methods binding with class name.

as method doesn't return any value.

main():-

→ It is the name of the method understand by JVM.

String args[]:-

- This method is called from OS.
- The values which are send from operating system environment are of type String.
- It is array becoz it receives 0 or more values.
- It is also called command line arguments.

Programs:-

class Demo3

```
{ public static void main(String args[])
{
    System.out.println("Inside main of String");
}
public static void main(int args[])
{
    System.out.println("Inside main of int");
}
public static void main(float args[])
{
    System.out.println("Inside main of float");
}
```

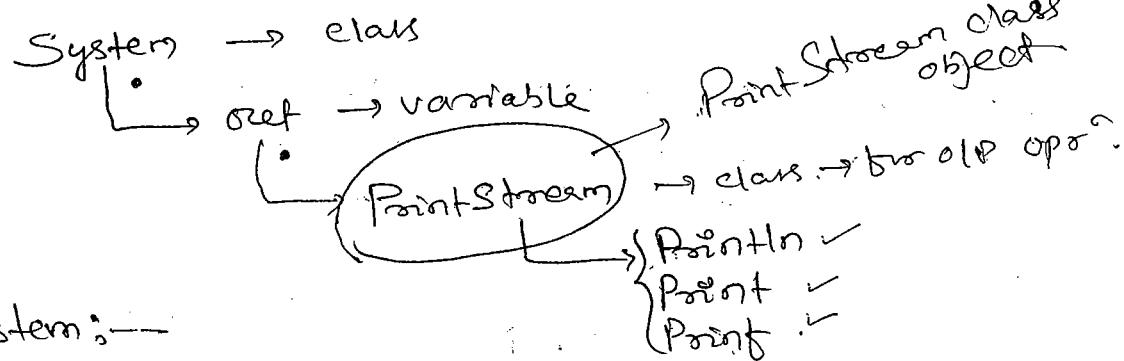
Output ↗
javac programs.java ↗

java Demo3 ↗

\$ Inside main of String

Printing methods :-

- 1) System.out.println
- 2) System.out.print
- 3) System.out.printf → Java 1.5/5.0

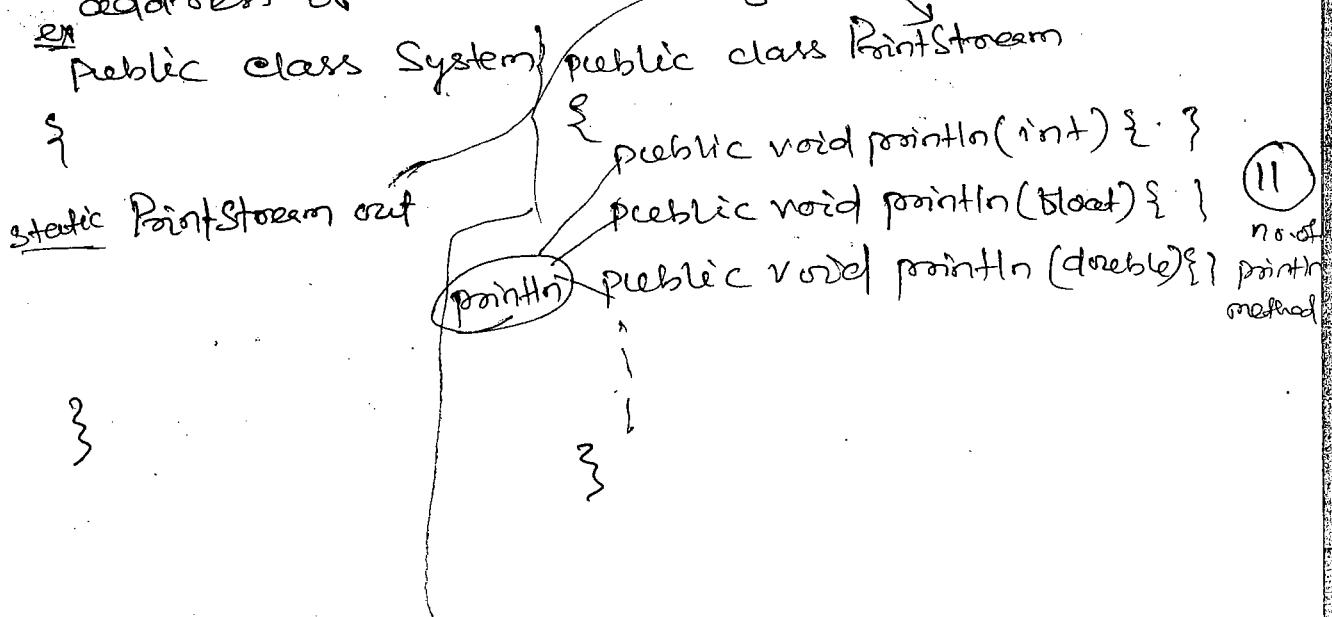


System:-

- System is a pre-defined class.
- This class available in `java.lang.package`.
- This default class provided by Java.

out :-

- It is a variable of **PointStream** class.
- It is a reference variable which holds an address of **PointStream** object.



- It is a static reference variable. So it is binding with class name i.e System.out

PointIn :-

- pointIn is a method of PointStream class.
- This method is overloaded ~~so~~ in order to print various types of values.
- Number of pointIn methods available are 11.
- PointIn, print data on console (monitor) / CUI
 - * CUI → Command user interface and insert new line.

Point :-

- It is a method of PointStream class.
- It print data without inserting a newline.
- No. of print methods are 10.

pointf :-

- It is a method of PointStream class.
- This method print various types of values.
- This method print one or more than one value.
- No. of pointf methods are 2.

Constants :-

- A constant is a value which never changes.

Ex:-

| | |
|------------------|---------------------------|
| int, short, byte | → 100, 200, 0, -100, -200 |
| long | → 100L, -100L |
| float | → 1.5f, 1.5F |
| double | → 1.0, -1.0 |
| char | → 'A', '0', '*' |
| String | → "java", "1.5", "100" |
| Boolean | → true, false |
| Reference | → null |

Program :-

Class Demo4

```
{ public static void main (String args[])
{
    System.out.println (100);
    System.out.println (1.5);
    System.out.println (2.5f);
    System.out.println (200L);
    System.out.println ('A');
    System.out.println ("Java");
    System.out.println (true);
    // System.out.println (null); error
}}
```

→ System.out.println (10, 20); // Invalid. It accepts only one argument

→ System.out.printf ("%d %d", 10, 20);

→ System.out.printf ("%s %f", "Java", 1.5f);

→ printing multiple values using printf

Output : path = C:\Program Files\Java\JDK1.6.0\bin ↴
javac prog4.java ↴

System.out.println (null);

→ The above statement generates compile time error

becoz, null constant cannot print directly.

→ In order to print null, it is stored inside some variable and printed.

Local variables:

- i) → A variable declared inside method is called local variable.
- ii) → It is having local scope.
- iii) → A local variable must assign value before accessing it.

e.g.,

C

```
void main()
{
    int x, y, z;
    y = 20;
    z = x + y;
    Point p("x,y,z");
}
```

no error

- Garbage collection is performed.
 - Not a strongly typed language.
 - Array may be static or dynamic
- examples &

Class Demo5

```
{public static void main (String args[])
{
    int x, y;
}}
```

Output:- java Demo5

Demo5 ↴

- The above program does not display any compile time error becoz x, y values are not access by program.

Java

Class Demo

```
P.SVM (String args[])
{
    int x, y, z;
    y = 20;
    z = x + y;
    S.O.P(z);
}
```

error.
→ No Garbage collection is performed.

- It is a strongly typed lang, i.e. Robust.
- Array must be dynamic

(v)

class Demof

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

```
{ int x, y ;
```

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

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

```
}
```

Output : javac progt.java ↴

error : → The above program display complete error
becz x & y variables not assign any variable.

→ To overcome this error we assign the
values to x & y .

☞ Local variables does not allows any modification
except final .

example :

class Demof

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

```
{ int x = 100 ;
```

```
long y = 200L ;
```

```
float f = 1.5f ;
```

```
double d = 2.5 ;
```

```
boolean b = true ;
```

```
String s = null ;
```

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

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

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

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

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

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

```
}
```

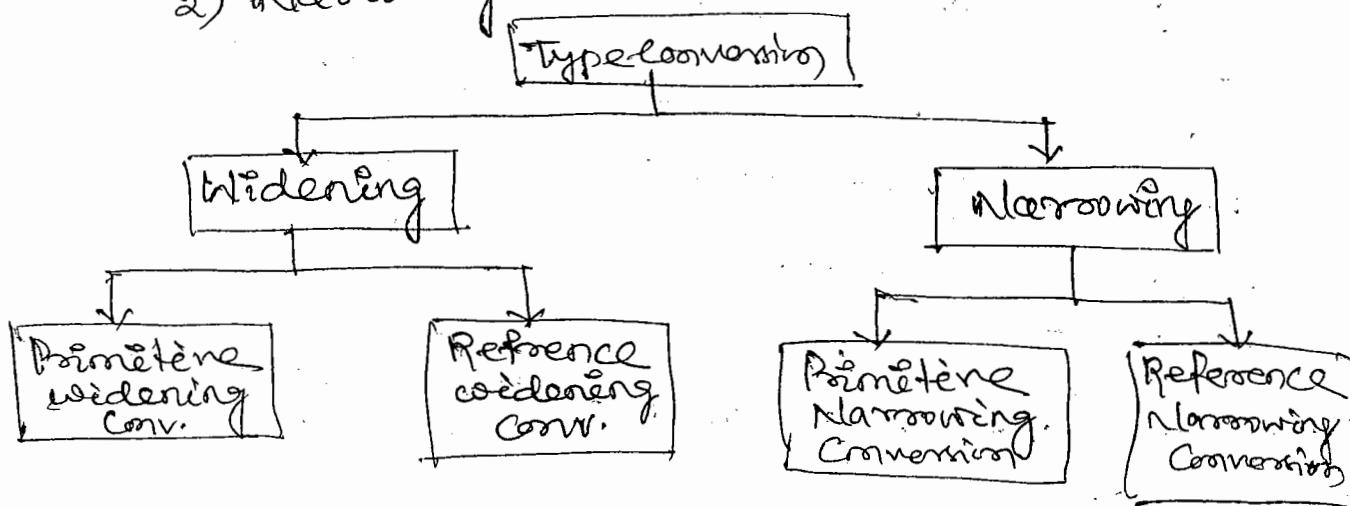
* boolean b = 1 // error

Date-14/9/19

Type Conversions:

- Converting one type of value to another type is called "type conversion".
→ These type conversions are 2 types

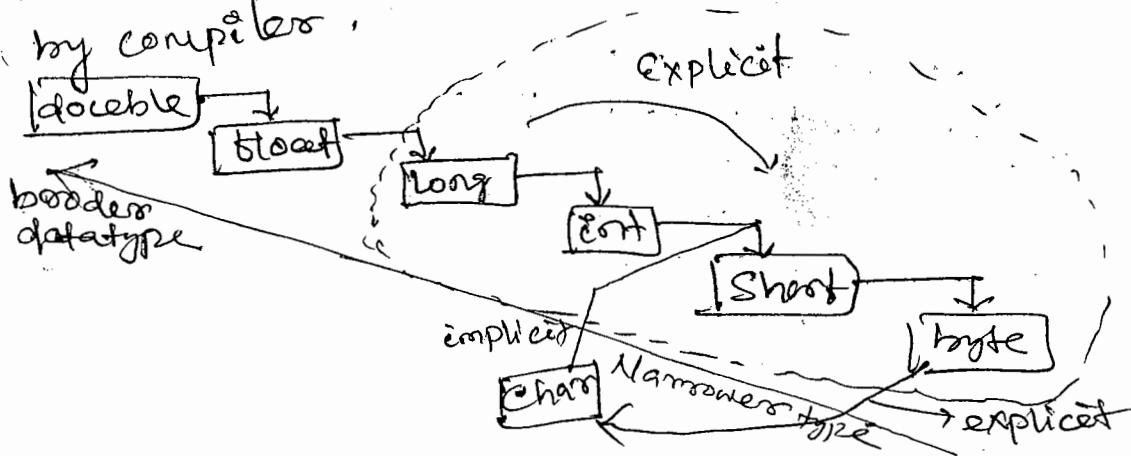
- 1) Widening conversion
- 2) Narrowing conversion.



} int - narrower datatype
float - broader datatype → more precision,

1) Widening Conversion:

- Converting of narrower datatype to broader datatype is called "widening conversion".
→ Converting narrower primitive datatype to broader primitive datatype is called widening primitive conversion.
→ This conversion is implicit and it is done by compiler.



Byte :-

- byte is 1 byte integer datatype.
- It accept value which range from -128 to +127.

example:-

Class CDemo1

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

```
    byte b1 = 100;
    byte b2 = 200;
```

}

Output:-

- The above program display compile-time error.
becoz '200' is not ~~value~~ within byte range.
- 200 is of integer value.

example:-

Class CDemo2

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

```
    byte b1 = 100;
```

```
    byte b2 = 20;
```

```
    byte b3 = b1/b2;
```

```
    System.out.println(b3);
```

}

Output:-

- The above program display compile-time error
becoz any arithmetic operation on byte
overlaid of type integer.
- The conversion between integer and byte
is not implicit.

Note:-

byte b3 = 10+20; // valid, no error.

int value X
integer constant

30 → it is the constant & inside the
byte range

→ It will produce one
byte b3 = 10 + 20 ;

→ The above program does not display any
compiletime error. Any arithmetic operation
integer constants, result ~~can~~ can be byte,
short, int or long.

Program 3 :-

```
class CDemo3
{
    p. s. v. m (String args[])
{
    byte b3 = 100 * 20 ;
    S. O. P. (b3);
}
```

Output :-

→ Loss of precision, ~~can~~ exceed the range

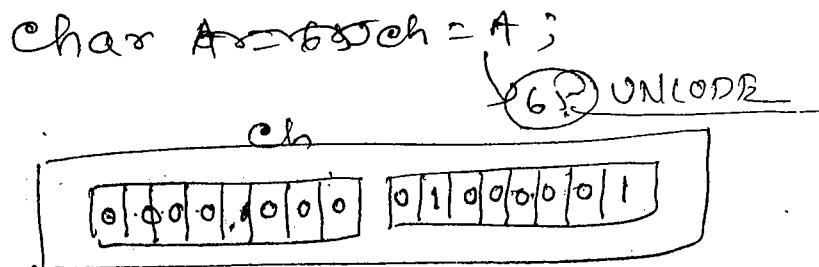
Character :-

→ Character is character datatype and bytes
integer datatype.

→ The range of char is "0 to 65535" (UNICODE)

Program :-

```
class Demo4
{
    public static void main (String args[])
{
    byte b = 65 ;
    char c = b ;
    S. O. P. (b) ;
    S. O. P. (c) ;
}
```



→ error:- Here the
The above program display compile time error.
be coz conversion between bytes & characters is
not implicit.

→ Conversion between integer constant and character
is implicit.

ex:- char ch1 = 65;
char ch2 = 'A';
char ch3 = 92;

SOP(ch1); → A
SOP(ch2); → A
SOP(ch3); → a

→ So the conversion is explicit.

int:-

→ Integer datatype whose size is 4 bytes.

→ Range of integer is -2147483648 to 2147483647.

Program:- (for range)

Class D
{ public static void main (String args[])
{ S.O.P (Integer.MAX_VALUE);

Short:-

→ It is also an integer datatype, whose size is
2 byte.

→ Its range is -32768 to 32767.

Program:-

Class CDemo5

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

```
{ byte b1 = 65;
```

```
short s1 = 100;
```

```
int i1 = b1; } → implicit conversion.
```

```
int i2 = s1;
```

```
S.O.P (b1);
```

```
S.O.P (b1);
```

```
S.O.P (i1);
```

```
S.O.P (i2);
```

```
}
```

Output:

65

100

65

100

Program:

Class CDemo6

```
{ public static void
```

```
{ char ch = 'A';
```

```
int i = ch;
```

```
S.O.P (ch);
```

```
S.O.P (i);
```

```
}
```

Output:

A
65

long:-

- It is a integer datatype whose size is 8 bytes.
- The range of long is

-9223372036854775808 to
+9223372036854775807

Program:-

Class CDemo 6

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

```

    int x = 100;
    byte b = 65;
    char c = 'B';
    long l1 = x;
    long l2 = b;
    long l3 = c;
    S.O.P(l1);
    S.O.P(l2);
    S.O.P(l3);
  }
```

Output

Op 100, 65, 66.

65
66

Program:-

Class CDemo 7

{ PSVM (String[])

```

    long l1 = 100;
    long l2 = 65536;
    long l3 = 2147483647;
    long l4 = 2147483649L; //to exceed the
                           integer range
    S.O.P(l1);
    S.O.P(l2);
    S.O.P(l3);
    S.O.P(l4);
  }
```

float :-

- It is a real datatype having precision.
- Size of this datatype is 4 bytes.
- Range of this datatype is

$$3.4028235 \times 10^{-38} \text{ to } 3.4028235 \times 10^{38}$$

$$= 1.4 \times 10^{-45} \text{ to } 1.4 \times 10^{38}$$

$$\text{i.e., } 10^{-45} \text{ to } 10^{38}$$
- float value must be prefix with f or F.

example :-

```
class Demo8
```

```
{ public void main(String args[])
{
    int x = 100;
    byte b = 65;
    char ch = 'A';
    float f1 = 1.5f;
    float f2 = 2;
    float f3 = b;
    float f4 = ch;
    System.out.printf("x.%f.f1,f2,f3,f4", x, f1, f2, f3, f4);
}}
```

```
class Demo9
{
    public void main(String args[])
    {
        float t1 = 10; ✓
        float t2 = 10.0; X
        float t3 = 10.05; ✓
        ;
    }
}
```

}

double :-

- Double is real datatype it contain precision.
- The size of datatype is 8 bytes.
- Range of these is

$$1.7976931348623157 \times 10^{-308}$$

$$\text{to}$$

$$1.9 \times 10^{-324}$$

Class CDemo9

{ Public static void main (String args[])

```
{ int x = 100;  
    char y = 'A';  
    byte b = 65;  
    float f = 1.5f;  
    long l = 100;  
    double d1 = 1.25;  
    double d2 = x;  
    double d3 = y.;  
    double d4 = f;  
    S.O.P (d1);  
    S.O.P (d2);  
    S.O.P (d3);  
    S.O.P (d4);  
}
```

OP
1.25
100-0
65
1.5f.

example :

double d1 = 100; → 1.
double d2 = 100f; → 1
double d3 = 100L; → 1

Date - 15/9/12

Narrowing Primitive Conversion:

→ Converting broader primitive datatype to narrower primitive datatype is called narrowing primitive conversion.

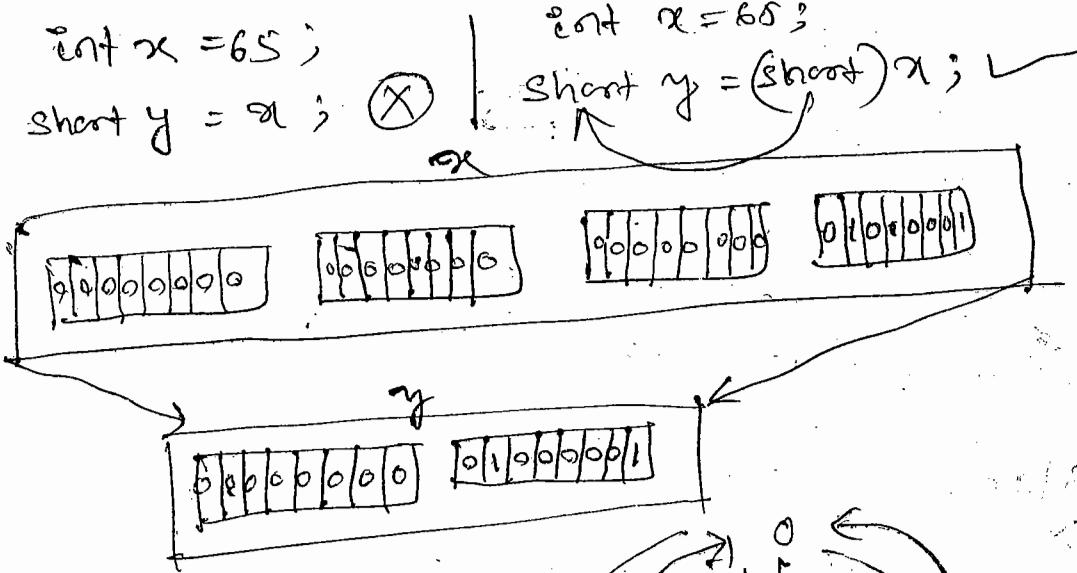
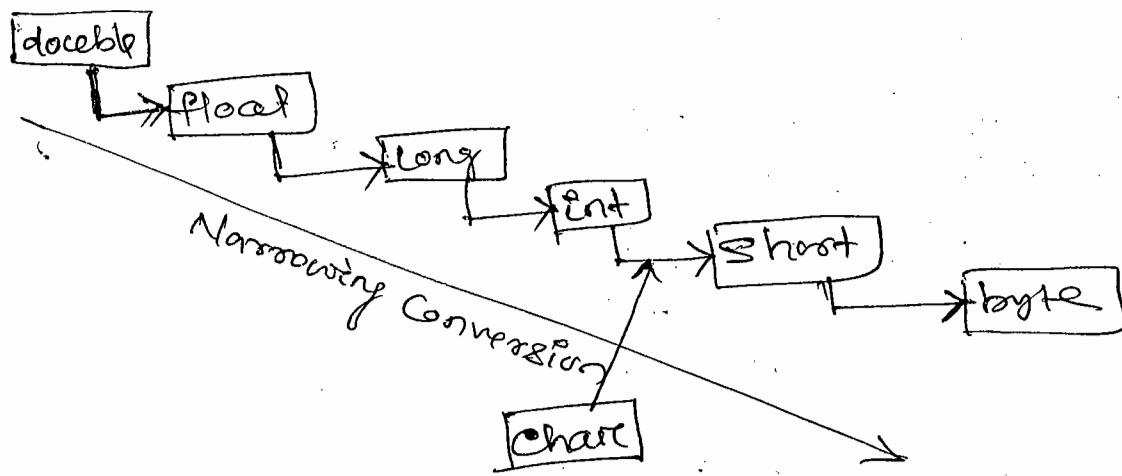
→ This conversion is explicit. It has to done explicitly by programmer.

→ There is a chance of loss of value.

→ This conversion is done by using type casting operator.

→ Syntax :-

(type) → destination type -



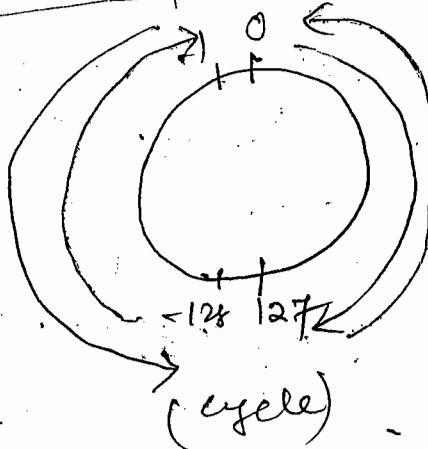
Short x = 200;

byte y = (byte)x;

S.O.P(x); → 200;

S.O.P(y); → -56;

such kind of conv. are
not done in java.



Class CDemo10

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

```
{  
    short x = 200;  
    byte b = (byte) x; ...  
    System.out.println(x);  
    System.out.println(b);  
}
```

Output
200
-58

Program :

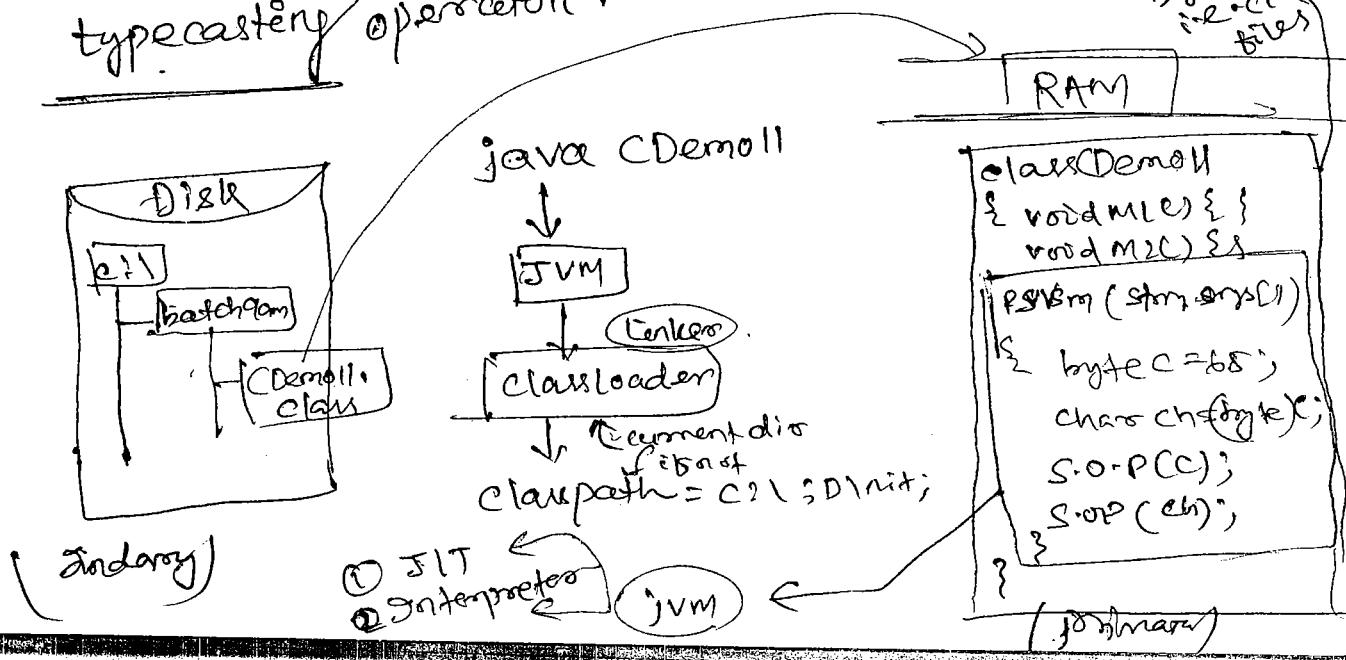
Class CDemo11

```
{ public static void main (String args[]) }  
{  
    byte c = 65;  
    char ch = (char)c;  
    byte b = (byte) ch;  
    S.O.P (c);  
    S.O.P (ch);  
    S.O.P (b);  
}
```

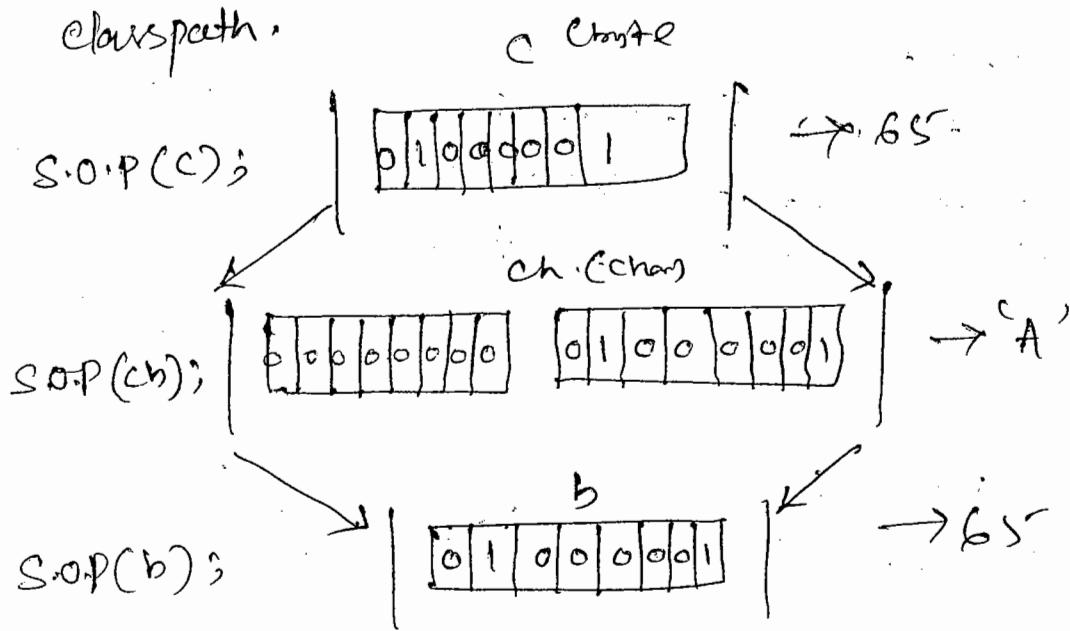
Output
65
A
65

→ ~~get~~ conversion between byte to character and character to byte is not implicit. This conversion has to be done explicitly using typecasting operator.

~~Bytecode~~
~~ie. class files~~



→ whenever an error comes class is not found
 → if class is not defined then set the
 classpath.



example:

class CDemo12

{
 <pre>svm (String args[])</pre>

{ double d = 1.5;
 float f = (float)d;
 int i = (int)d; // loss of precision.

S.O.P(d);

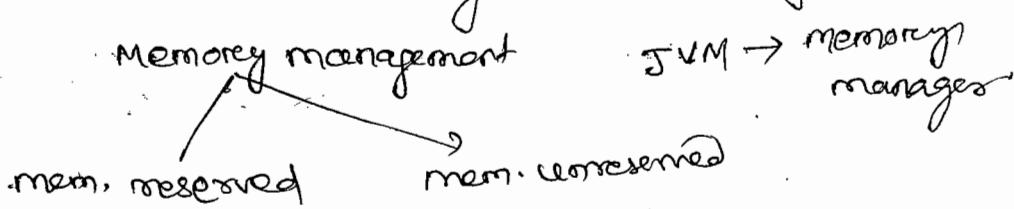
S.O.P(f);

S.O.P(i);

} }

Arrays:

- Array is a collection of similar type of data elements.
- An array is a reference datatype, ~~because~~ it holds the address.
- Array is in java are dynamic.
- JVM allocates/reclaims memory for every java program, which is divided into 2 parts.
 1. Static memory Area (SMA)
 2. Dynamic memory Area (DMA)
- Static memory area is unmanaged memory area which cannot managed by java programme. This memory is managed by JVM.
ex: Local variable.
- Dynamic memory area is managed memory area which is managed by java programme.
- DMA avoids the wastage of memory.



- Array is for avoiding more no. of variables & required for grouping.

new :-

- This operator allocates memory within dynamic memory area (heap area)
- new reserve memory and return address of reserved memory.

Date - 16/9/12

Types of Array :-

- 1) Single Dim. Array
- 2) Multi Dimension Array
- 3) Jagged Array

Single dimension Array :-

- An array which is refers with one subscript is called single dimension array.

Syntax :-

1. `datatype array-name[];`

Ex:- `int a[5]` → Java → As in java array is dynamic.

int a[]; → memory allocated for reference variable.

2. `datatype[] array-name;`

- For declaration for array the memory is allocated for reference variable.

Syntax for creating array :-

`new type-name [size];`

- One creation of array memory is allocated for elements / data.

→ Array is an implicit object.

Size Rules:

- 1) Size can be a constant or variable
- 2) Size must be ≥ 0

Example:

Class ArrayDemo1

{ Public static void main (String args[])

{ int a[];

a = new int [5];

System.out.println (a);

System.out.println (a.length);

}

O/P

[I@3e25a5]

5

→ [I @3e25a5]
Single array Integer hashCode → address

→ [] → 2D array [] → 3D array

Example - 2

Class Array Demo2

{ public void main (String args[])

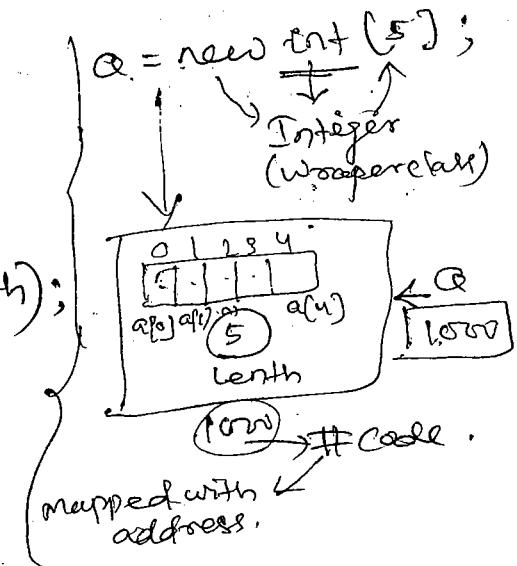
{ int a[];

a = new int [-2];

S.O.P (a);

}

S.O.P (a.length);



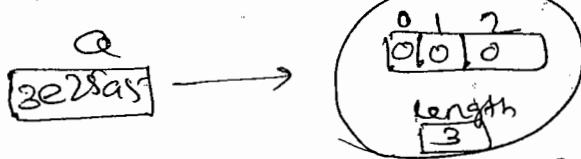
* The above program displays an error during execution, bcoz an array cannot be created with -ve size.

How to Initialize an array?

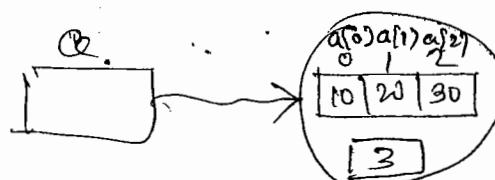
Syntax :-

[datatype array-name] = {list of values};

example :- declaration creation
1. int a[] = new int[3]; ← declaration & creation.



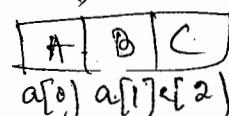
2. int a[] = {10, 20, 30} ← Initialization.



3. float b[] = {1.5f, 2.5f, 3.5f};

4. char c[] = {'A', 'B', 'C'};

char c[] = {"ABC"}; → (A group of char. concatenated)



Note :-

5. float a[] = {10, 20, 1.5f}; ✓ (implicit conversion).

int b[10] = {10, 'A'}; ✓ (widening conversion)

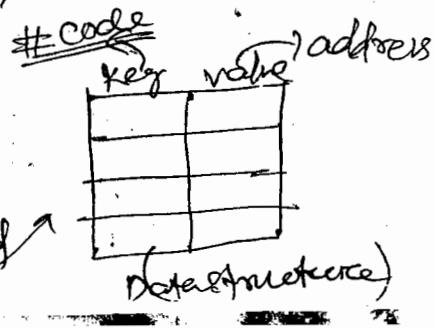
double d[] = {10, 1.5, 2.5f, 'A'}; ✓ / valid.

SOP(a); → [F@ —]

SOP(b); → [I@ —]

SOP(d); → [D@ —]

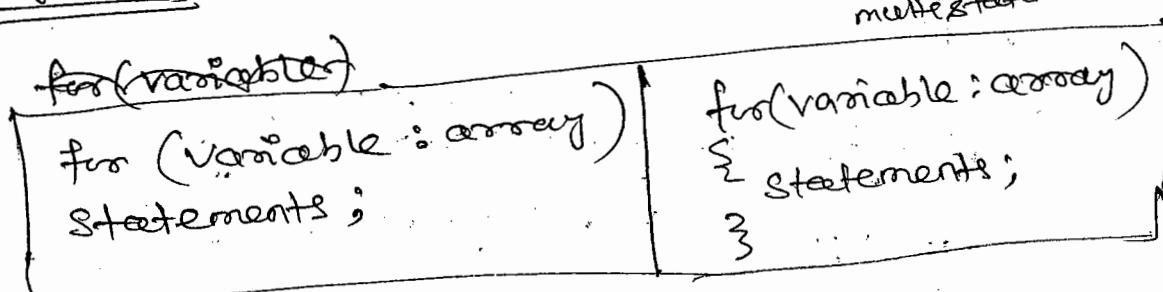
* Array is built in a datastr. called hashcode. It contains 2 things



→ Reading and writing elements within array is done using index.

→ Java 5.0 supports enhanced for loop, which is used for reading elements from array without using index.

Syntax :- (for enhanced for loop.)



example

example
int a[] = {10, 20, 30, 40, 50};

$\text{SOP}(\alpha[0]) \rightarrow 10$

$SOP(\alpha[1]); \rightarrow w;$

1. `for (int i=0; i<a.length; i++)` → This can be used for any purpose.

~~800 R (off)~~

~~8.0P~~($\alpha[i]$); $\rightarrow 10, 20, 30, 40, 50$.

2. `for (int n:<)` → enhanced for loop
• This can only use

$\text{SOP}(n)$:

Programme:

Class Array Demo 2

{ public static void main (

$$\{ \inf a = \{ 10, 20, 30, 40, 50 \} ;$$

```
for (int n : a);
```

∴ $i_5(87\% 2 == 0)$

S.O.P ("Even no.");

else

} } } else S.O.R.P ("Odd no.");

~~Reading the arrays & collecting~~

↓
collection
of
different
types
of
leaf

Miltè dim. Array :-

Date - 17/9/12

2D array :-

~~~~~

- An array of arrays is called 2D array or multi dim array.
- This array is addressed using two ~~one~~ scripts.
- It is a matrix.

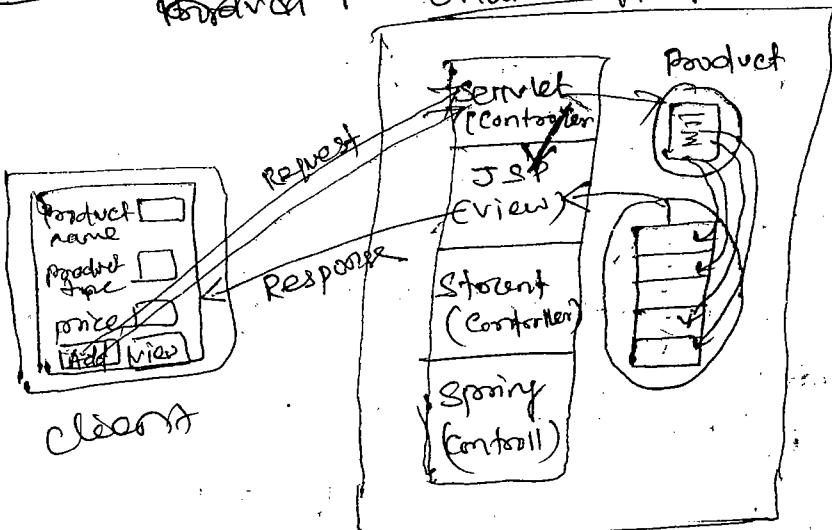
Syntax :-

datatype array-name [ ] [ ] ;

datatype [ ] [ ] array name ;

ex:-

Product → Onlineshopping



- Array is may be primitive type or Collection of diff. types.

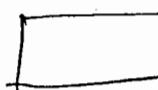
Syntax for Creating an Array :-

{ new datatype [resize] [csize] ;  
[ new datatype [resize] [ ] ;

Ex :-

int a[][];

a



4 bytes

Q.) What is hash code?

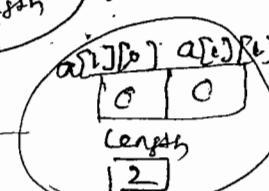
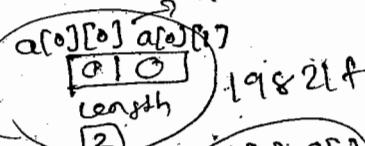
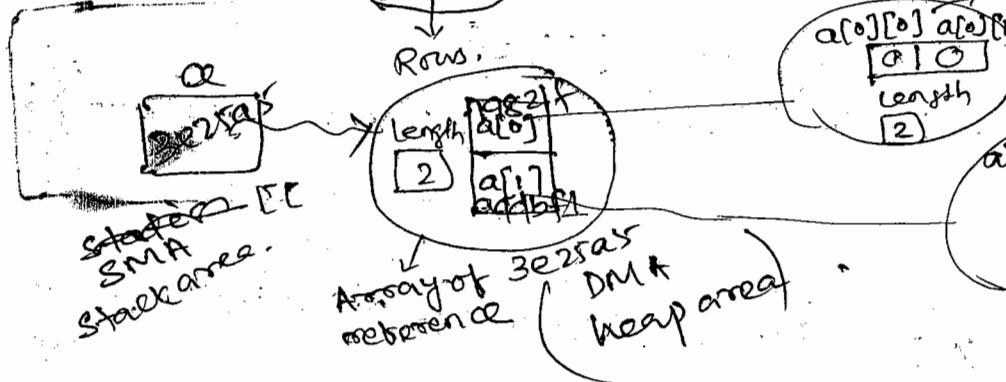
- The hash code of java object is simply a number (32bit sign int) that allows an object to be managed by a hash-based data structure.
- It contains 2 things: 1) Keys  
2) Values.

int a[3][3] → X → Invalid.

int a[3][] → Invalid

int[][] a → Valid.

a = new int[2][2]; ✓      Array of arrays is called 2D array of values.



add[1]

Program:-

class Array Demo3

{ public static void main (String args[])

{ int a[][];

a = new int[2][2];

System.out.println(a);

System.out.println(a.length); // no. of rows.

System.out.println(a[0].length); // no. of columns.

System.out.println(a[1].length);

System.out.println(a[0]);

System.out.println(a[1]);

}

Object :-

[ [ @3e25a5 ] ]

2

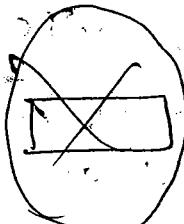
2

2

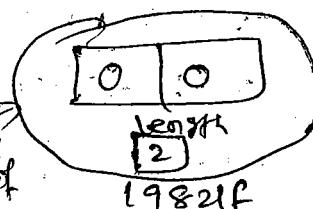
[ I @19821f ]

[ I @ addbf1 ]

3e25a5



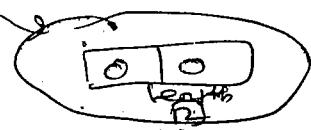
Arrayed values



Arrayed  
Reference



3e25a8



addbf1

Syntax for Initialization :-

datatype array\_name[ ][ ] = {

{ { row1 values }, { row2 values }, { row3 values }, ... } ;

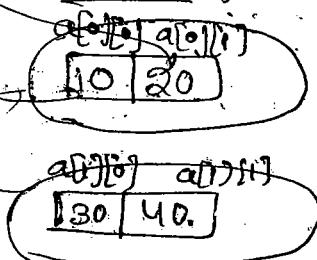
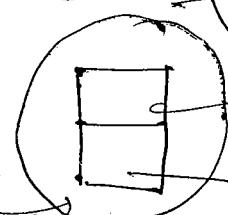
Ex :- int arr[ ][ ] = { { 10, 20 }, { 30, 40 } }  $\rightarrow$  X  $\rightarrow$  in C .

C in C col.size is mandatory .

int arr[ ][ 2 ] = { { 10, 20 }, { 30, 40 } }  $\rightarrow$  ✓  $\rightarrow$  in C .

int arr[ ][ 2 ] = { { 10, 20 }, { 30, 40 } }  $\rightarrow$  ✓  $\rightarrow$  in Java .

a



Program:-

class Array Demo4

{ public static void main (String args[])

{

    int a[ ][2] = {{10,20},{30,40}};

}

op:

Error : displaying compile time error.

\*→ The above program displaying compile time error.  
becoz array cannot be initialize with size.

Reading Elements from 2D Array :-

class Array Demo5

{ public static void main (String args[])

{ int a[][] = {{10,20},{30,40},{50,60}};

    for(int b[]: a) // row

{

        for (int n: b) // column

{

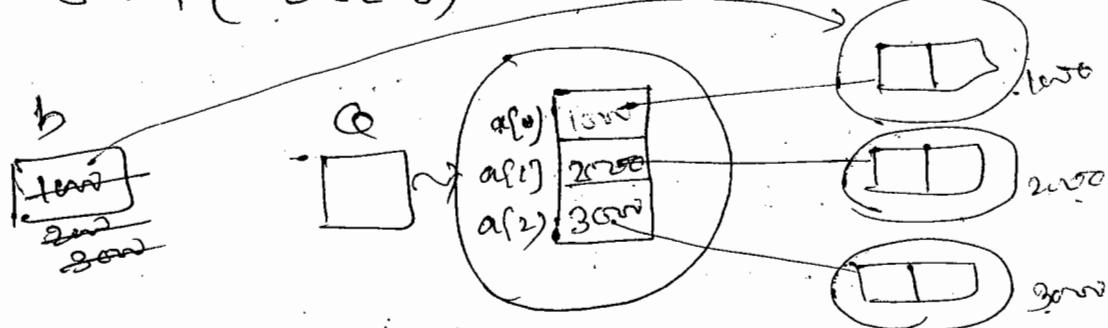
            S.O.P.(n);

~~for~~

} } }

for

{  
    for (int i=0; i<a.length; i++)  
        for (int j=0; j<a[i].length; j++)  
            S.O.P(a[i][j]);



## Jagged Array :-

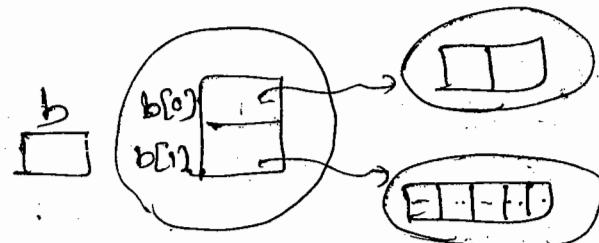
→ Jagged Array is a multi-dimensional array having fixed no. of rows & variable length columns.

e.g. int b[2][ ];

{ b = new int[2][ ];

    b[0] = new int[2];

    b[1] = new int[5];

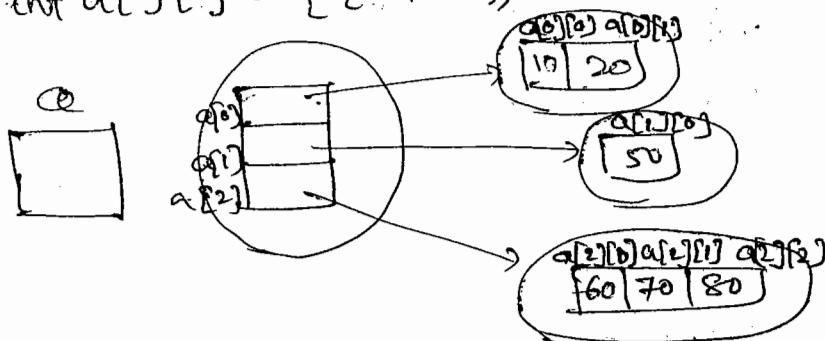


→ The above example shows the creation of Jagged array.

→ For Jagged array enhanced for loop is used.

## Initialization of Jagged Array :-

int a[][] = {{10, 20}, {50}, {60, 70, 80}};



## Program:-

### Class ArrayDemo6

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

```
    int a[][] = {{10, 20}, {50}, {60, 70, 80}};
```

```
    for (int b[] : a)
```

```
        for (int n : b)
```

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

```
}
```

Date - 18/9/12

## Command Line Arguments :-

- The values which are send from command prompt to main() are called commandline arguments.
- These arguments are of type String.

example :-

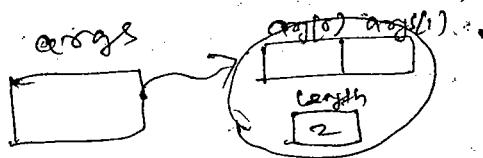
```
class CommDemo1
{
    public static void main (String args[])
    {
        System.out.println (args.length);
        for (String s : args)
            System.out.println (s);
    }
}
```

Output  
2  
10  
20

javac CommDemo1.java

c:\> batch\arg>-

java CommDemo1 10 20 ← command arguments.



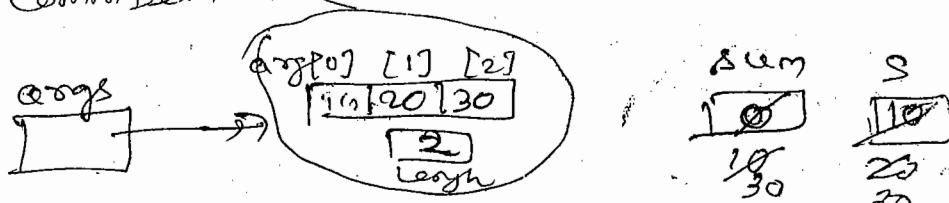
Program :-  
Sum of Numbers send from command prompt.

```
class CommDemo2
{
    public static void main (String args[])
    {
        int sum = 0;
        for (String s : args)
        {
            sum = sum + Integer.parseInt (s);
        }
        System.out.println (sum);
    }
}
```

Q18

`javac CommDemo2.java`

`java CommDemo2`



$$0 + 10 = 10$$

$$10 + 20 = 30$$

$$30 + 30 = \underline{\underline{60}}$$

Integer.parseInt():

→ Integer is a predefined class available in `java.lang` package.

→ `parseInt` is the static method of Integer class.

→ This method converts string representation of integer to int.

`psvm(String... args)`

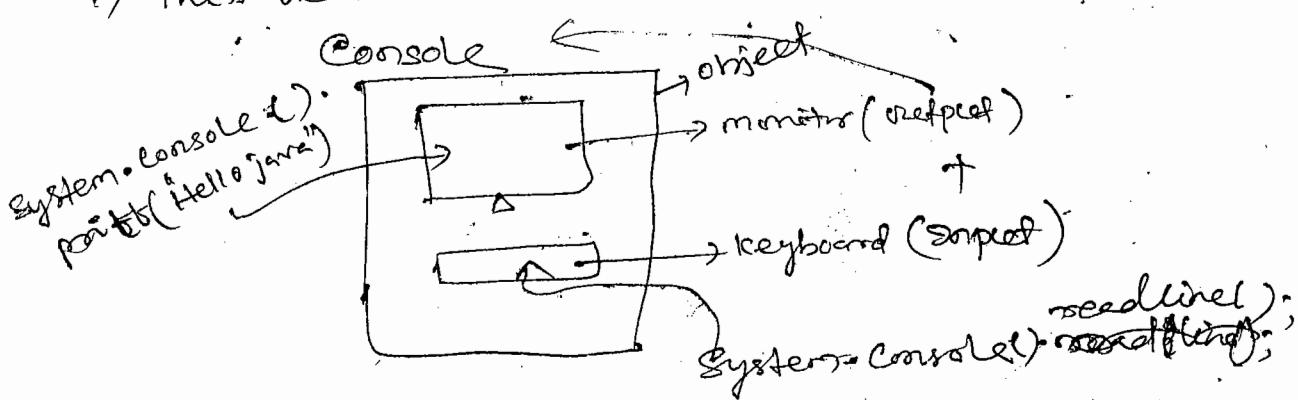
↳ Variable length arguments.  
added in Java 5.0.

System.console():

→ `console()` is a static method of System class.

→ This method returns reference of console object.

→ This feature is added in Java 6.0.



### Program:-

```

class ConsoleDemo1
{
    public static void main (String args[])
    {
        System.out.println ("Hello Java");
    }
}

```

Output: Hello Java

### Program:- (instead of above)

```
class ConsoleDemo2
```

```
{ public (String args[])
```

```
{
    Console out = System.out;
    out.println ("Hello Java");
}
```

//WAP to read 2 no.s from console & print sum

```
class ConsoleDemo2
```

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

```
    System.out.println ("Input any two
numbers");
```

```
String s1 = System.out.readLine();
```

```
String s2 = System.out.readLine();
```

```
int n1 = Integer.parseInt(s1);
```

```
int n2 = Integer.parseInt(s2);
```

```
int n3;
```

```
int n3 = n1 + n2;
```

```
System.out.println ("The sum is " + n3);
```

## readLine():

→ It is a method of Console class which  
read string;

→ If we want read the data from keyboard →

a) In C → Scanf()

b) In Java → System.console().readLine()

+ Operator :- Adding nos.

+ → Concatenating strings.

→ In Java '+' operator performs 2 operations.

i) Adding numbers.

ii) concatenating strings.

→ It add two numbers if both operands are  
of type number.

→ It performs concatenation, if any one operand  
are of type String.

→

### Example:-

S.O.P(10+20); → 30

S.O.P("10"+20); → "1020"

S.O.P(10+"20"); → "1020"

S.O.P("10."+"20"); → "1020"

\* S.O.P("Sum is "+10+20) → Sum is 1020

\* S.O.P("10+20" + "Sum is") → 30 is Sum

① System.out.println()

② System.out.print()

③ System.out.printt()

④ System.console().printt()

⑤ System.console().readline()

# Object oriented Programming (OOP) :-

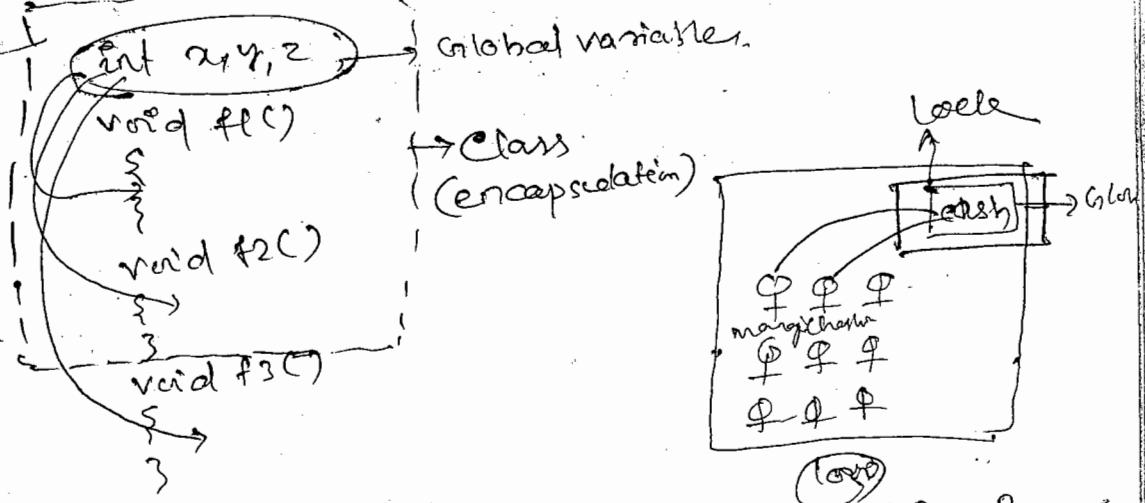
20/9/12

→ Object oriented Programming is a programming method, which define set of rules and regulation for organization of data & instruction.

elements are  
→ Programming consists of 2 types - ① Data.

② Instruction.

→ Organization of Data & Instruction is called programming.  
as it's giving problem



→ Drawback of Structure/Procedural oriented Prog.?

→ There is no proper organization of data & instruction.

→ Data is global, which can be access by unrelated function and unrelated function.

→ Data is not secured.

→ Debugging application is complex. (i.e. In a large program identifying which data is operated which operation(function) is complex.

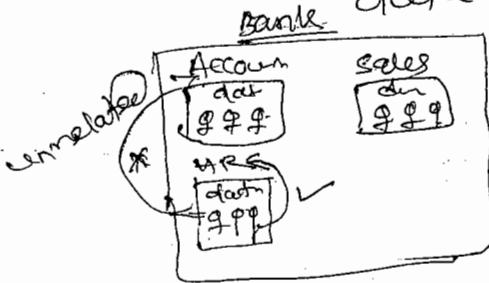
## Encapsulation:

→ It is a process of grouping data & instructions which operates on <sup>that</sup> data as a single entity.

→ Merging of data & instructions is called encapsulation.

## Adv. of encapsulation :-

1) Data hiding :- Preventing data access from unrelated operation is called data hiding.



2) Binding :- Linking operations with data is called binding.

## Class :-

~~= X =~~ Class is a building block of an object oriented programming.

→ Class is a building block of an object oriented programming. (In C → function).

→ Class is a collection of fields and methods. Fields are nothing but variables.

→ Class is a blueprint of object.

→ Class defines the structure of object.

→ Class is planning before creating object.

→ Class is datatype.

→ Class is metadata i.e. data of data.

→ Class is a reusable code component of module.

Syntax :- (optional)

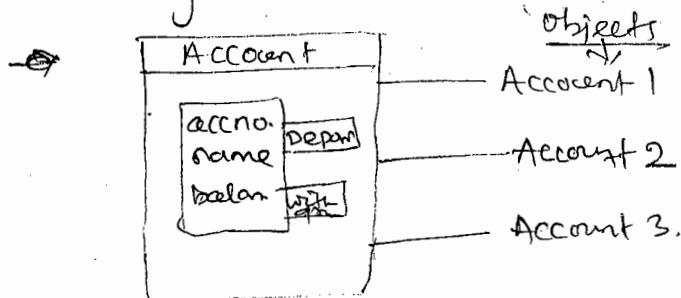
[modelname] class class-type-name

{  
    Variables ; [fields]  
    functions ; [methods]  
}

Dehra

# Object :-

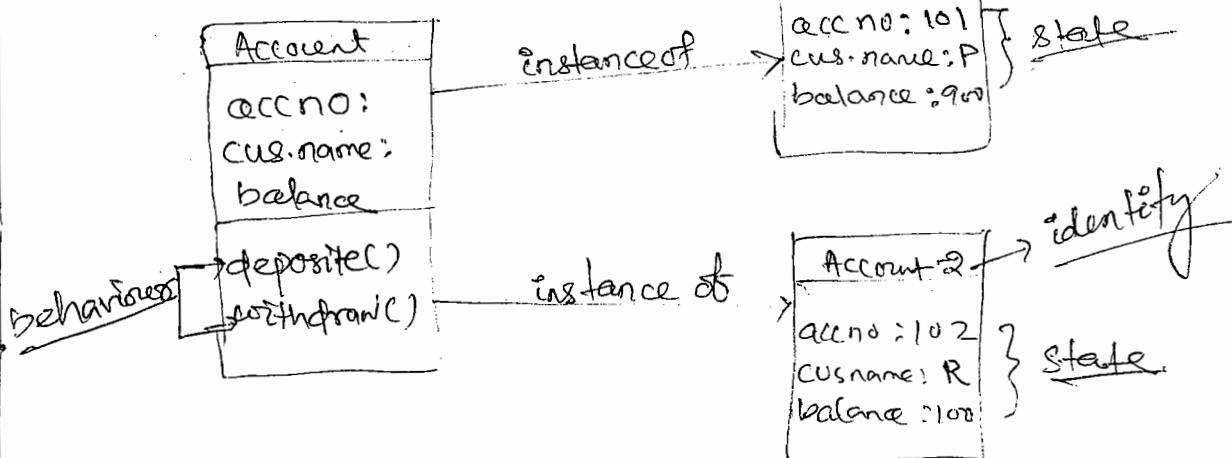
- Object is an instance of a class.
- In object oriented programming data is represented as objects.
- An instance is nothing but allocating memory for variables exist within class.
- Using one class programmer can create any number of objects or instances.



- Every object is having 3 properties —
  - 1) state
  - 2) behaviours
  - 3) Identity.

## State:-

- The value given to an attribute of an object define state.
- State of an object can be changed using behaviours.



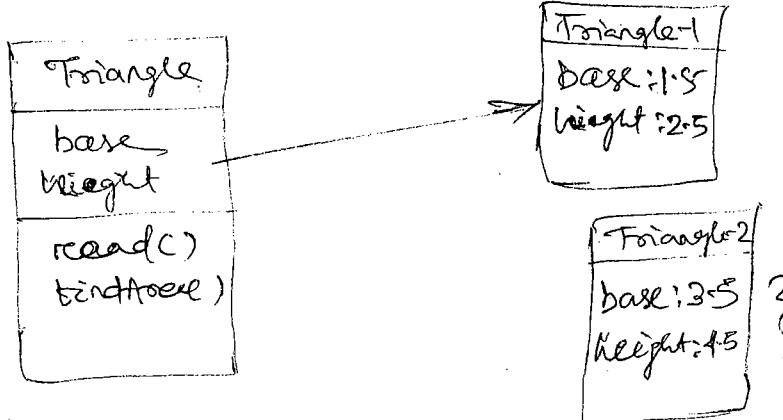
- State is nothing but data hold by object.

### Behaviour :-

It is an operation performed by an object is called behaviour.

### Identity :-

Each object is identified by the unique name.



### Variables :-

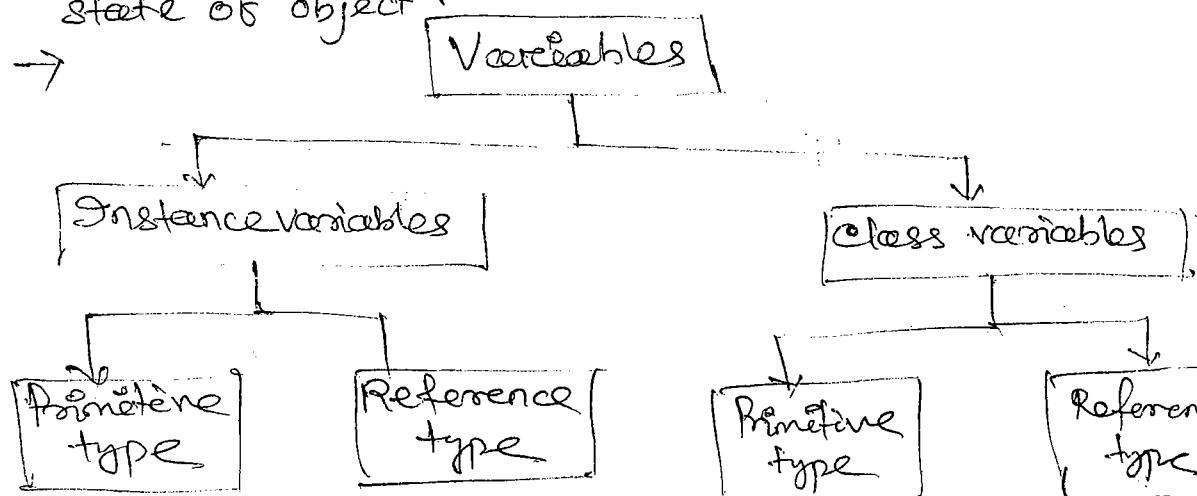
The variables declared inside class is of 2 types —

1) Instance variables .

2) Class variables .

The variables declared within class define state of object .

→



## Instance Variables:

- A non-static variable of a class is called instance variable.
- Instance variable memory is allocated on creation of object.
- Every object has its own state, which is defined by using instance variable.
- These variables bind with object name.

Ex:- Class Book

```
{ String bname;
  String author;
  float price;
}
```

- instance variable  
becoz no static keyword is used.
- These are called when object is created.

## How to create object?

- In java objects are dynamic.
- These are created during execution of program.
- All objects are created within managed memory area called heap area.
- Java allows to create object in 3 ways
  - 1) Using new operator.
  - 2) class.forName("classname").newInstance()
  - 3) factory method.

## New Operator:

- new is a operator which perform 3 operations.
  - 1) Loading. (harddisk to RAM)
  - 2) Instantiating. (allocating memory for non-static variables)
  - 3) Initialization.

## Loading :-

→ Copying ".class" files to class from binary to primary.

## Instantiation :-

Allocating memory to non static variables of a class (Creating object).

## Initialization :-

```
Class A
{
    int x; → default value
    can be assign
    P.S.V.M ( S )
    {
        int y; → x
    }
}
```

→ Assigning default values to object is called initialization.

## Default value :-

|                          | Default<br>value                 |
|--------------------------|----------------------------------|
| int, short<br>long, byte | 0                                |
| float, double            | 0.0                              |
| char                     | \00000 → non printeable<br>value |
| boolean                  | false                            |
| Reference                | null                             |

## Ex:-

### class Book

```
{
    String bname; → null
    String auther; → null
    float price; → 0.0
}
```

] default  
value

## Syntax :-

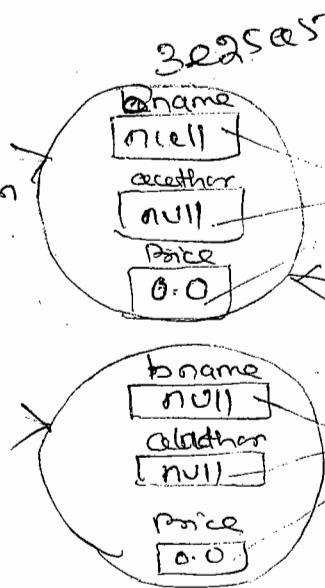
|                        |
|------------------------|
| new class-type-name(); |
|------------------------|

Ex:- new Book();  
new Book();

RAM

① Loading  
Book Class  
Class Books  
{ String bname;  
String author;  
float price;  
}

② instantiation  
instant.



③ Initialization

book1  
book2

④ instt  
book2

→ Class file only once loaded into the RAM.

## Reference Variable:-

→ A variable of type class is called reference variable.

→ Reference variable is used to store an address/hashcode of object.

→ A reference variable is required in order to reach an object by program.

## Syntax :-

|                                |
|--------------------------------|
| class-type-name variable-name; |
|--------------------------------|

Ex:- Book book1, book2;  
book1 = new Book();  
book2 = new Book();

object

Reference variables.

not object, but it holds the reference of book object.

reachable  
object

unreachable  
object.

→ Reference variable always refers a non-static class.

Q) what is reachable object or referenced object?

→ An object binds with reference variable is called reachable object.

Q) what is unreachable object or unreferenced object?

→ An object which does not bind with any reference is called unreachable objects.

→ All unreachable objects automatically removed by JVM.

Example:-

```
Class A
{
    int x;
    non-static
    {
        S.O.P(x);
    }
}
```

\* → The above program display compilation error, bcz non-static members of class can't access by static members.

→ In order to access we need to create object.

Members declared/used by the variables declared within the class:

{  
① private  
② public  
③ protected  
④ default

⑤ static  
⑥ final  
⑦ transient  
⑧ volatile

→ access modifiers

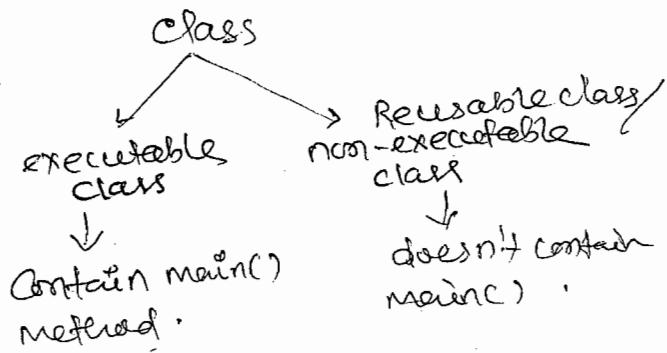
access specifiers

## Creating class & object:

Date-22/9/12

### Class A

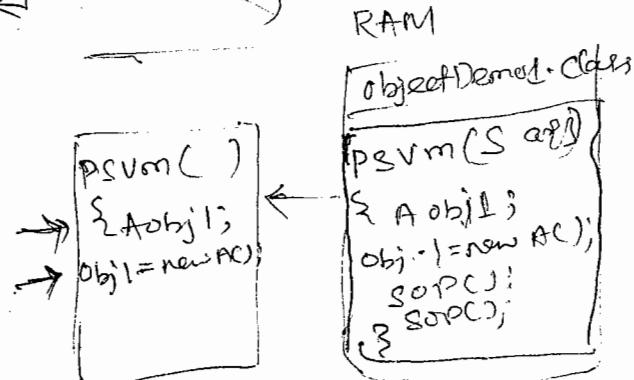
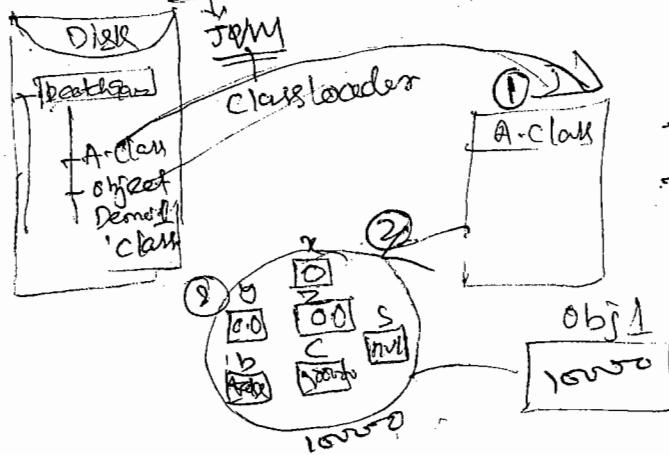
```
{
    int x;
    float y;
    double z;
    boolean b;
    char c;
    String s;
    String str;
}
```



### Class ObjectDemo1

```
{
    public static void main (String args[])
    {
        A obj1;
        obj1 = new A();
        System.out.println (obj1.x);
        System.out.println (obj1.y);
        System.out.println (obj1.z);
        System.out.println (obj1.b);
        System.out.println (obj1.c);
        System.out.println (obj1.s);
    }
}
```

SIP:  
javac objectDemo1.java  
java objectDemo1



\* It takes only main cause  
Interprete job is not storage  
but just execute.

## Access Specifiers:

```

    Package
    public class A
    {
        int x;
        private int y;
        protected int z;
        public int p;
    }

```

```

ObjectDemo2.java
class ObjectDemo2
{
    public void main(String args[])
    {
        A obj1;
        obj1 = new A();
        System.out.println(obj1.x); → X
        System.out.println(obj1.y); → X
        System.out.println(obj1.z); → X
        System.out.println(obj1.p); → ✓
    }
}

```

- Private members can't access outside the class & package.
  - Default members cannot access outside the package. but it can accessible outside the class.
  - Public members are accessible in anywhere outside the class & package.
  - Protected members also not access by outside the package.
- Example:

```

class Book
{
    private String bname;
    private String cname;
    private float price;
}

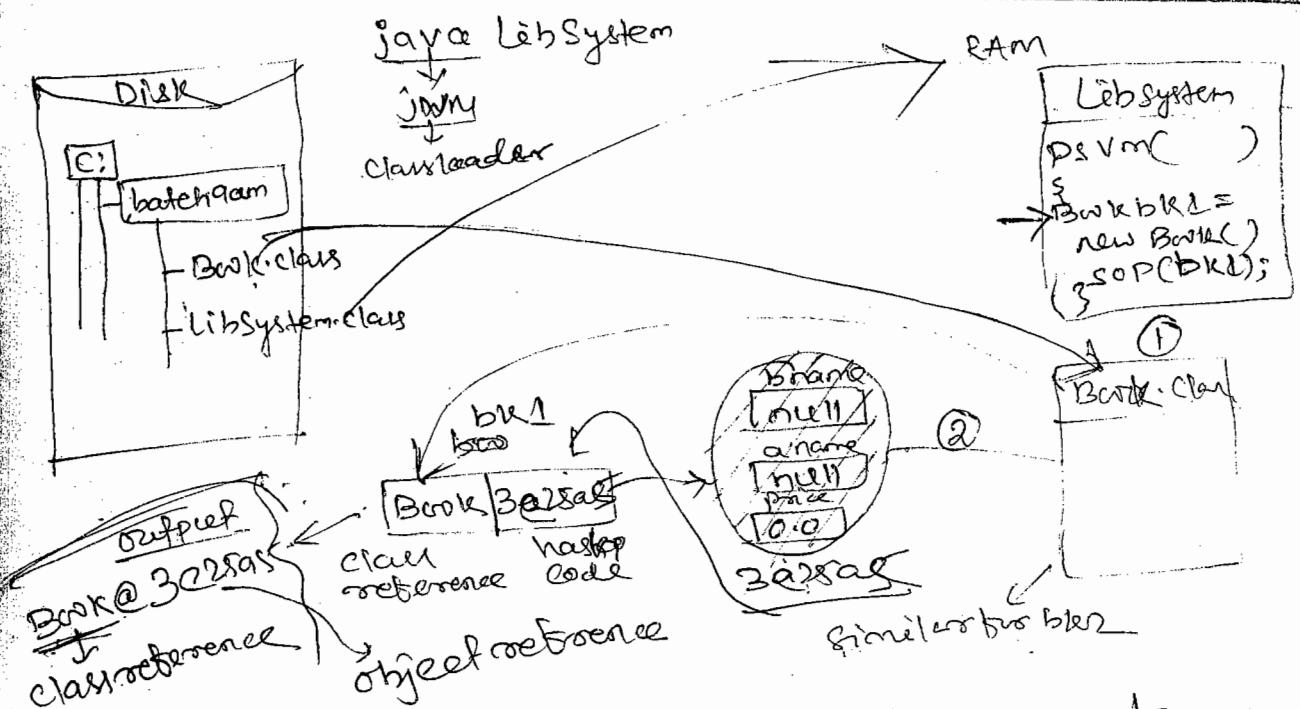
```

\* Data hiding can be achieved by using the variable as private scope.

```

class LibSystem
{
    public static void main(String args[])
    {
        Book bk1 = new Book();
        System.out.println(bk1);
        Book bk2 = new Book();
        System.out.println(bk2);
    }
}

```



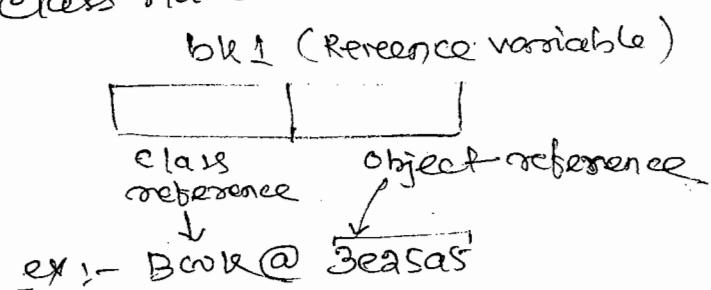
→ Reference variable is divided into 2 parts -

- 1.) object reference
- 2.) class reference

→ Object created with recent time is referred with hashCode.

→ A class loaded within RAM during recent time is referred with class name.

ex:-



\* → Data hiding in object oriented is achieved by declaring variables within class as private.

→ Private members cannot access outside the class.

→ `int a;`  
`a=null;` // invalid as null is reference value,

→ Class is loaded with class name & object is loaded with hashCode (hexadecimal no.).

~~QUESTION~~

Class A

```
{ int x;  
}
```

@java ObjectDemo2

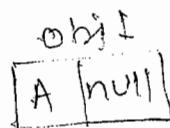
```
{ public static void main (String args [ ] )
```

```
{ A obj1 = null;
```

```
 S.O.P (obj1.x);
```

```
 } // S.O.P (obj1.y); → error
```

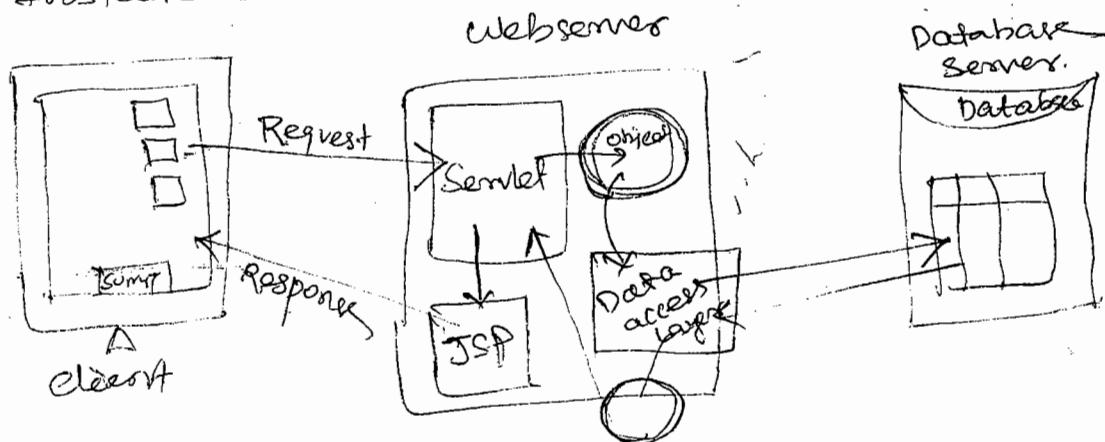
```
}
```



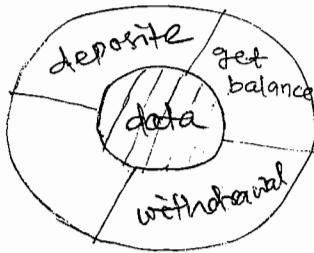
\* → ~~error~~ The above program displays an error during runtime becoz non-static members cannot access without creating objects.

Date - 23/9/12

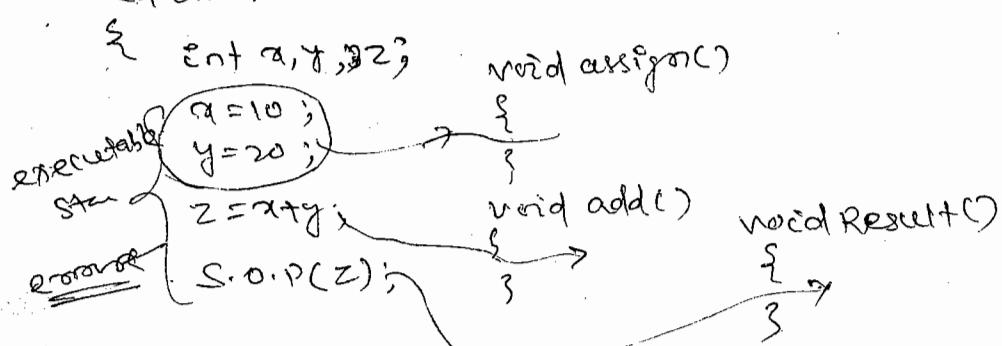
- Instance variables does not bind with class name.
- Instance variables bind with object name or reference.
- Instance variables cannot access without creating objects.
- Instance variables define state of an object.



## methods :-



- A method define behaviour of ~~an~~ object.
- In OOP any operations performed on object is represented as method.
- An object communicate with another object by ~~method~~ using class sum.



- Any executable statement must be present write inside the method.

- methods are of 2 types —

- 1) Instance method.
- 2) Class method.

- A small piece of program inside the class is called method.

- A method is a function which bind with either class or object.

### Instance method :

- Non-static method of class is called instance method.

- This method bind with object name.

- This method cannot called without creating object.

## Syntax:-

```
[modifiers] return-type method-name (parameters)
{
    Statements ;
}
```

- Java does not allow to write method without return type.
- If method does not return any value it should be define with void.
- Instance method performs object operations.

Ex:-

```
class Account
{
    private int accno;
    private String name;
    private float balance;
    void setAccount()
    {
        accno = 101;
        name = "Prem";
        balance = 5000f;
    }
    void printAccount()
    {
        S.O.P (accno);
        S.O.P (name);
        S.O.P (balance);
    }
}
```

Class is loaded with class name  
→ Account is a object bcoz we can create class on the view of object.

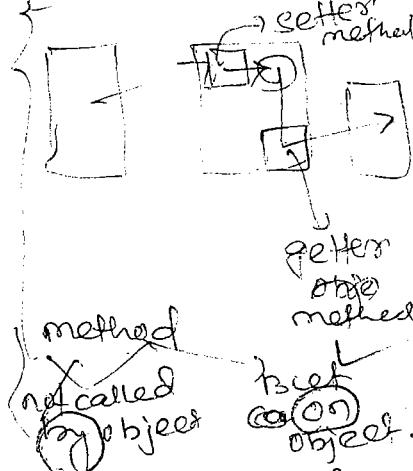
```
class Bank
{
    public static void main (String args[])
    {
        Account acc1, acc2;
```

```

acc1 = new Account();
acc2 = new Account();
acc1.setAccount();
acc2.setAccount();
acc1.printAccount();
acc2.printAccount();
}
}
}

```

\* immutable objects  
→ if we don't access  
the value i.e private  
members.



→ A method within class performs 2 types of operations.

- 1) Setter operation
- 2) Getter operation.

Q) what is Setter method?

→ An operation which changes state of the object is called setter method / modifier method.

Q) what is Getter method?

An operation which does not change the state of object is called getter method. This performs accessing operations.

Date - 24/9/12

Method with parameters:

→ method having parameter receives values and performing operation on object.

→ One method communicate with another method by passing values.

→ Java supports two ways of calling method.

- 1) Pass by value
- 2) Pass by reference

A  
B

met  
ch  
t  
&  
c

- method parameters are called local variables.
- These variables exist until execution of method.

### Class Account

```
{  
    private int accno;  
    private string name;  
    private float balance;  
    void setAccount (int a, String b, float b)  
    {  
        accno = a;  
        name = n;  
        balance = b;  
    }  
}
```

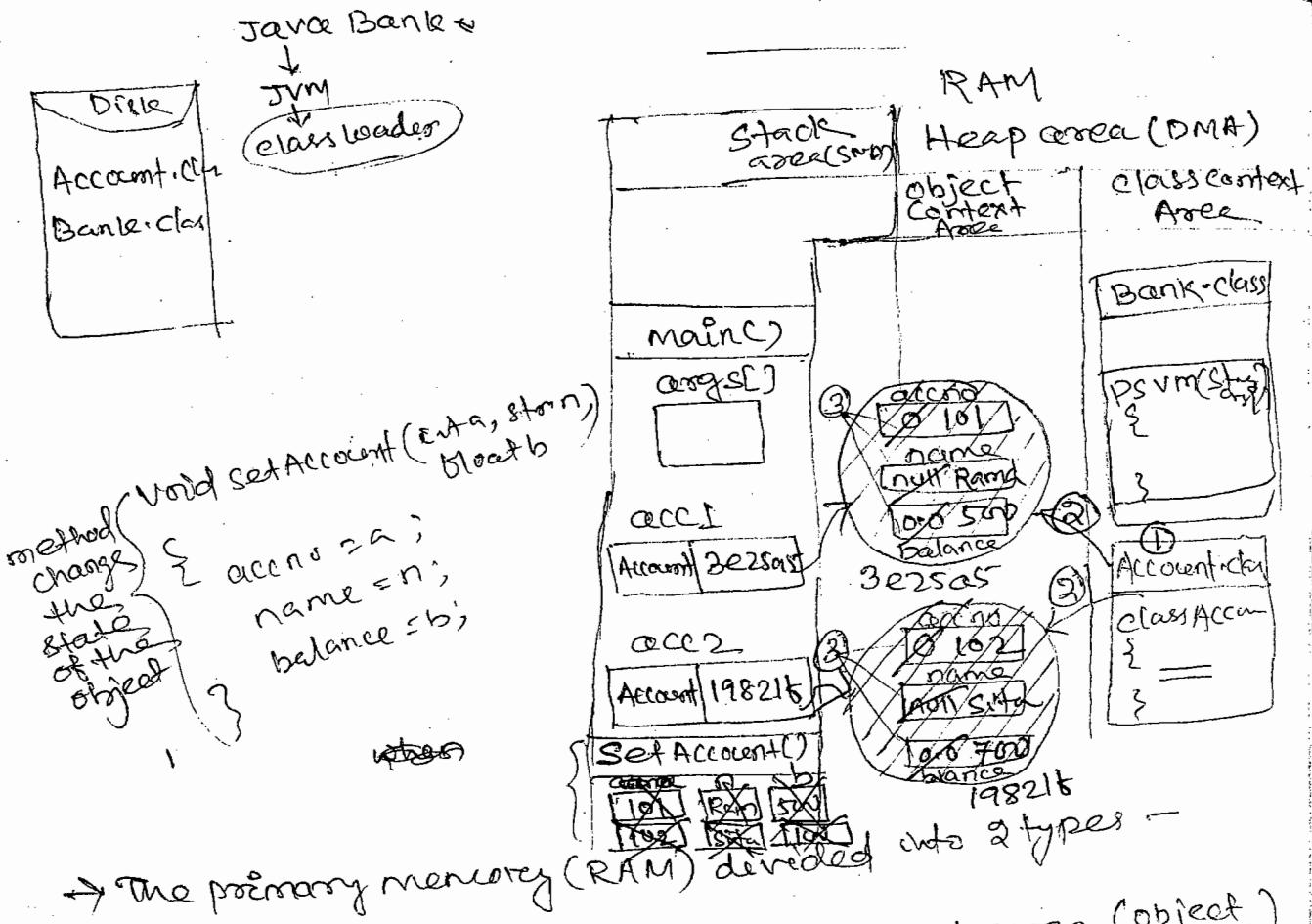
### void printAccount()

```
{  
    S.O.P (accno);  
    S.O.P (name);  
    S.O.P (balance);  
}
```

### Class Bank

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

```
{  
    Account acc1 = new Account ();  
    Account acc2 = new Account ();  
    S.O.P (acc1);  
    S.O.P (acc2);  
    acc1.setAccount (101, "Rama", 5000f);  
    acc2.setAccount (102, "Sita", 6000f);  
    acc1.printAccount ();  
    acc2.printAccount ();  
}
```



→ The primary memory (RAM) divided into object

- ① Stack area → object context area (object stored)
- ② Heap area → class context area (classes)  
→ shared areas

→ When stack area is full it send a msg that "Stack overflow".

→ methods are stored in stack area

→ JVM calls the method by sending hashcode of object on which it performs operation or correctly binds.

Q) What is binding :-

Linking method call with method body is called binding.

example:-

### Class Product

```
{ private int pid;
    private string pname;
    private float price;
    void SetpidName(int P, string pn)
    {
        pid = P;
        pname = pn;
    }
    void setPrice(float P)
    {
        price = P;
    }
    void point()
    {
        S.O.Pf("%d %s %f", pid, pname,
               price);
    }
}
```

### Class Shopping

```
{ public static void main(string args[])
{
    Product p1 = new Product();
    Product p2 = new Product();
    p1.setpidName(101, "Mouse");
    p1.setPrice(200f);
    p1.point();
    p2.setpidName(102, "Keyboard");
    p2.setPrice(300f);
    p2.point();
}
```

Q) What is immutable object?

- An object whose state (values) cannot be changed after creating with initial values is called immutable object.
- This ~~class~~ class doesn't provide setter methods.

Q) What is mutable object?

- An object whose state (values) can be changed after creating with initial values is called mutable object.

Date - 25/9/12

Ex - 1

Class A

```
{ private int x;  
private int y;  
}
```

Class B

```
{ psVm()  
A obj1 = new AC();  
obj1.x = 100; → X  
obj1.y = 200 → X.  
}
```

→ The above is an example of immutable object coz the initial values cannot change.

→ No setter method in immutable object.

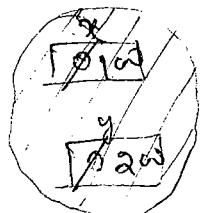
Ex - 2 :-

Class A

```
{ private int x;  
private int y;  
void setXY()  
{  
x = 100;  
y = 200;  
}
```

Class B

```
{ psVm(String args[1])  
{  
A ob = new AC();  
ob.x = 100; → X  
ob.y = 200; → X  
ob.setXY();  
}
```

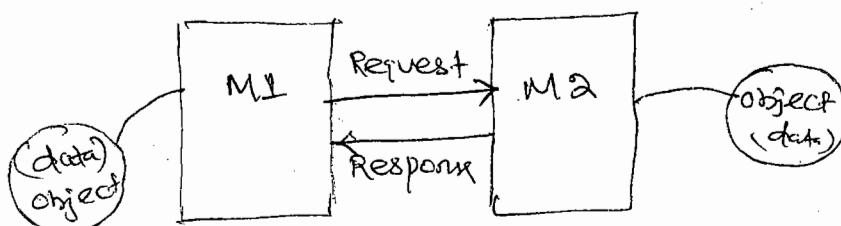


→ The above is an example of mutable object

→ Setter method must be present

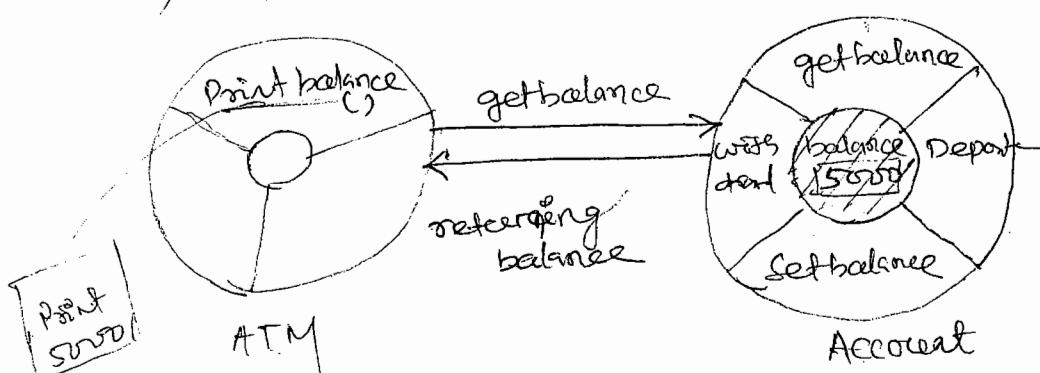
## Method with reference type :-

- A method having reference type as void  
can't return any value.
- A method performs operation on object and return values.



- Calling method
- Called method
- This method have return type.
- always having parameter.

- A method can have return type as -
- 1) Primitive type → it returns value.
- 2) Reference type → it returns object.



Ex :-

@Class Accout

```

{
    private int accno;
    private float balance;
    void setAccout(int a, float b)
    {
        accno = a;
        balance = b;
    }
}
  
```

```

float getBalance()
{
    return balance;
}

```

Class ATM

```

public static void main(String args[])
{

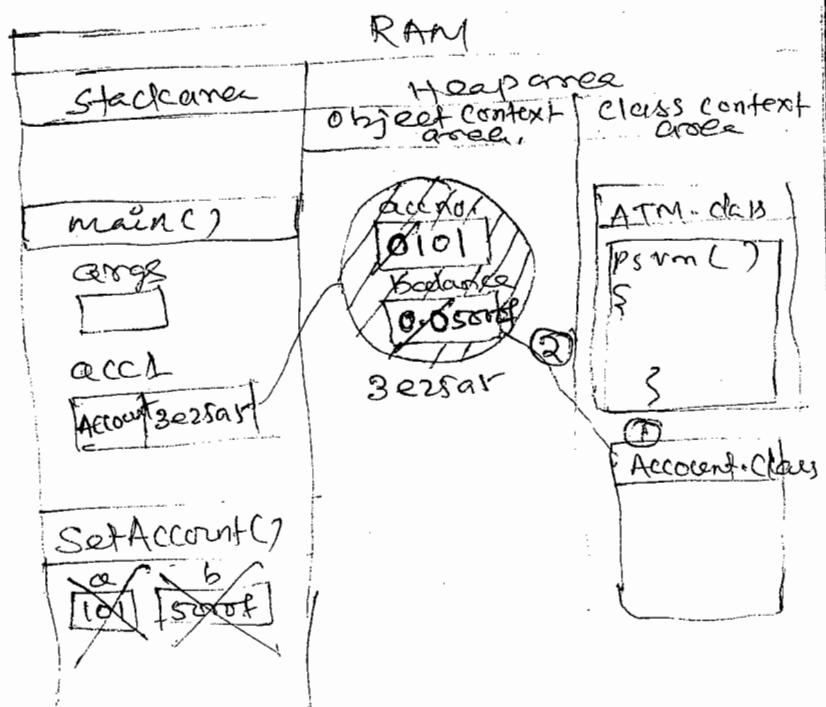
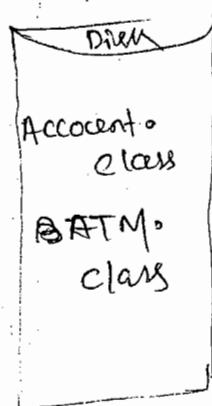
```

```

    Account acc1 = new Account();
    acc1.setAccount(101, 5000f);
    float bal = acc1.getBalance();
    System.out.println("Balance" + bal);
}

```

java ATM <



→ Every method should have a method stack area.  
so that it must create their local variables.

example

class Book

```
{ private String bname;
  private float price;
  void setBook(String bname, float price)
  {
    this.bname = bname;
    this.price = price;
  }
```

```
String getBook()
```

```
{ return bname + " " + price;
}
```

}

class BooksLib

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

```
  Book bk1 = new Book();
  bk1.setBook("Java", 50f);
  String b = bk1.getBook();
  System.out.println(b);
}
```

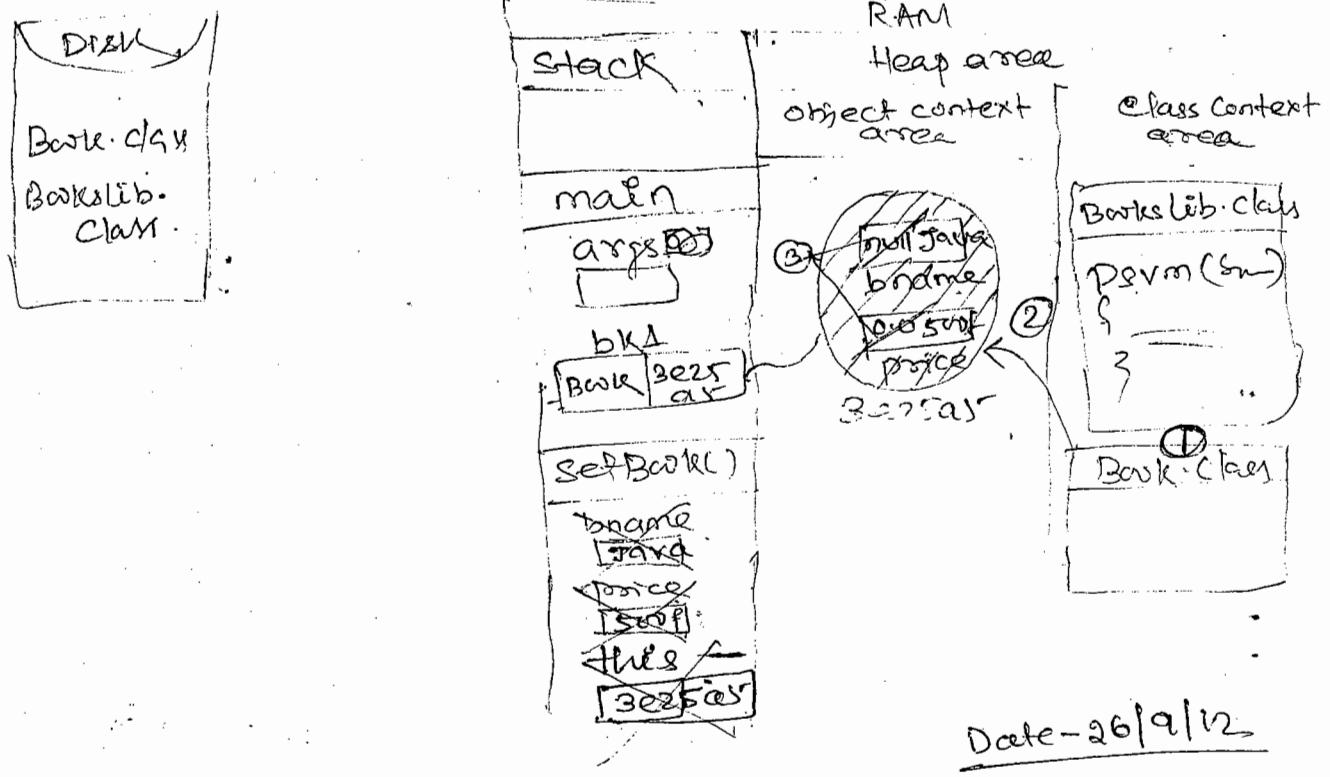
Note:-

statements

↓  
executable

↓  
declaratory

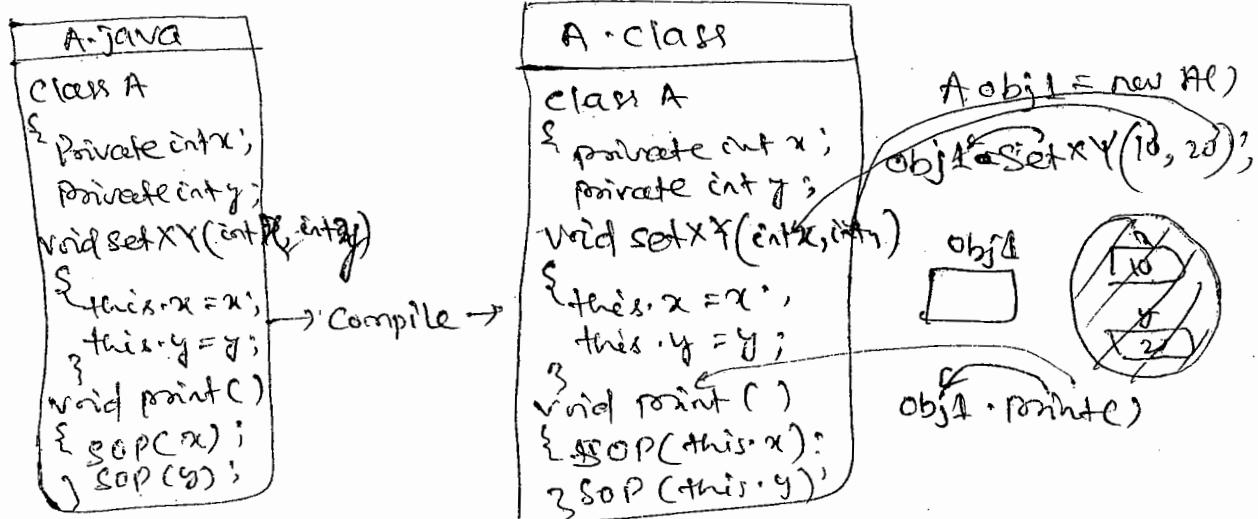
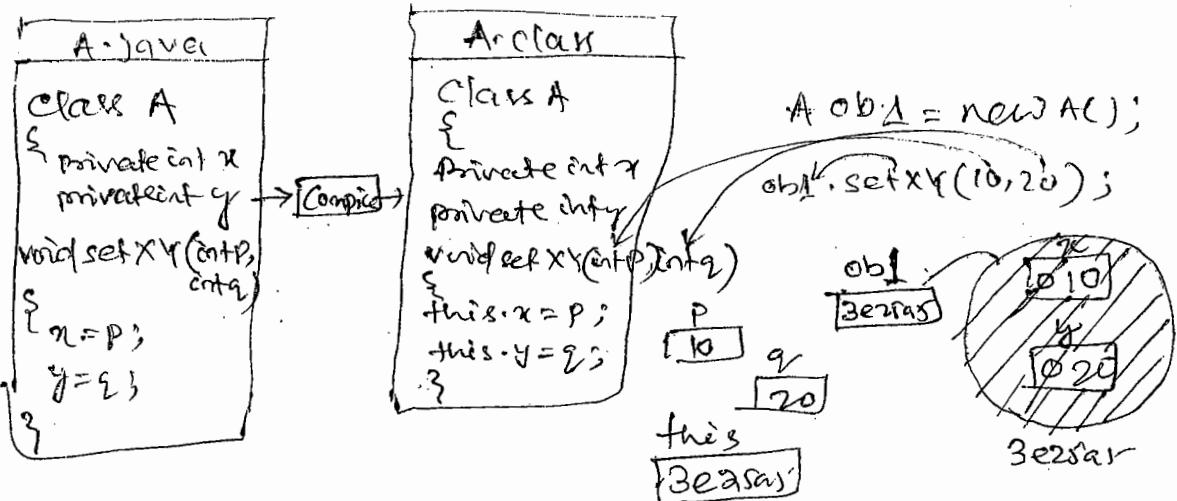
↓  
of always resides  
inside method.



Date - 26/9/12

this reference:

- Every non static method of class/java compiler add a reference variable "this".
- "this" is a keyword.
- It is a reference variable which stores/holds the hashcode of current object, on which method performs op<sup>n</sup>.
- "this" is a constant variable whose value is never changed.
- It instance & local variables are declared with same, instance variable of current object can access using "this" reference. (it must explicitly done)
- "this" is a local variable.
- "this" reference can be implicitly call. i.e it should be call by compiler implicitly.
- Instance method always called by sending the hashcode of object i.e stored in "this" pointer.



Program: —

Class Account

```

private int accno;
private float balance;
void setAccount (int accno, float balance)
{
    this.accno = accno;
    this.balance = balance;
}
void deposite (float tamt)
{
    balance = balance + tamt;
}
void withdraw (float tamt)
{
}

```

if (amt > balance)  
S.O.P ("Insufficient balance");  
else  
balance = balance - amt;

}  
String getAccount()  
{  
referen accno + " " + balance;  
}  
}

Class Bank  
{ public static void main (String args[])

{ Account acc1 = new Account();  
acc1.setAccount (101, 5000f);

String a;  
a = acc1.getAccount();

S.O.P (a);

acc1.deposite (5000f);

a = acc1.getAccount();

S.O.P (a);

acc1.withdraw (2000f);

a = acc1.getAccount();

S.O.P (a);

}

Method overloading:

→ Define more than one method with same name by changing :-

1. no. of parameters.
2. types of parameters.
3. order of parameters.

is called overloaded method.

When more than

Q) When to overload method?

When more than one method having similar operations but different implementation, method is overloaded.

Q) Why should I overload method?

1) Simplicity:—

As a developer / programmer does not required to remember no. of method name.

2) Extensibility:—

Adding new features <sup>in</sup> to existing methods without modifying it.

3) Reusability:—

Allows to reuse method name & functionality.

→ Overloaded method having Polymorphic behaviour.

→ Defining one thing in many form is called "polymorphism".

→ This is called name polymorphism or ad-hoc polymorphism.

Class A

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

error :-  
becoz we cannot write two methods with same signature.

Date - 23/9/12

→ Compiler recognise method with its signature, which include no. of parameters, types of parameters and order of parameters.

od  
Class A

```
{ int m1() { }  
float m1() { }  
}
```

→ same signature

OP Error :- (invalid overloading)

→ compiler

Class A

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

signature = method name + parameters.

OP :- valid overloading.

Program :-

Class Employee

```
{ private int empno;  
private String ename;  
private float Salary;  
void setEmployee( int empno, String ename)  
{  
    this.empno = empno;  
    this.ename = ename;  
}
```

L. 1

```
void setEmployee (int empno, String ename,  
float salary)
```

```
{  
    this.ename = ename;  
    this.empno = empno;  
    this.salary = salary;  
}
```

```
String getEmployee ()
```

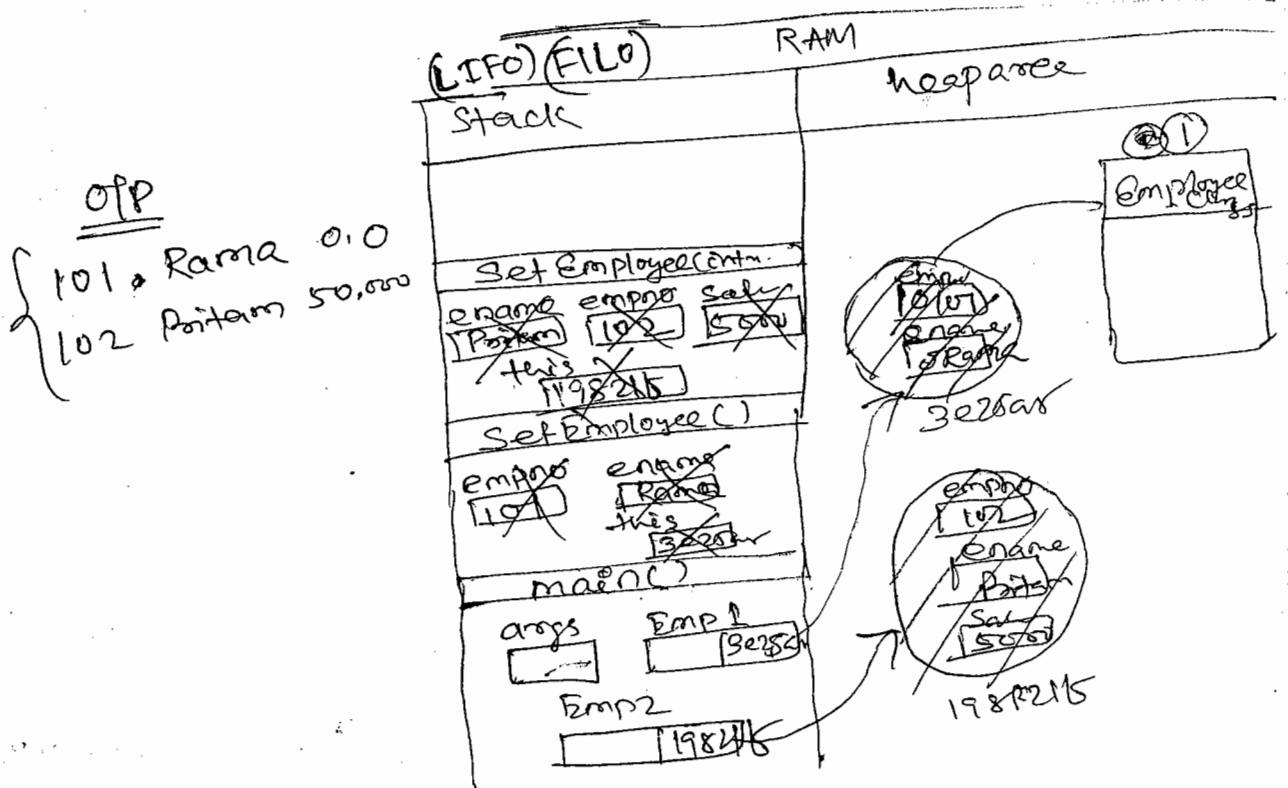
```
{  
    return empno + " " + ename + " " +  
          salary;  
}
```

```
class Payroll
```

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

```
{  
    Employee Emp1 = new Employee ();  
    Employee Emp2 = new Employee ();  
    Emp1.setEmployee (101, "Rama");  
    Emp1.setEmployee (102, "Pratam", 5000);  
    String e1 = Emp1.getEmployee ();  
    String e2 = Emp2.getEmployee ();  
    System.out.println (e1);  
    System.out.println (e2);  
}
```

The behaviour (selection of method) is done  
during compilation. So this is called  
compile time polymorphism.



### Compile time polymorphism:

→ method overloading exhibit compile time polymorphism.

- As selection of method is done during compilation based on no. of parameters, types of parameters or order of parameters.
- The method which is selected by compiler same method is executed by JVM during runtime.
- As behaviour of this method is changed at compile time, it is called compile time polymorphism.

(1)

## Program:-

### Class Account

```
{ private int accno;  
private String name;  
private float balance;  
void setAccount(int accno, String name)
```

```
{ this.accno = accno;  
this.name = name;  
} //without balance  
void setAccount(int accno, String name, float balance)
```

```
{ setAccount(accno, name);  
this.balance = balance;  
} //with balance
```

```
String getAccount()
```

```
{ return accno + " " + name + " " + balance;
```

```
}
```

### Class Bank

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

```
{ Account acc1 = new Account();  
Account acc2 = new Account();  
acc1.setAccount(101, "Rama");  
acc2.setAccount(102, "Sita", 5000f);
```

```
String a1 = acc1.getAccount();
```

```
String a2 = acc2.getAccount();
```

```
S.O.P(a1);
```

```
S.O.P(a2);
```

```
}
```

### Note :-

#### ① Class A

```

    {
        void point (double d)
        {
            System.out.println ("Inside double");
        }
        void point (float f)
        {
            System.out.println ("Inside float");
        }
    }
  
```

#### class ObjectDemo9

```

    {
        public static void main (String args[])
        {
            A obj1 = new A ();
            obj1.point (10);
        }
    }
  
```

### O/P:-

javac ObjectDemo9

java ObjectDemo9

Inside float method.

→ Compiler always look for a method having

similar type parameters.

→ If it does not found similar type parameter it look for method having border datatype parameters.

→ If method with border datatype parameters not available it leads to compile time error.

### Ex:-

#### ② Class A

```

    {
        void point (byte b)
        {
            System.out.println ("Inside byte");
        }
        void point (int x)
        {
            System.out.println ("Inside int");
        }
    }
  
```

```

class ObjectDemo10
{
    public static void main(String args[])
    {
        Aobj1 = new AC();
        obj1.point(65);
        {
            byte a = 65;
            obj1.point(a);
        }
    }
}

```

O/P Inside Integer method  
Inside Byte

→ If the type is not specified, then it is by default integer.

(3) class A

```

void point(Byte b)
{
    SOP("Inside byte");
}

void point(Char ch)
{
    SOP("Inside character");
}

```

class ObjectDemo11

```

public void (String args[])
{
    Aobj1 = new AC();
    obj1.point(65);
}

```

}

O/P :- Compiler error

→ This above program display compiler time error becoz there is no method of type integer.

Date - 28/9/12

## Order of parameters :-

### Class A

```
{ void xx M1( int x, float y )  
{  
    }  
    void M1 ( float x, int y )  
    {  
        }  
    }
```

### Static :-

~~xxxxxxxx~~

→ Static is a modifier used for declaring,

- Variables
- methods
- blocks

### Static variables :-

~~x =~~ → Static variables are class variables for which memory is allocated on loading of class.

→ Java does not support global variables.

→ In order to declare global variable, which is global to more than one object, it is declared with static modifier.

### Syntax :-

```
static data-type variable-name;
```

→ The static variable is loaded in class context.

→ Static variables are shared by more than one object.

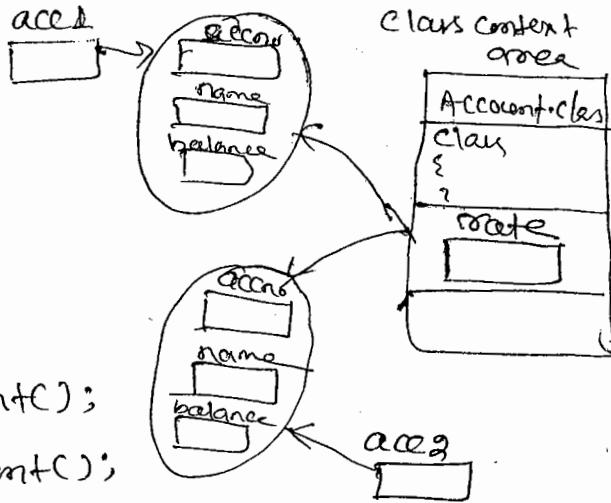
→ Static variables bind with class name or object name.

### Class Account

```
{ int accno;  
    String balance; name;  
    float balance;  
    static float rate;  
}
```

```
Account acc1 = new Account();
```

```
Account acc2 = new Account();
```



→ In order to declare a variable which is common for more than one object declared as static.

Q. Difference b/w static & non-static variables?

#### Non-static Variables

① Non-static variables of a class called <sup>one</sup> instance variable.

② Memory for this variable allocated on creation of object.

③ These variable belongs to object. So every object having its own copy of instance variables.

④ It is bind with object name.

⑤ Memory for this allocated in object context area.

#### static Variables

① static variables of a class called class variables.

② Memory for this variable allocated on loading of class.

③ These variables belongs to class, more than one object share static variable. becoz it is a global variable.

④ It is bind with class name or object name.

⑤ The memory for this allocated in class context area.

Ex:-

Class A

```
{ int x = 10;  
static int y = 20;
```

}

Class static Demo1

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

{

```
    System.out.println (A.y); // valid
```

```
    System.out.println (A.x); // invalid  
        non-static variable  
        cannot access by  
        class name.
```

```
    A obj1 = new A();
```

```
    A obj2 = new A();
```

```
    S.o.p (obj1.x);
```

```
    } S.o.p (obj2.x);
```

```
    } S.o.p (obj1.y);
```

```
    S.o.p (obj2.y);
```

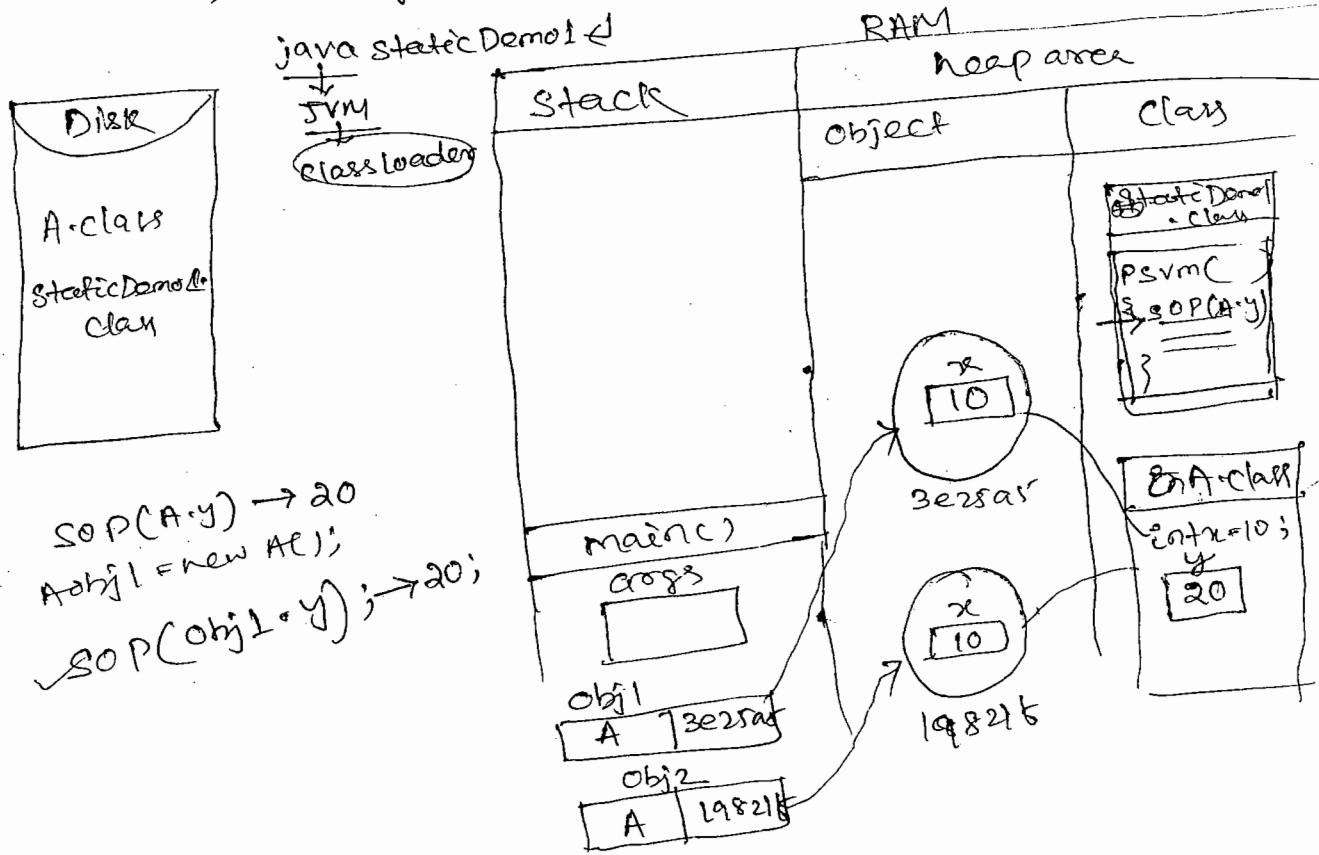
Note: → Class can be loaded into RAM using 3 types —

by using —

1) new operator

2) Class.forName ("classname");

3) Accessing static member of a class.



Note:-

Class A

{ void mac()

{ static int x;

}

→ This is an error.

→ The above program display compilation error  
becoz static modifier is not allowed for local  
variables.

→ Both C, supports but Java doesn't support it.

Program:-

Class Account

{ private int accno;

private float balance;

static float rate;

void setAccount (int accno, float balance)

{ this.accno = accno;

this.balance = balance;

}

float getInterest()

{

: reteven (balance \* rate \* 6) / 100;

}

String getAccount()

{

: reteven accno + " " + balance;

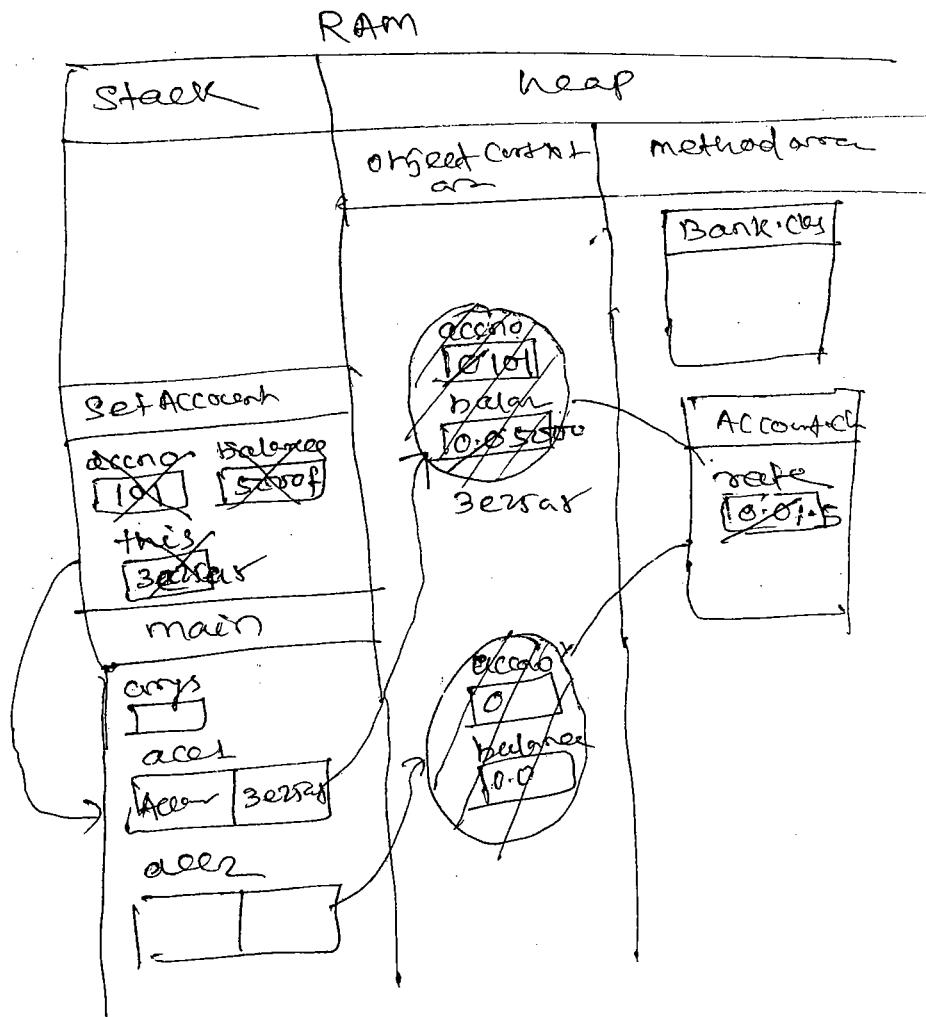
}

{}

## Class Bank

```
{ public static void main (String args[])
{ Account acc1 = new Account();
  Account acc2 = new Account();
  acc1.setAccount(101, 5000f);
  acc2.setAccount(102, 6000f);
  Account::rate = 1.5f;
  float i1 = acc1.getInterest();
  float i2 = acc2.getInterest();
  S.O.P(i1);
  S.O.P(i2);
  String e1 = acc1.getAccount();
  String e2 = acc2.getAccount();
  System.out.printf("%s %s", e1, e2);
}
```

3 3



## Class A

```
{ int x = 10;
```

```
 static int y = 20;
```

{

## Class B

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

```
{ A obj1 = new A();
```

```
A obj2 = new A();
```

```
s.out.println ("x.y", obj1.x, obj1.y);
```

```
s.out.println ("x.y", obj2.x, obj2.y);
```

```
obj1.y = 40;
```

```
obj2.y = 90;
```

```
System.out.println ("x.y", obj1.x, obj1.y);
```

```
System.out.println ("x.y", obj2.x, obj2.y);
```

{}

Date - 29/9/12

## Static methods:

→ The methods define within class can be static.

→ Static methods are class methods.

→ These methods are bind with class name or objectname.

→ It is a global method, which is global to more than one class or object.

→ An operation which doesn't involve on creation of object, that operation or method declared as static.

## Syntax :-

```
static return-type method-name (parameters)
```

```
{ statements;
```

{

Q) What is the diff. b/w static method & non-static methods?

### Non-static

(1) It's called instance method.

(2) It is bind with object name.

(3) To call this method required to create object.

(4) It is with "this" reference.

(5) It access both static & non-static members of class.

(6) It allows Run-time polymorphism.

(7) It defines object op<sup>n</sup>.

example:-

~~ext:~~  
Class A

```
{   Ent x = 10;  
    static void print()  
    { S.O.P (x);  
    } }
```

→ The above program displays an error because static method can't access non static variable.

### Static

(1) It is called class method.

(2) It is bind with object name or class name.

(3) The method is called without creating object.

(4) It is without "this" reference.

(5) It can access only static members of a class; not access non static members.

(6) It doesn't allow run-time polymorphism.

(7) It defines class operations.

Ex-2 :-

Class StaticDemo2

{ int x = 10;

public static void main(String args[])

{ S.o.p(x);

}

→ The above program display compile time error

becoz main is static method which can't access 'x'.

→ For invoking non-static we must create objects.

Ex-3 :-

Class Math

{ static int sqr(int)

{ refeeren n\*n;

{ static int max(int x, int y)

{ if (x > y)

refeeren x;

else

refeeren y;

}

Class StaticDemo3

{ public static void main(String args[])

{ int s = Math.sqr(5);

int m = Math.max(20, 10);

S.o.p("Sqr is " + s);

S.o.p("Max is " + m);

}

Ex-4:

Class A

{ void m1()

{ S.O.P("Inside non static method");

} static void m2()

{ S.O.P("Inside static method");

}

Class static Demo 4

{ public static void main(String args[])

{ A.m1(); // Error →

A.m2();

}

}

(A.m1()) ~~obj~~

non-static members can't access in static context

area -

Ex-5:

Class A

{ void m1()

{

}

static void m2()

{

}

Class static Demo 4

{ Psvm (String args[])

{ Obj1 = new A();

Obj1.m1();

Obj2.m2();

}

}

QX-6 :-

### Class Account

```
{ private int accno;
    private float balance;
    private static float rate;
    static void setRate( float rate )
    {
        Account::rate = rate;
    }
    void setAccount( int accno, float balance )
    {
        this.accno = accno;
        this.balance = balance;
    }
    void deposit( float tamt )
    {
        balance = balance + tamt;
    }
    static float getRate()
    {
        return rate;
    }
    String getAccount()
    {
        return accno + " " + balance + " " + rate;
    }
}
```

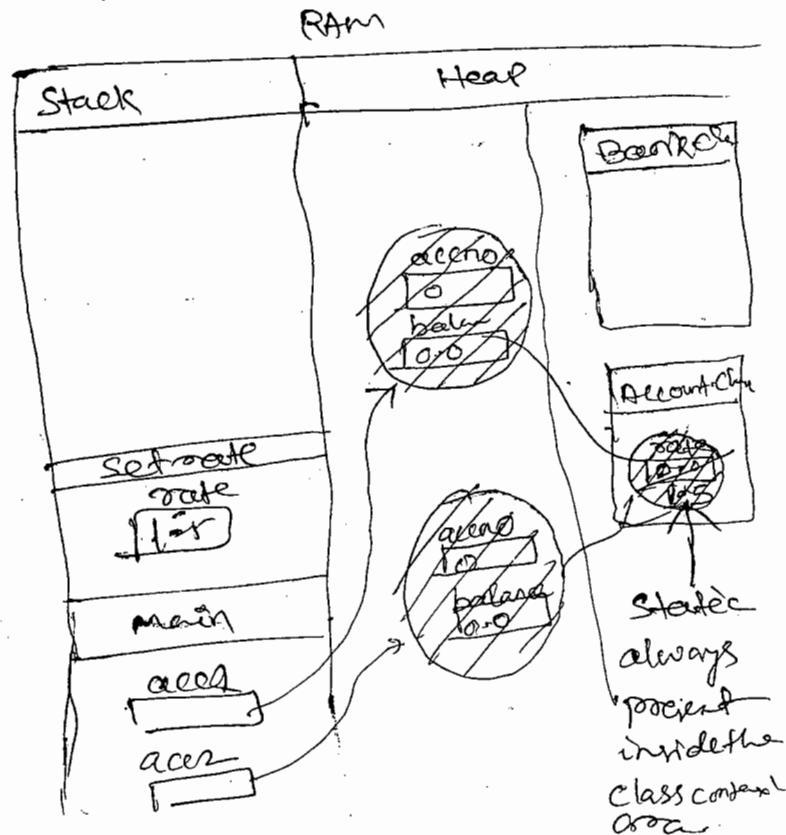
### Class Bank

```
{ public static void main( String args[] )
{
    Account::setRate( 1.5f );
    S.O.P( Account::getRate() );
    Account acc1 = new Account();
    acc1.setAccount( 101, 5000f );
    S.O.P( Account::getAccount() );
}
```

acc1.deposite(2000\$);  
S.O.P(acc1.getAccout());

{

3



### Class A

```
{ int x;  
  static int y;  
  void setXY( int x, int y )  
  { this.x = x;  
    this.y = y; (one) A.y = y ;  
  }
```

String gotXY()

```
{ return x + " " + y;  
}
```

{

class B

{ psvm (String arr[ ]) }

{ A obj1 = new AC();

obj1.setXY(10,20);

S.O.P( obj1.getXY());

}

→ Local variable of non-static method can be access static members of a class by using this reference or class name.

→ A non-static method can access static members of a class using this reference.

Final :

→ Final is a modifier used for declaring.

→ Final variable

- final parameters
- final methods
- final class.

Final variable :

→ The value of final variables never changes.

→ Final variables are constants.

→ These variables can be,

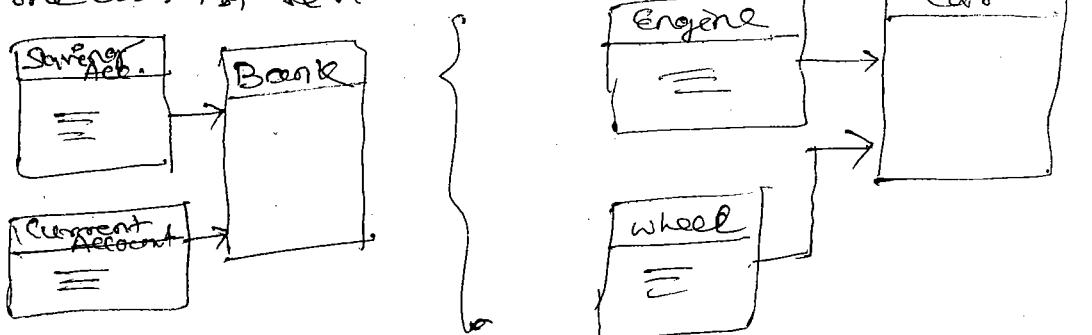
1. Final instance variable
2. Final class variable
3. Final local variable

## Class Reusability :-

→ Object oriented allows to use content of one class inside another class using 2 approaches — or methods —

1. Composition (bet' correlated classes)
2. Inheritance (bet' related classes)

→ If we write all ~~the~~ data in one class then it leads to ① redundancy & ② inconsistency. So we divide the one class in several class so.



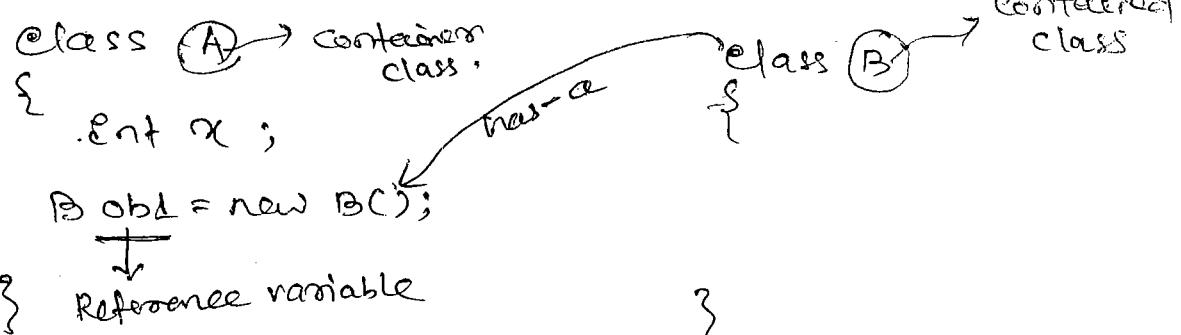
## Composition :-

→ It is a process of creating an object of one class inside another class.

→ It allows to use contents of one class inside another class but doesn't allow to extend. (i.e. it allows only reusability but not extending)

→ The relationship between classes in composition is called has-a relation.

→ This relationship is between correlated classes where one class doesn't share properties & behaviour of another class.



## Class System

```
class Pointstream  
{  
    static Pointstream *ref =  
        new Pointstream();  
}
```

```
System.out.println()  
" "  
" "
```

```
} class Pointstream  
{ void printin()  
{  
    void point()  
    {  
        void printf()  
        {  
            }  
        }  
    }  
}
```

→ In order to use contents of one class inside another class, we required a reference variable.

→ Reference variables can be declared within the class

1. static Reference Variable
2. non static reference Variable
3. Local reference Variables

### example:-

#### Class A

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

#### Class B

```
{  
    A obj = new A();  
    void m2()  
    {  
        S.O.P ("Inside m2() of class B");  
    }  
}
```

## class CompDemo1

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

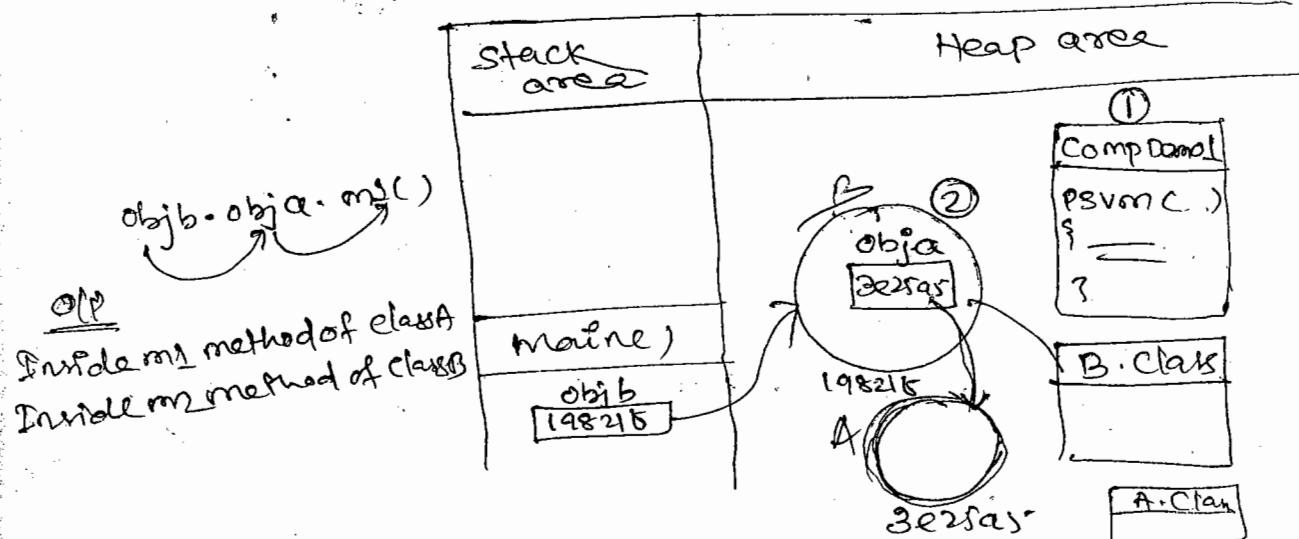
```
    B objb = new B();
}
```

```
    objb.obja.m1();
```

```
    objb.m2();
```

```
}
```

RAM



→ The reference variable may be private, public, default etc.

→ If we want to hide the object within another object . then we declare inner object as private.

example :-

Class A

```
{ int x;
  int y;
}
```

Class B

```
{ int P;
  int q;
}
```

```
A obja = new A();
```

}

class CompDemo2

```
{ public (String args[])
{
```

```
    B objb = new B();
}
```

```
    S.OP(objb.P); → 0

```

```
    S.OP(objb.q); → 0

```

```
    S.OP(objb.obja.x); → 0

```

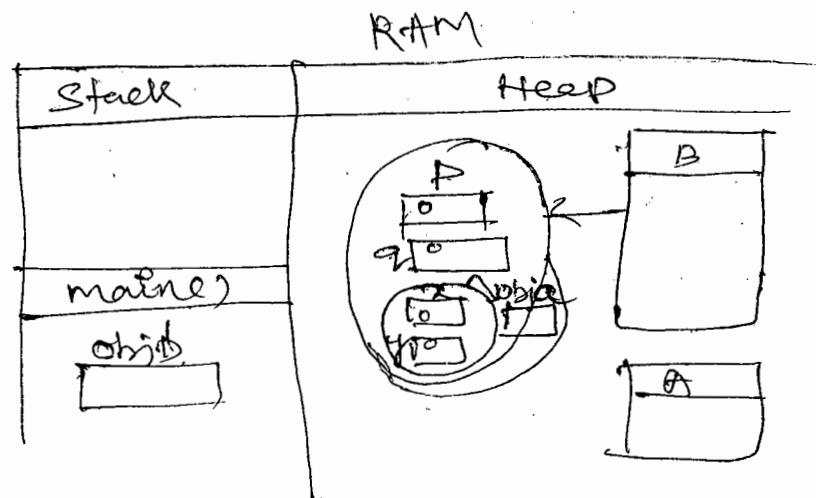
```
    S.OP(objb.obja.y); → 0

```

(Q)

```
S.O.println("In y.d y.d", objb.P, objb.q);
S.O.println("In y.d y.d", objb.obja.x,
            objb.obja.y);
```

- When object is created it allocate memory for non-static variables.
- In 'B' class 3 variables are there.



example - 3

Class A

```
{
    private int x, y;
    void setXY (int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

int getX()

reference x;

int getY()

reference y;

}

Class B

```
{
    private int p, q;
    A objA = new A();
    void setPQ (int p, int q)
    {
        this.p = p;
        this.q = q;
    }
}
```

```

int getPC()
{
    referenP;
}

int getQC()
{
    referenQ;
}
}

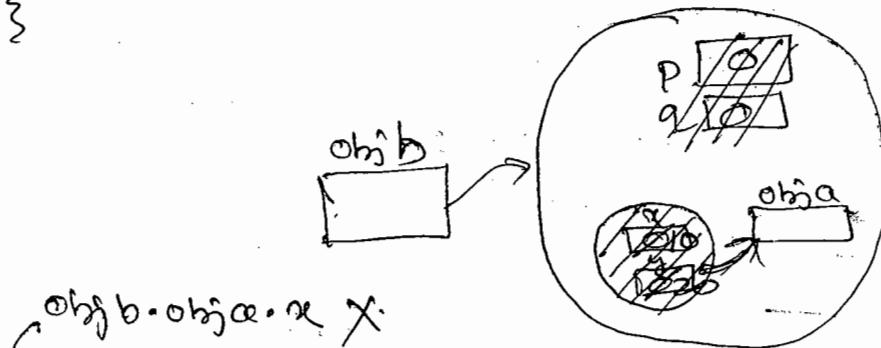
```

### Class CompDemo3

```

public static void main (String args[])
{
    Obj b = new B();
    obj.b.setPQ(10,20);
    obj.b.obj.a.setX(30,40);
    S.O.println("In %d %d", obj.b.getPC(), obj.b.getQC());
    S.O.println("In %d %d", obj.b.obj.a.getX(), obj.b.obj.a.getY());
}
}

```



$\rightarrow$  obj.b.obj.a.X  
 $\rightarrow$  obj.b.P  
 $\rightarrow$  obj.b.obj.a.setX(10,20); ✓

So

### example-4:

#### Class A

```

private int x;
void setX(int x)
{
    this.x=x;
}

int getX()
{
    referen x;
}
}

```

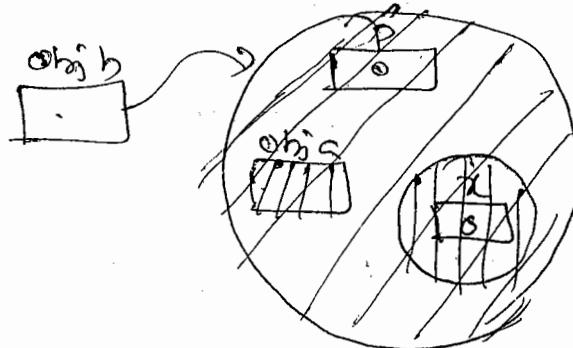
## Class B

```
{ private A obja = new A();  
private int p;  
void set (int x, int p) // container class must  
{ obja.setX(x); read value from its  
this.p = p; own & also its contained  
} class.  
String get()  
{ return obja.getX() + " " + p;  
}  
}
```

## Class CompDemo4

```
{ public static void main (String arg[]){  
B objb = new B();  
objb.set(10, 20);  
S.O.P (objb.get());  
}  
}
```

objb.obja → X



Date - 3/10/12

### Example

Static Reference Variable :-

- Static Reference variable is static within class
- So it is bind with class name.
- It allows to create global object, which is shared by more than one class or object.

example :-

Class P

{ void point (int x)

{ S.O.P (x);

}

void point (float x)

{ S.O.P (x);

}

}

Class Computer

{ static P ref = new P();

{

Class CompDemo5

{ public static void main (String args[])

{ Computer.ref.point (10);

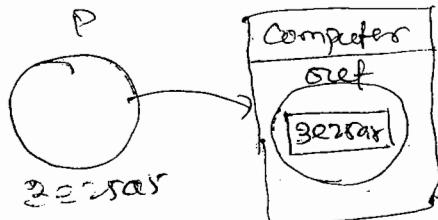
Computer.ref.point (1.5f);

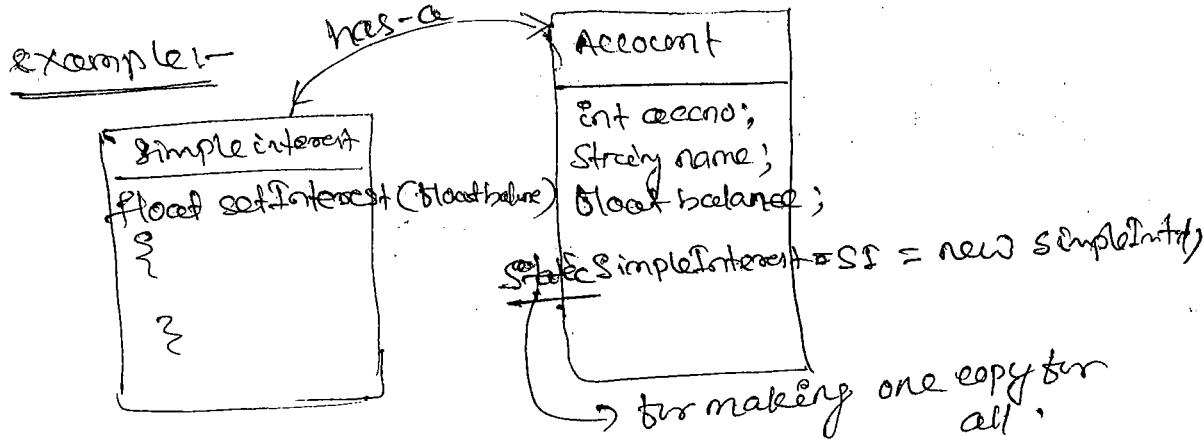
}

}

Output

10  
1.5f





Class SimpleInterest

```

{ float getInterest (float balance)
  {
    return (balance * 3 * 1.5f) / 100;
  }
}
  
```

Class Account

```

{ private int accno;
  private String name;
  private float balance;
  private static SimpleInterest si =
    new SimpleInterest();
}
  
```

```

void setAccount (int accno, String name,
                 float balance)
  
```

```

  {
    this.accno = accno;
    this.name = name;
    this.balance = balance;
  }
  
```

```

float getBalance ()
  
```

```

  {
    return si.getInterest(balance);
  }
  
```

Class Bank

```

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

```

  Account acc1 = new Account();
}
  
```

```

  Account acc2 = new Account();
}
  
```

```

  acc1.setAccount (101, "mike", 5000f);
}
  
```

```

  acc2.setAccount (102, "Sam", 6000f);
}
  
```

```

  float b1 = acc1.getBalance();
}
  
```

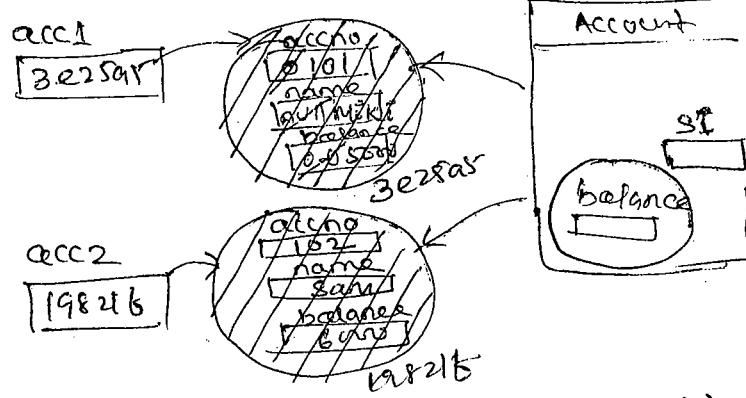
```

  float b2 = acc2.getBalance();
}
  
```

```

  System.out.println ("%f %f", b1, b2);
}
  
```

OLP  
5225.0  
6270.0



→ Reference variable always refers non-static class or a class.

class A

```
{ void m1()
{
}
static void m2()
{
    System.out.println("Inside m2()");
}
```

class B

```
{
A obj = new A();
void m3()
{
    A.m2(); ←
    obj.m1(); ←
}
```

→ Here m2() is static so we don't require to create an object it can be accessed by class name.

class B

```
{
A obj = new A(); ← It may be static or non-static
void m3()
```

```
{
A obj1 = new A(); ← local variable
=
```

```

}
void m4();
}
```

Method with parameters of type reference:

(Pass by Reference)

Java allows to call a method in 2 ways

1. pass by value
2. pass by Reference

- Passing reference to a method does not send address but send values called hashcode.
- This hashcode mapped with address of object.
- A method performs operation on object by receiving another object by receiving using method with parameters of type ~~reference~~<sup>reference</sup>.
- This is called "use-a" relation.
- These allow to perform operation on more than one object.

Date - 4/10/12

- Method with parameters of type pointer
  - pass by value.
- Method with parameters of type reference →
  - pass by reference. it is internally send a value not address called hashcode.
- If method is having parameters of ~~above~~  
~~(int type class)~~  
 below 4 types it receives hashcode of object
  1. Class
  2. Interface
  3. enum
  4. Array

example:-

```
class RefDemo1
{
    static void print(int a[])
    {
        for(i=0; i<a.length; i++)
            S.O.P(a[i])
    }
}
```

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

```
{ int b[] = {10, 20, 30, 40, 50};
```

```
    point(b);
```

```
}
```

```
point(b);
```



→ Here b ~~will~~ contain  
hashcode. So we  
send reference.

→ point() method is  
called by main()  
method.

if point() is not  
static then we  
must create object  
for this.

→ As main() function present in the same class so we ~~need not~~ <sup>do not</sup>  
required to call the class name for calling the point()  
method.

example: —

Class A

```
{ private int x;
```

```
private int y;
```

```
void set(int x, int y)
```

```
{ this.x = x;
```

```
    this.y = y;
```

```
}
```

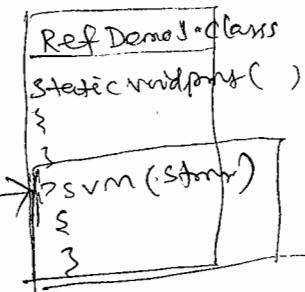
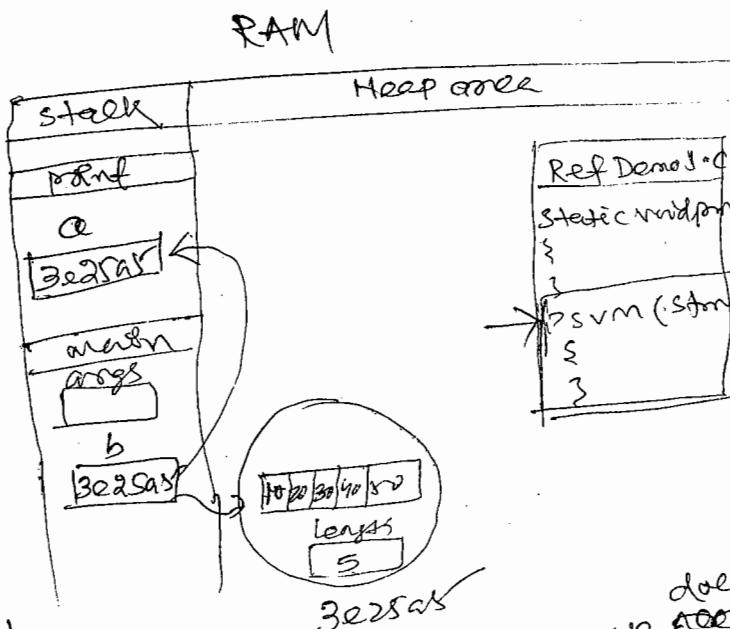
```
void compare(A obj)
```

```
{ if (x == obj.x & & y == obj.y)
```

```
    s.o.p("equal");
```

```
else
```

```
    s.o.p("not equal");
```



does not  
need not

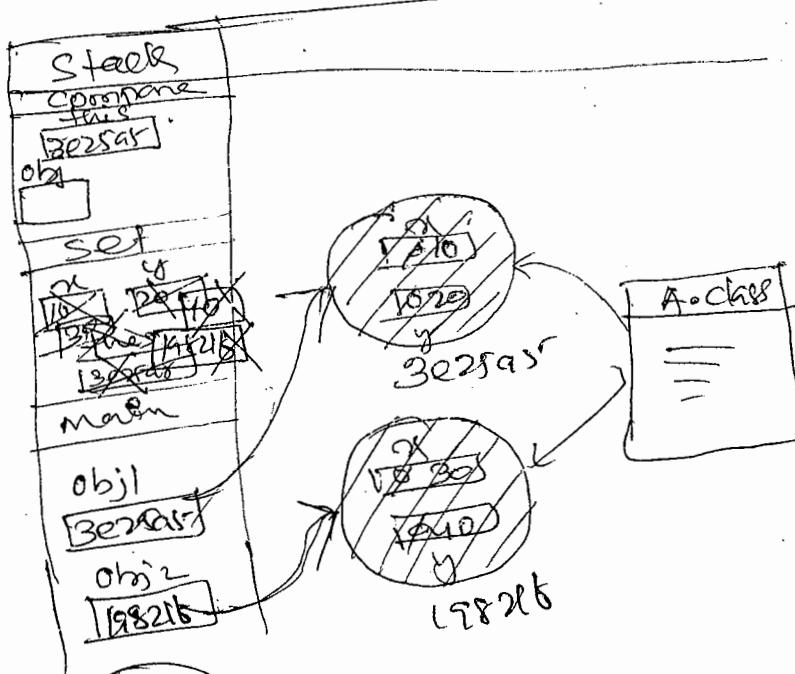
we need not

```

class RefDemo2
{
    public static void main( String args[])
    {
        A obj1 = new A();
        A obj2 = new A();
        obj1.set(10, 20);
        obj2.set(30, 40);
        obj1.compare(obj2);
    }
}

```

RAM



void compare(A obj)

→ obj is a reference variable as it declares  
with class name. It always stores the hashCode.

example:-

```
class Student
```

```

    {
        private int rno;
        private String name;
        void setStudent( int rno, String name)
        {
            this.rno = rno;
            this.name = name;
        }
    }

```

~~void~~  
String getStudent()  
{  
    return " " + name;  
}

### Class Marks

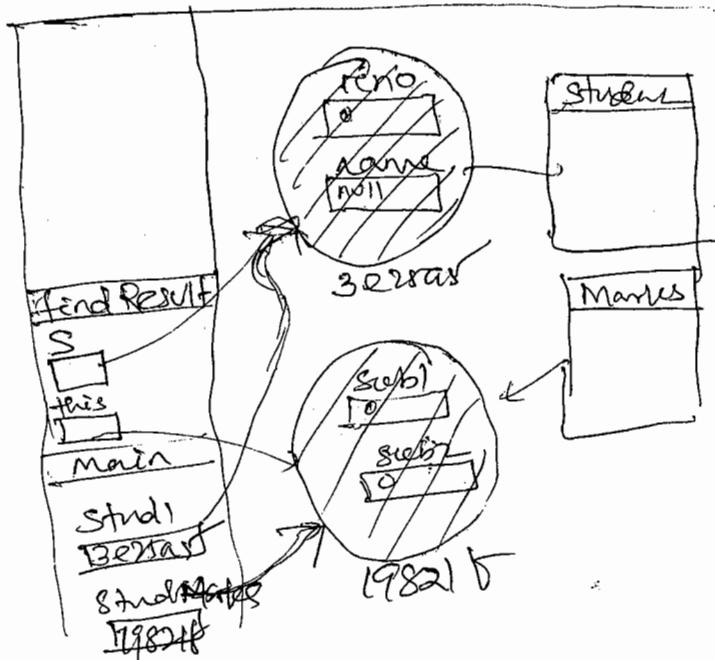
{ private int sub1, sub2, ~~sub3~~;  
void SetMarks (int sub1, sub2)  
{ this.sub1 = sub1;  
    this.sub2 = sub2;  
}

### Void findResult (Student S)

{  
    S.O.P (S.no); // invalid <sup>becoz 'no' is  
                                private cannot access  
                                outside the class.</sup>  
    S.O.P (S.getStudent());  
    if(sub1 < 40 || sub2 < 40)  
        S.O.P ("fail");  
    else  
        S.O.P ("pass");  
}

### Class RefDemo3

{ public static void main (String args[] )  
{  
    Student stud1 = new Student();  
    Marks stud1Marks = new Marks();  
    stud1.setStudent (101, "Rama");  
~~stud1.Marks~~  
    stud1Marks.setMarks (40, 30);  
    stud1Marks.findResult (stud1);  
}



## Constructors :-

- Block of code which has to be executed on creation of object is defined inside constructor.
- Constructor is a special method executed on creation of object.
- Constructor is a initializer method, which defines initial state of an object/initial resources of object.
- An object is never created without invoking constructor.
- Automatic initialization of object is done using constructor.

## Constructors :

### Types of Constructors :-

Java provides 2 types of Constructors.

1. Default Constructor

2. Parameterized Constructor.

→ Default constructors are of 2 types —

- a. Compiler defined
- b. User defined.

Date - 5/10/12

### Properties of Constructor:

→ Constructor name is same as class name.

→ It can be declared as a private, default, protected or public.

→ It is invoked on creation of object.

→ It cannot be called explicitly.

→ It does not have any return type not even void.

→ It doesn't use any modifiers (static, final, abstract etc.)

→ Its operation is Initialization.

→ What is the diff. b/w a method and constructor?

### Method

→ Method name may be same name as a class name or any name.

→ It having return type.

→ It allows access specifiers as well as modifiers.

→ It is called explicitly.

→ It is called any number of times.

→ Operation performed by method are Setters & Getters op's.

→ Method can be called with name.

### Constructor

→ Constructor name must be same name as class name.

→ Doesn't have return type.

→ It allows only access specifiers but doesn't have access modifiers.

→ It is called implicitly at the time of creating object.

→ It is called only once on object.

→ Here the operation is initialized.

→ Constructor never called with name.

## Method

## Constructor

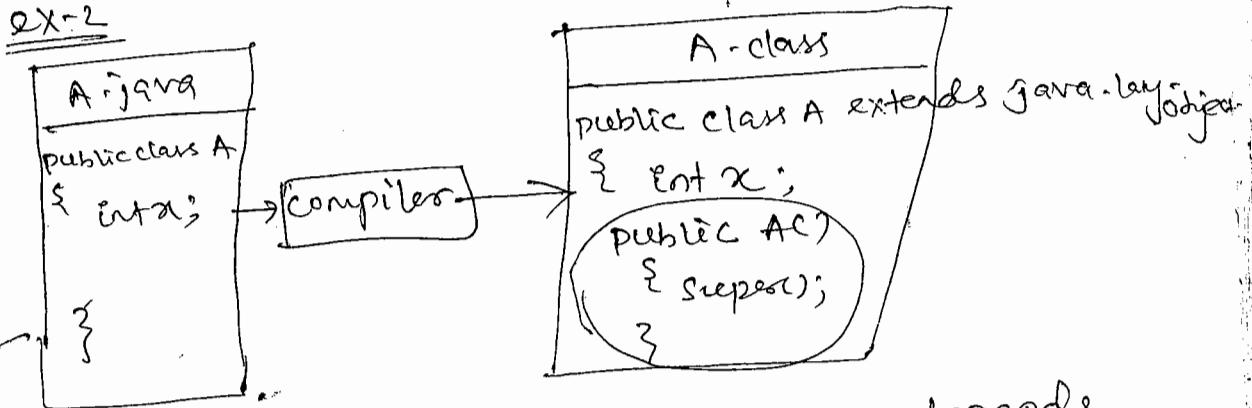
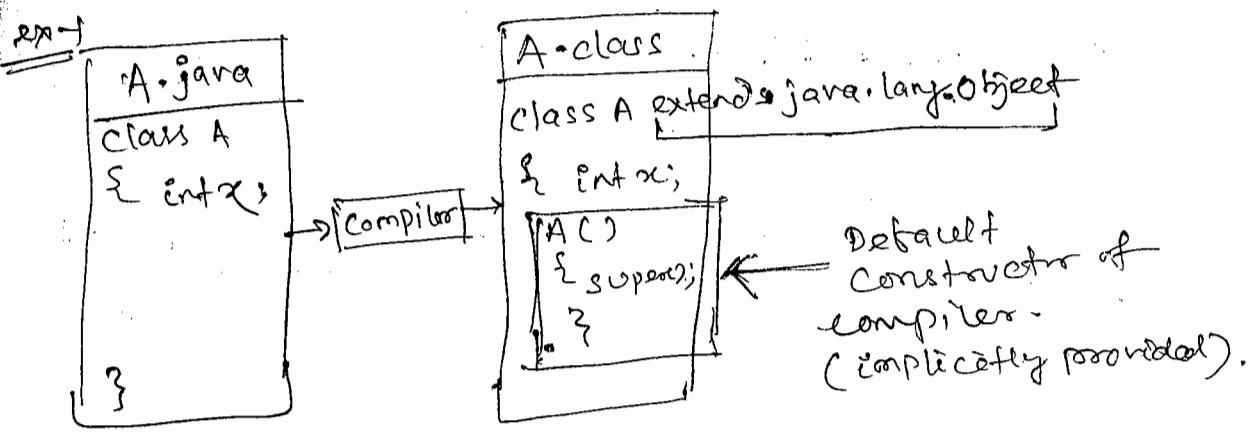
- Methods are inherited. → Constructors are not inherited.
- Constructor doesn't create object. but when the object created it invokes constructor whose duty is initialization.

### Default Constructor :-

- ~~~~~ constructor of class is called
- Non-parameterized constructor or class is called "default constructor".
  - (OR) → A constructor which does not have any parameters is called default constructor.
  - It is divided into 2 types -
  - (a) Be written default const.
  - (a) Compiler written default constructor.
  - (b) User-defined default constructor.

### Compiler-defined default Constructor :-

- ~~~~~
- If there is no constructor defined within the class, compiler provides a constructor without any parameters, which is called default constructor.
  - This constructor calls the constructor of superclass.
  - \* For all classes of java, root class is object.
  - Every class of java must be inherit properties & behaviors of object class.



→ Constructor access specifier always depends upon the access specifier of class :  
(correct as shown in above examples)

I/P → `javac A` ←  
Compiled form "A.java"  
class A extends java.lang.Object  
{ int x;  
A();  
}

JAVAC P. :-  
→ It means Java class compiler.

Syntax :- `javac class-name`

## User-defined default Constructor's

→ A constructor is written by user without any parameters is called userdefined default constructor.

Ex

Class A

```
{ private int x;
  private int y;
```

A() { }

```
{ x=10;
  y=20; }
```

String getXY()

```
{ return x+" "+y; }
```

}

Class ConstDemo

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

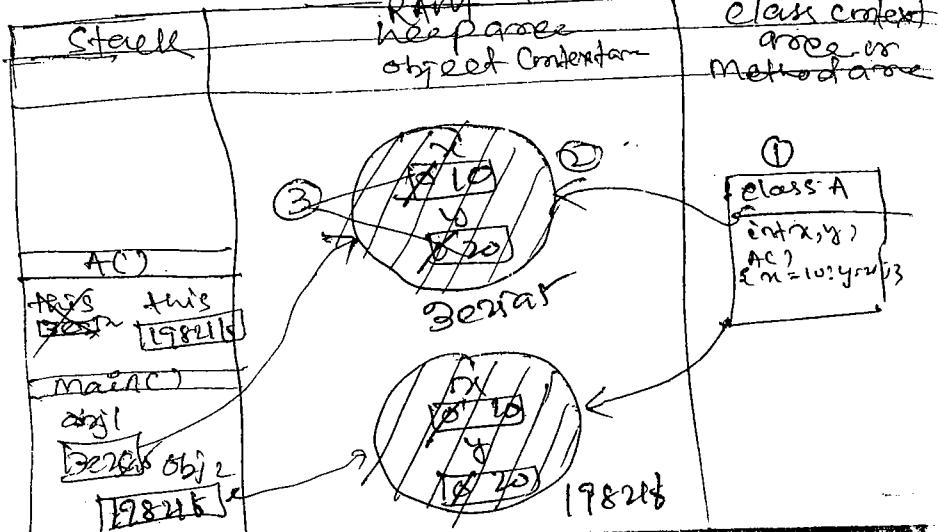
A obj1 = new A();

A obj2 = new A();

S.out( obj1.getXY() );

S.out( obj2.getXY() );

}



- Inside the RAM, Runtime Data area is present, which contains stack & heap area.
- JVM reserves memory by using RAM.
- Constructor is called along with class therefore constructor name is same as class name.
- Constructor is a non-static.

A Obj1 = new A();

→ Calling the constructor along with new.

Obj1 = A();

↓  
method

→ Therefore when object created at that time

→ Constructors invoke & initialize the member variables.  
So constructor is called only once on the object

example:

Class Date

```
{ private int dd;
  private int mm;
  private int yy;
```

Date()

```
{ dd=1;
```

```
  mm=1;
```

```
  yy=2000;
```

}

```
void setDate(int dd, int mm, int yy)
{
  this.dd = dd;
  this.mm = mm;
  this.yy = yy;
```

}

void getDate()

```
{ return dd + "/" + mm + "/" + yy;
```

}

}

## Class ConstDemo 2

```
{ public static void main (String args[])
{
    Date date1 = new Date();
    S.o.pn (date1.getDate());
    date1.setDate (5, 10, 2012);
    S.o.pn (date1.getDate());
}
```

example:-

### Class A

```
{ AC)
{ S.o.pn ("Inside Constructor");
}
void AC)
{ S.o.pn ("Inside method");
}
```

### Class ConstDemo3

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

```
{ A obj1 = new AC();
    obj1.AC();
    obj1.AC();
}
```

OP  
} Inside constructor  
} Inside method  
} Inside method.

## Parameterized Constructor :-

- A constructor with parameters is called parameterized constructor.
- This is user-defined constructor.
- This constructor initializes object by receiving values.
- example :-  
Any statement that will be executed must be written inside a method.
- By using constructor we can execute the statement without creating method.

example :-

```
class login
{
    Button b1, b2;
    TextField t1, t2;
    Label l1, l2;
    Login()
    {
        b1 = new Button();
        b2 = new Button();
        l1 = new Label();
    }
}
```

→ If we want to change the state of an object long at the time of creation taking user input then we must declare parameterized constructor.

- These parameters can be —
- 1. Primitive
- 2. Reference type

## Class Account

```
{ private int accno;
  private String name;
  private float balance;
  Account (int accno, String name, float balance)
```

```
{   this.accno = accno;
    this.name = name;
    this.balance = balance;
```

```
} void deposit (float amt)
```

```
{   balance = balance + amt;
```

```
} void withdraw (float amt)
```

```
{   if (amt > balance)
```

```
     S.o.pn ("Insufficient balance");
```

```
   else
```

```
     balance = balance - amt;
```

```
}
```

```
String getAccount()
```

```
{
```

```
  return accno + " " + name + " " + balance;
```

```
}
```

## Class Bank

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

```
  Account acc1 = new Account ();
```

```
  Account acc1 = new Account (101, "Postam", 5000f);
```

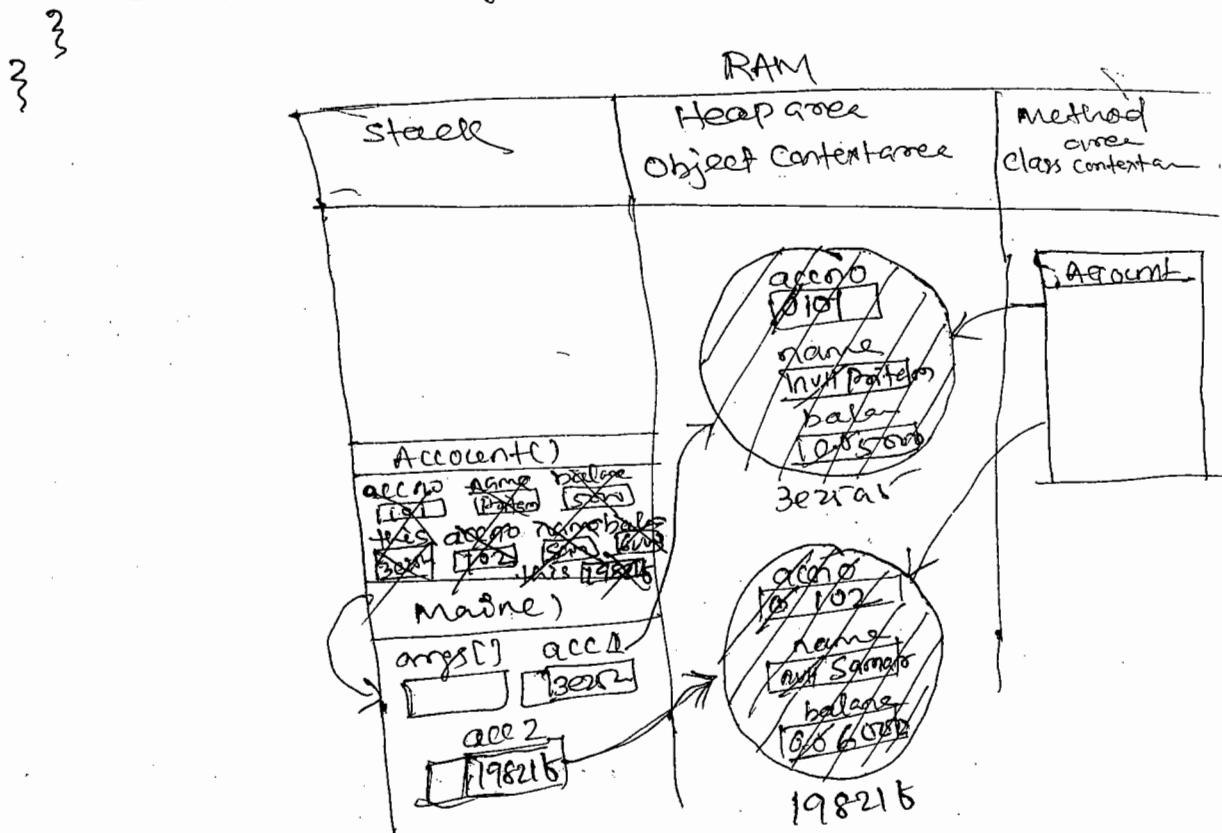
```
  Account acc2 = new Account (102, "Bamra", 6000f);
```

```
  S.o.pn (acc1.getAccount());
```

```
  S.o.pn (acc2.getAccount());
```

```
  acc1.deposit (1000f);
```

acc2.withdraw(500f);  
 S.o.pen(acc1.getAccout());  
 S.o.pen(acc2.getAccout());



→ Once the constructor invoke and executed then memory reserved for constructor is deallocated & return to main().

### Constructor Overloading

→ Defining more than one constructor within class by changing • no. of parameters,  
 • type of parameters  
 • order of parameters

is called Constructor Overloading.

→ Constructor is overloaded in order to extend the functionality of existing constructors.

→ Overloaded Constructors having polymorphic behaviour.

examples :-

example :-

class Employee

```
{ private int empno;
    private String name;
    private float salary;
    Employee(int empno, String name)
```

```
{ this.empno = empno;
```

```
    this.name = name;
}
```

```
Employee(int empno, String name, float salary)
```

```
{ this.empno = empno;
```

```
    this.name = name;
```

```
    this.salary = salary;
}
```

```
void setSalary(float salary)
```

```
{ this.salary = salary;
}
```

```
String getEmployee()
```

```
{ return empno + " " + name + " " + salary;
}
```

class Company

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

```
{ Employee e1 = new Employee(101, "abc");
```

```
    Employee e2 = new Employee(102, "XYZ", 5000);
```

```
    System.out.println("e1.getEmployee());
```

```
    System.out.println("e2.getEmployee());
```

```
}
```

→ We can call a constructor inside another constructor to avoid redundancy.

Date - 8/10/12

"this" constructor call:

→ Calling a constructor of a class within another constructor of same class is done using "this()" constructor call.

→ It allows to include functionality of one constructor inside another constructor in order to avoid redundancy.

Rules :-

→ It must be a first statement within constructor.

→ It should not be recursive.

→ It called only once within the constructor.

→ It is used only inside constructor but not inside method.

Q) What's the diff. b/w "this" & "this()"?

this

this()

→ It hold hash code of current object.

→ It refers to constructor of same class.

Ex :-

Class A

{

A()

{ S.O.P("Inside Default constructor"); }

}

A(int a)

{ this(); }

S.O.P("Inside parameterized constructor");

}

Ans;

Class ConstDemo6

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

{ A obj1 = new A(10);

} } Op :- Inside default Const  
Inside Parameterized Const.

Ex-2 :

Class A

{ A ()  
  { S.O.P("Inside default") } \* By using this() we  
  } can call the default  
A (int i)  
  { A ();      this();      \* otherwise  
  }      { A ();      error  
  }      S.O.P("Inside Param. Const");  
void A ();      this();  
  { S.O.P("Inside method");  
  }

Class ConstDemo7

{ PSUM (String args[])

{ A obj = new A(10);

} }

Op :- Inside default const method 2  
Inside default const 1  
Inside param. const 3

Ex-3 :-

Class A

{ A ()  
  { S.O.P("Inside default const");

}

A (int x)

{

this();

A();

S.O.P("Inside param const.");

}

```

void AC()
{
    S.O.P ("Inside method");
}

class ConstDemo8
{
    psvm (String args[])
    {
        Aobj1 = new AC();
    }
}

```

O/P :- Inside default const.  
 Inside method.  
 Inside param. const.

Ex-4:-

```

class Account
{
    private int accno;
    private String cname;
    private float balance;
    Account (int accno, String cname)
    {
        this.accno = accno;
        this.cname = cname;
    }
    Account (int accno, String cname, float balance)
    {
        this(accno, cname);
        this.balance = balance;
    }
    String getAccount()
    {
        return accno + " " + cname + " " + balance;
    }
}

```

class ConstDemo9

```

psvm (String args[])
{
    Aobj1 = new Account (101, "abc");
    Account acc1 = new Account ();
    // Account acc2 = new Account (102, "xyz", "sowd");
}

```

Account acc3 = new Account(102, "XYZ", 5000);  
S.O.P(acc1.getAccount());  
S.O.P(acc2.getAccount());

}  
O/P      101 abc 0.0  
              102 XYZ 5000.

Ex-5 :- (Rule-1)

Class A

{ AC)

{ S.O.P("inside default const");

}

A(int n)

{ S.O.P("inside parameterized const");

{ this(); //encore.

}

→ The above program display compile time error, becoz called to this must be first statement within constructor.

Ex-6 :- (Rule-2)

Class A

{ AC)

{ this();

{ S.O.P("Default Const");

}

→ The above program display compile time error becoz a constructor cannot be recursive.

→ As the constructor is initialize ~~the once~~ so constructor cannot be ~~initialize~~ more than one. (but method allows recursive).  
The above example shows direct recursion.

Ex-7 :-

Class A

{ A( )

{ this(10);  
S.O.P ("Inside default");

}  
A( int x)

{ this();  
S.O.P ("Inside parameterized");

}

- The above program display compile time error  
becoz recursive constructor call is not allowed.
- It shows indirect recursive.

Ex-8 :- (Rule-3)

Class A

{ A( )

{ S.O.P ("Inside default");

}  
A( int x)

{ S.O.P ("Inside one para.");

}  
A( int x, int y)

{ this();

this(10); // error

S.O.P ("Inside Two para.");

}

- The above program display compile time error, becoz  
call to this must be first statement.

### Ex-109 :- (Rule-4)

Class A

{ A() {

{ S.O.P("Inside default Const"); }

Void m1()

{ this(); }

S.O.P("Inside method"); }

} }

→ The above program displays compilation error, becz call to this() must be first statement inside constructor, but not inside method.

### Ex-10

Class A

{ A()

{ S.O.P("abc"); }

S.O.P("XYZ"); }

}

A(<sup>int</sup>x)

{ S.O.P("abc"); }

S.O.P("PQR"); }

}

A(<sup>int</sup>x, <sup>int</sup>y)

{ S.O.P("abc"); }

S.O.P("MNO"); }

}

}

→ Non static block is used to initialize the object.

→ The block having no name

→ If the code is common for all the constructors, then that

block must code must be present in a non-static block.

A rectangular box with curly braces at the top and bottom, containing the text "SOP('abc')".

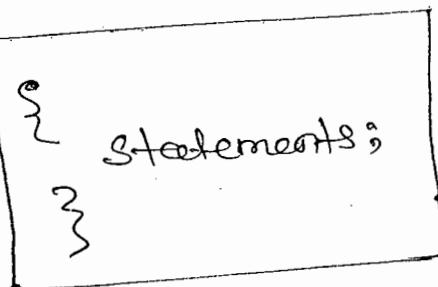
Non static block.

~~get executed~~

## Non static block :

- A block of code which has to be executed on creation of object & before invoking constructor included inside this block.
- An operation which is common for all constructor is performed using non static block.
- It can be called any no. of type inside the class.

## Syntax :



Date - 9/10/12

- This block get executed on creation of object before invoking constructor.

Ex:-

Class A

{

{ S.o.p("Inside non static block");

}

A()

{ S.o.p("Inside default Constructor");

}

A(int x)

{ S.o.p("Inside parameterized constre.");

}

Class ConstDemo15

{ public static void main (String args[])

{ Aobj1 = new A();

Aobj2 = new A(10);

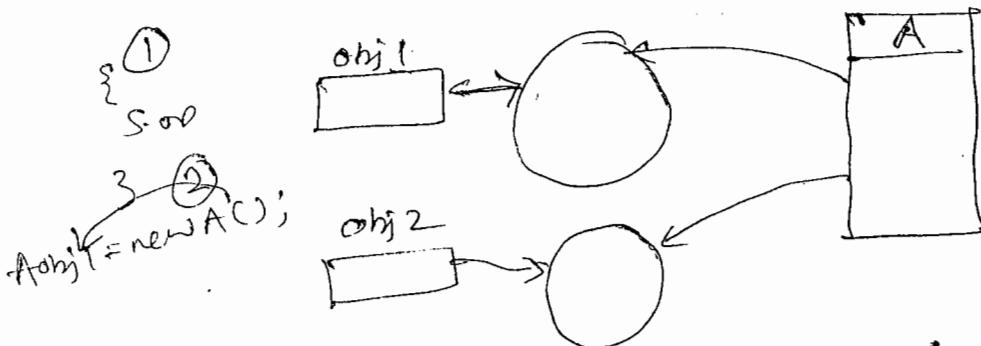
{ }

{ }

that  
ok.

O.P :-

- Inside non static block
- Inside default Constructor.
- Inside non static block
- Inside parameterized constructor.



→ It is an anonymous block. & this block has no name.

→ The class have only no. of non-static block.  
→ The order of their execution is that the order in which they appear.

Ex - 2

Class A

```

{   {
    S.O.P("Inside non-static block-1");
}
A()
{
    S.O.P("Inside default Const");
}
{
    S.O.P("Non-static block 2");
}
A(int x)
{
    S.O.P("Inside para. const");
}
}
  
```

## Class Const Demo 16

```
{ public static void main (String args[])
{
    A obj1 = new A();
    A obj2 = new A(10);
}
```

- O/P :- Inside non-static block - 1  
Non-static block - 2  
Inside default const.  
Inside non-static block - 2  
Non-static block - 2  
Inside para const.

→ Order of executing non-static blocks are the order in which they appear / are defined within the class.

## Static block :-

### Class System

```
{ static PointStream out;
    {
        out = new PointStream();
    }
}
```

- A block of code which has to be executed on loading class included within static block.  
→ It is used for static init or class init.  
→ It can access only static members of a class but cannot access non static members.

## Syntax :-

```
static
{
    statements;
}
```

Note :-

→ Inside the class we can define the following things -

1. Variables

- non static variable
- static variable

2. methods

- non static method
- static method

3. Constructors.

4. blocks

- static block
- non-static block

5. classes.

Ex:-

Class A

{

    int x;

    static int y;

    {  
        System.out.println("Inside non-static block");  
    }

}

    static

    {  
        System.out.println("Inside static block");  
    }

}

    A()

    {  
        System.out.println("Inside Constructor");  
    }

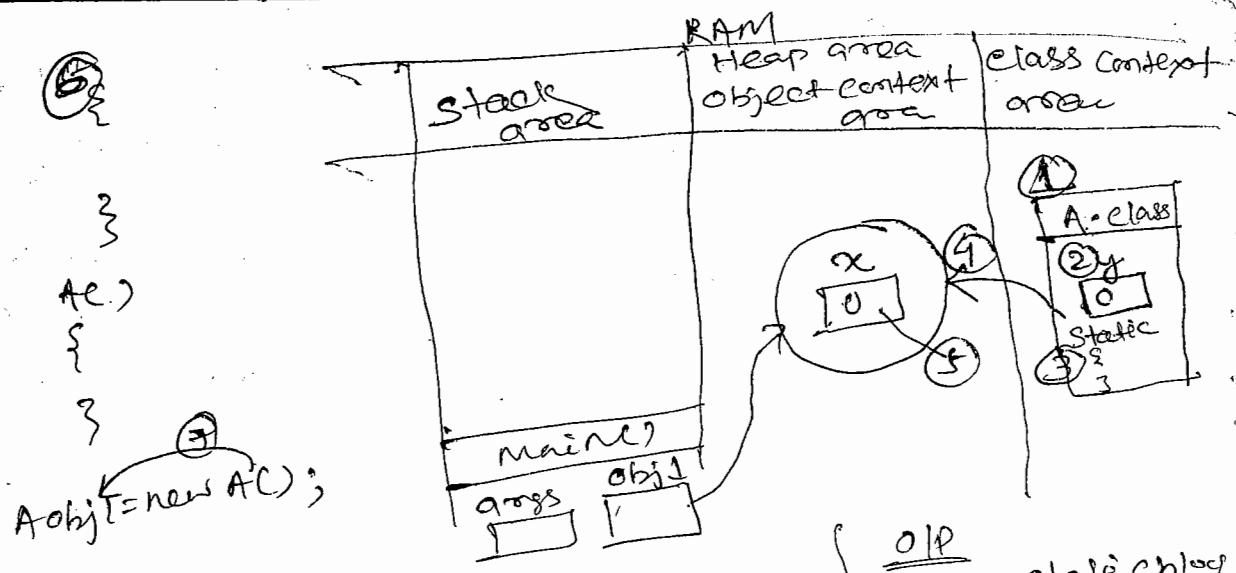
}

    Class ConstDemo

{  
    public static void main (String args[])

    {  
        A obj1 = new A();  
    }

}



- ① → Class is loaded.  
 ② → Allocate memory for static variable  
 ③ → execute static block  
 ④ → Creating object  
 ⑤ → Initialize ~~non~~ instance variable.  
 ⑥ → execute non-static block  
 ⑦ → execute constructor.
- When class is loaded let ①, ② & ③ are happen.  
 but when object is created all are execute sequentially.  
 → sequence may change in inheritance.  
 → sequence may change in static & non-static block.

Q) what is the diff. bet? static & non-static block

Non-static block

- static block
- It is use for class initialization.
  - It access only static members → It access static & non-static members of a class.
  - It get executed on loading → It get executed on creation of object & before invoking constructor.
  - Executed only once.
  - It is without this.
  - It is use for object initialization.
  - It access static & non-static members of a class.
  - Executed whenever object of a class is created.
  - It is with this.

- Q) Can you execute a class without main?
- Yes, but it is done by writing everything including main() in a static block.
- But it is not giving any error except Java 6 after this it was provide error becoz if Java 6 it first checks main() method exist or not then execute.

Ex :-

Class ConstDemo 18

{

static

{ S.O.P("inside static block");

}

O/P :- inside static method

Exception in thread main()

Ex

Class ConstDemo 18

{

static

{ S.O.P("inside static block");

System.exit(0); // terminating execution of program.

}

O/P inside static block.

Ex :-

Class Pointee

{ void point(String s)

{

S.O.P(s);

}

class Computer

{ static Pointers P ;

static

{ p = new Pointer();

}

}

class Word

{ public static void main (String args[])

{ Computer . P . print ("Hello");

Computer . P . print (" Java");

}

3

obj Hello

Java

RAM

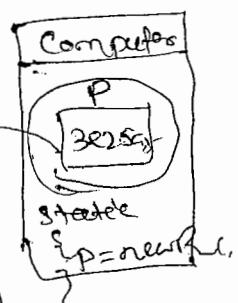
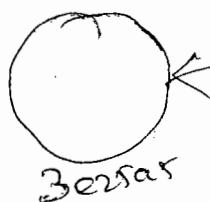
Computer . P . print  
("Hello")

void print (S)

S

3

Computer . P . print (" Java");



→ A constructor within a class can be private,  
protected, public & default (called access specifiers).

## Private Constructor :-

- Constructors of a class can be declared as private.
- Private constructor cannot access outside the class.
- Private constructor is declared in order to restrict a class from creating object.
- Programmer is not allowed to create object directly, it allows to create object indirectly with the help of methods.
- We restrict the class from creating object bcoz to restrict the programmer from creating no. of objects which avoids wastage of memory.
- If method having referent type of class then it returns object.

Ex

RentOne

RentOne r = new RentOne() X

RentOne r = <sup>RentOne</sup>Factory.getRentOne();

Q) Develop a class which allows to create one object.

Q) What is Factory method?

→ Factory method having following properties —

1. static

2. return type is class.

→ A method which creates an object and return hashcode of that object is called factory method.

→ Its method having best referent type as class, then it returns hashcode of object.

### Ex-1: Class A

```
{
    private int x;
    private A();
}
S.O.P("inside default const");
```

### Class ConstDemo20

```
{
    public static void main (String args[])
{
    A obj1 = new A();
}
```

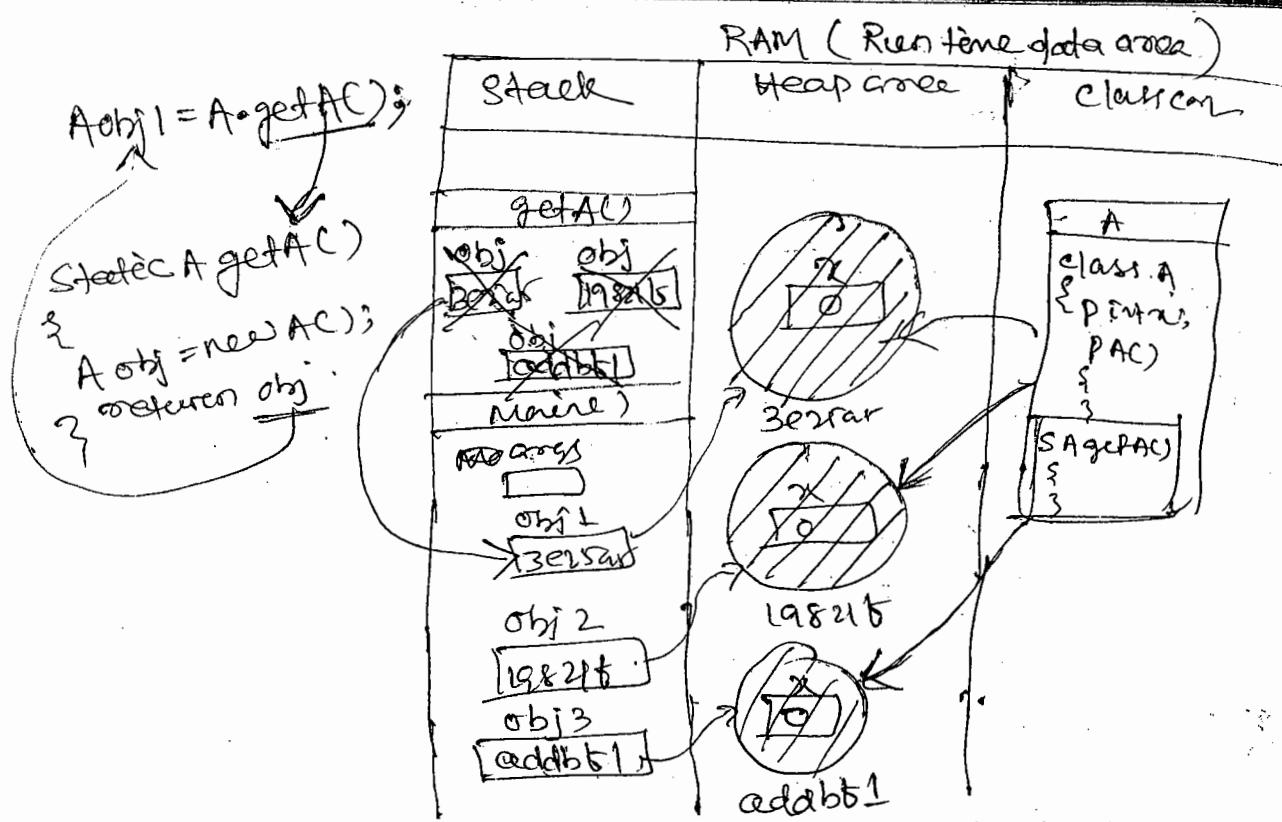
\*→ The above program display compilation error  
constructor of class A is private, which cannot access  
outside the class.

### Ex-2 Class A

```
{
    private int x;
    private A();
}
S.O.P("inside default const");
{
    static A getA()
{
    A obj = new A();
    return obj;
}
```

### Class ConstDemo21

```
{
    public static void main (String args[])
{
    A obj1 = A.getA();
    A obj2 = A.getA();
    A obj3 = A.getA();
    S.O.P (obj1);
    S.O.P (obj2);
    S.O.P (obj3);
}
```



### Singleton Class

→ A class which is having following properties is called Singleton class.

1. private constructor.
2. final class (not inherited).
3. Allows to create only one copy of object.
4. provide factory method to create object.

→ A class which allows to create only one copy of object is called Singleton class.

ex:-

Class A

```
{ private int x;
private static A obj;
```

static

```
{
    obj = new AC();
}
```

static A getAC()

```
{
    return obj;
}
```

## Class ConstDemo22

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

```
{ // A obj1 = new A(); // exercise,
```

```
A obj1 = new A().getAC();
```

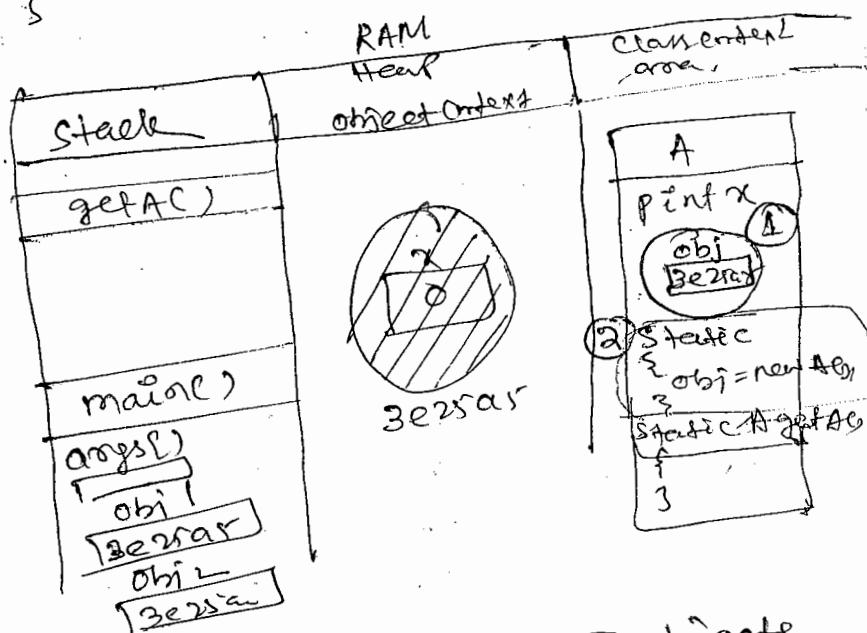
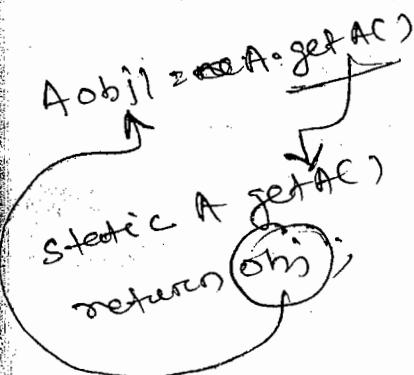
```
A obj2 = A().getAC();
```

```
S.o.p (obj1);
```

```
S.o.p (obj2);
```

}

Obj1 3e28a5 } refers same hash code.  
3e28a5 }



Q) Develop a class which allows to create 5 objects or instances.

```
Class Product
```

```
{ private static int count;
```

```
private String name;
```

```
private float price;
```

```
private product()
```

```
{
```

```
    }
```

```
    static Product getProduct()
```

```
{
```

```
    Product p = null;
```

```
    }
```

if (coent < 5)

{  
    p = new Product();

    coent = coent + 1;

}  
    reteeren p;

}

Class ConstDemo23

{  
public static void main (String args[])

{  
    Product prod1 = Product.getProdut();

    Product prod2 = Product.getProdut();

    Product prod3 = Product.getProdut();

    Product prod4 = Product.getProdut();

    Product prod5 = Product.getProdut();

    Product prod6 = Product.getProdut();

    S.O.P (prod1);

    S.O.P (prod2);

    S.O.P (prod3);

    S.O.P (prod4);

    S.O.P (prod5);

    S.O.P (prod6);

}

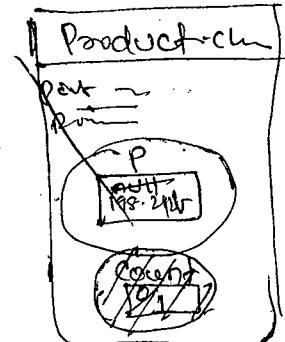
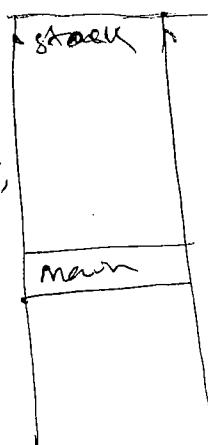
O/P :- 1982.6

add b61

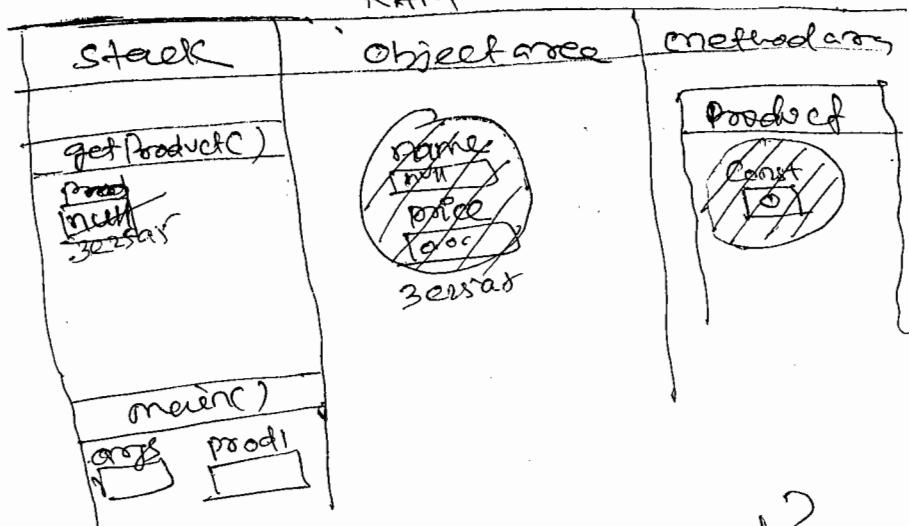
422816

930461

Point prod1 -> Product.getProdut();



11/10/12



- Q) What is a Singleton class? Where is it used?
- Singleton is a design pattern meant to provide only one instance of an object. Other objects can get a reference to this instance through a static method (class constructor is kept private).
  - Restricting the no. of instances may be necessary or desirable for technological or business reasons.

for example, we may only want a single instance of a pool of database connections.

- Constructor with parameters of type reference:
- A constructor having parameters of type reference receive hashcode of object or object.
  - An object can be initialized with resources of another object.

Ex: Class A

```
{ void m1()
{
    S.O.P ("Inside m1 of class A");
}
```

Class B

{ private A obj;  
B(A obj)

{ this.obj = obj;

} void m1()

{ obj.m1();

S.O.P("Inside m1() of class B");

}

Class ConstDemo2

{ public static void main (String args[])

{ A obj1 = new A();

B obj2 = new B(obj1);

obj2.m1();

} A obj3 = new A();

A obj2 = new A();

→ A obj1 = new A();

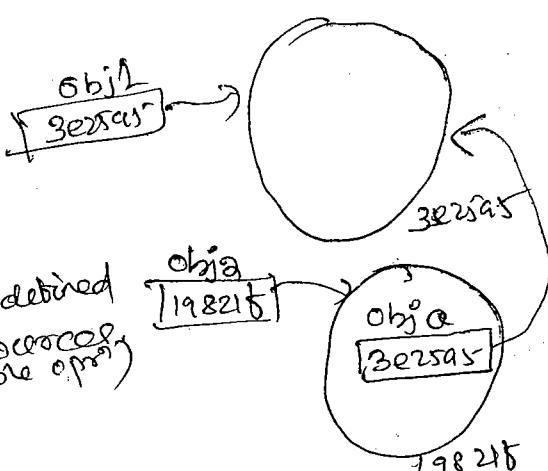
→ B obj2 = new B(obj1);

Object may be defined  
object may be defined  
dynamic resources  
dynamic variable opn

→ B obj2 = new B(obj1);

B(A obj)

OP:- Inside m1 of class A  
Inside m2 of class B.



ex :-

class A

{ int x = 10 ;

}

class B

{ int p = 20 ;

A obj ;

B(A obj)

{ this . obj = obj ;

}

void seem()

{ int s = p + obj . x ;

s . o . p (" seem is " + s) ;

}

class ConstDemo2

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

{ A obj1 = new AC () ;

obj1 . x = 50 ;

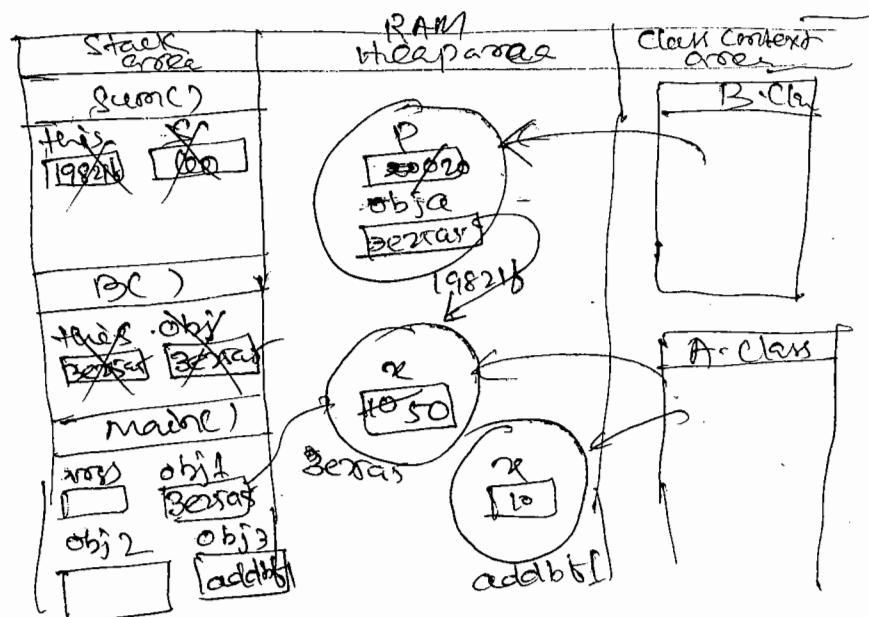
A obj3 = new AC () ;

B obj2 = new B (obj1) ;

obj2 . seem () ;

}

}



Ex:-

Class Acethore

{ private String cname;  
void setAcname (String cname)

{ this.cname = cname;

}  
String getName ()

{ return cname;

}

Class Book

{ private String bname;

private Acethore a;

Book (Acethore a)

{ this.a = a;

}  
void setBname (String bname)

{ this.bname = bname;

}  
void printBook (C)

{ S.O.P ("Book name " + bname);  
S.O.P ("Author name " + a.getName());

}

Class ConstDemo2

{ public static void main (String args [])

{ Acethore author1 = new Acethore ();

Acethore author2 = new Acethore ();

author1.setName ("James");

author2.setName ("Scott");

```

Book book1 = new Book(author1);
Book book2 = new Book(author1);
Book book3 = new Book(author2);
book1.setBname("Java");
book2.setBname("Complete Java");
book3.setBname("Oracle");
book1.printBook();
book2.printBook();
book3.printBook(); // author1.setFname("Gawlik")
book1.printBook();
    
```

O/P

Bookname Java

Authorname James

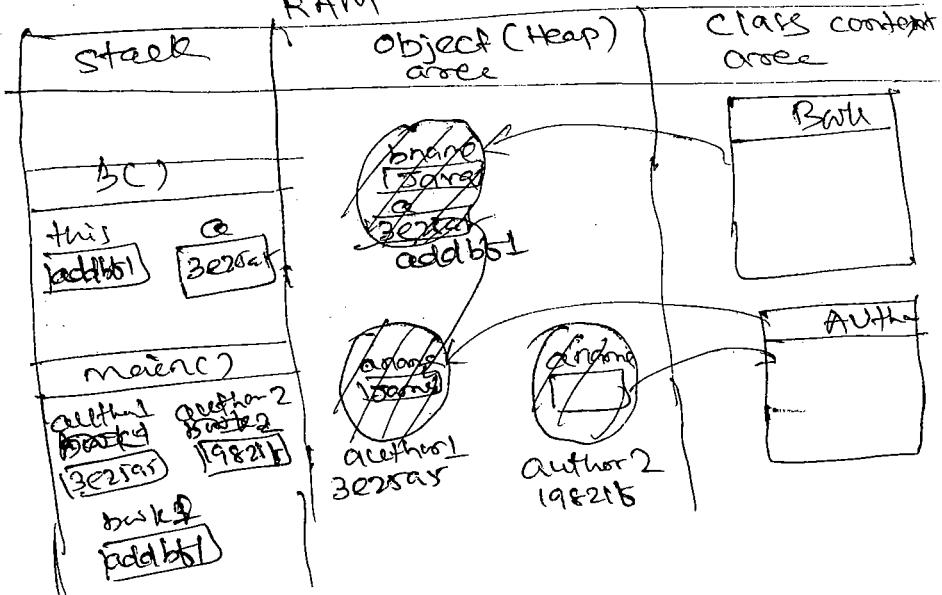
B-n → Complete Java

A-n → James

B-n → Oracle

A-n → Scott

RAM



# INHERITANCE

12/10/12

3:

Q) What is Inheritance?

- Inheritance is the concept of a child class (subclass) automatically inheriting the variables & methods defined in its parents class (super class).
- It's a primary feature of object oriented programming along with encapsulation & polymorphism.
- It's a process of creating group of classes which shares common properties and behavior.
- Inheritance relationship is called "is-a" relation.
- is-a relationship occurs in between selected classes.

Advantages of inheritance:

- (1) Benefits of inheritance in OOP : Reusability
- Once a behaviour (method) is define in a super class that behaviour is automatically inherited by all subclasses. Thus we write a method only once and it can be used by all subclasses.
  - Once a set of properties (fields) are defined in a super class, the same set of properties are inherited by all sub classes.
  - A class and its children share common sets of processes.
  - A subclass ~~must~~ only need to implement the difference between itself and the parent.

Allows to use the content of one class inside another class without creating object.

### (2) Generalization:

- It is a process of identifying common features (properties, behaviors) of more than one object.
- These features are defined once and used more than one time (Reusability).
- A superclass is called generalized class bcoz class is design based on commonality.

### (3) Specialization:

- Adding new features into an existing class without modifying it (extensibility) is called specialization.

Need of decomposition ① Redundancy  
② Inconsistency

### Extends Keyword:

- A class is inherited by another class using extends keyword.
- A class extends only one class, but cannot access extends more than one class.

### Syntax:

```
class subclass-name extends super-class-name
{
    fields;
    methods;
}
```

Ex:- Class A

{  
}  
Class B extends A { }

A → Superclass  
B → Subclass

ex-2

Class A

Class B

Java Inheritance part

3                           3  
 Class C extends A, B //error

→ A class which is inherited by other class is called Superclass.

→ A class which inherits superclass is called Subclass.

What you can do in a subclass regarding fields:-

→ The inherited fields can be used directly, just like any other fields.

→ You can declare new fields in a subclass that are not in the superclass.

→ You can declare a field in a subclass that are not in the superclass with the same name as the one in the superclass, thus hiding it (not recommended).

→ A subclass does not inherit the private members of its parent class. However if the super class is public or protected methods to access its private fields, these can also be used by the subclass.

Composition .

Ex:-

Class A

{ int x;

}

Class B

{ A ob = new A();

void m1()

{ S.O.P (ob.x);

}

composition

Class B extends A

{ void m1()

{ S.O.P (x);

}

ex-2: Class A      Class B extends A

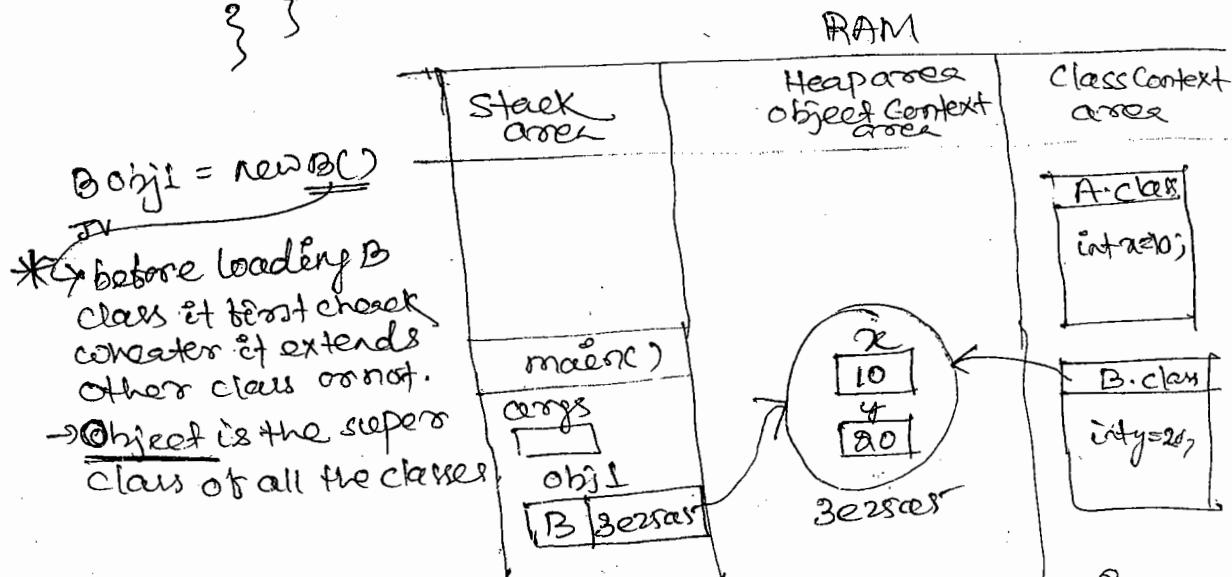
```

    { int x=10; }           { int y=20; }
    { }                     { }
  }
```

Class InheritDemo1

```

    { public static void main (String args[])
      {
        B obj1 = new B();
        S.o.p (obj1.x);
        S.o.p (obj1.y);
      }
  }
```



- On creation of subclass object memory is allocated for instance variables of superclass & subclass.
- JVM before loading subclass, it loads first Super class.
- Hierarchy of loading the class is top to bottom.

ex-3: Class A      Class B extends A

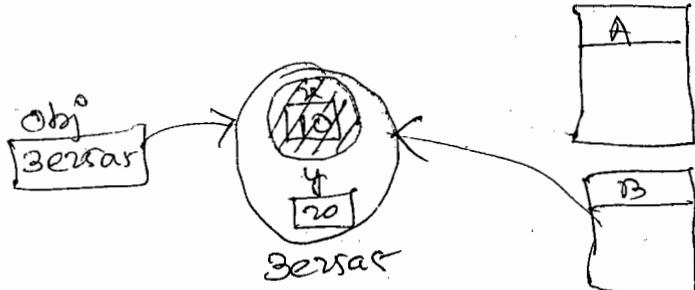
```

    { private int x=10; }   { int y=20; }
    { }                     { }
  }
```

Class InheritDemo2

```

    { public static void main (String args[])
      {
        B obj = new B();
        S.o.p (obj.x);       ← error
        S.o.p (obj.y);
      }
  }
```



→ Private members of super class cannot access within the subclass.

Date - 13/10/12

Q-1:

Class A

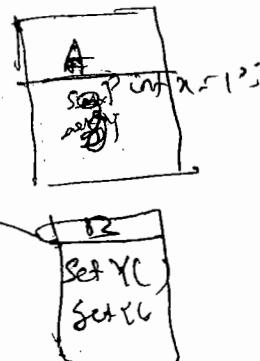
```
{
    private int x;
    void setA(int x)
    {
        this.x = x;
    }
    int getX()
    {
        return x;
    }
}
```

3      3

Class Inherit Dom B

```
{
    public void print()
    {
        B obj1 = new B();
        obj1.setX(10);
        obj1.setY(10);
        System.out.println(obj1.getX());
        System.out.println(obj1.getY());
    }
}
```

Q-2:  
10  
20



## what you can do in a subclass regarding Methods:

- The inherited methods can be used directly as they are.
- You can write a new instance method in the subclass that has the same signature as the one in the superclass thus overriding it.
- You can write new static method in the subclass that has the same signature as the one in the superclass thus hiding it.
- You can declare new methods in the subclass that are not in the superclass.

### in composition

ex-2:

~~class A~~

{ void m1();

{ S.O.P("inside m1");

}

Class Inherit Demo4

{ psvm(string args[])

{ Bobj1 = new B();

obj1.m1();

}

OP

inside m1

inside m2

class B extends A

{ void m2();

{ m1();

S.O.P("inside m2");

}

OP

~~QUESTION~~

## Scanner class:-

- Scanner class provides set of methods to read data from different sources.
- This class is available in java.util package.
- Scanner class provides methods like -
  - nextInt() → read integer value.
  - nextFloat() → read float value.
  - nextDouble() → read double value.
  - next() → read one word length string.
  - nextLine() → read more than one word length string.
  - nextBoolean() → read boolean value.
  - nextLong() → read long value.
- Basically in java all given from keyboard.

## example:-

```

import java.util.*;
class Employee
{
    private int empno;
    private String ename;
    Scanner scan = new Scanner(System.in);
    void read()
    {
        readEmployee();
    }
    void readEmployee()
    {
        s.o.p("Enter empno");
        empno = scan.nextInt();
        s.o.p("Enter ename");
        ename = scan.nextLine();
    }
}
  
```

```
void printEmployee()
```

```
{ S.o.p ("Employee No" + empno);
```

```
    S.o.p ("Employee name:" + ename);
```

```
}
```

Class SalariedEmployee extends Employee

```
{ private float salary;
```

```
void readSalary()
```

```
{ S.o.p (
```

~~Scanner~~

```
    S.o.p ("Input Salary"));
```

```
salary = scan.nextfloat();
```

```
}
```

```
void printSalary()
```

```
{ S.o.p ("Salary" + salary);
```

```
}
```

Class CommEmployee extends Employee

```
{ private float comm;
```

```
void readComm()
```

```
{ S.o.p ("Input Comm");
```

```
comm = scan.nextfloat();
```

```
}
```

```
void printComm()
```

```
{ S.o.p ("Comm" + comm);
```

```
}
```

Class Company

```
{ psvm (string arr[ ])
```

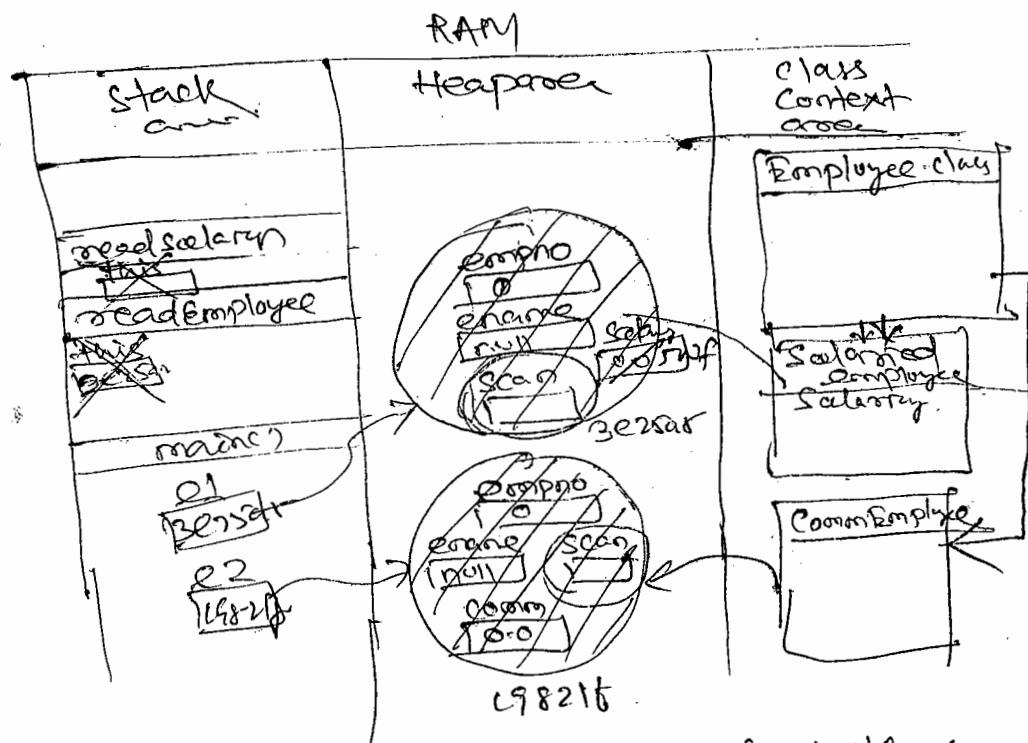
```
{ SalariedEmployee ob =
```

```
    new SalariedEmployee();
```

```

e1.readEmployee();
e1.readSalary();
CommEmployee e2 = new CommEmployee();
e2.readEmployee();
e2.readComm();
e1.printEmployee();
e1.printSalary();
e2.printEmployee();
e2.printComm();
}

```



→ The behaviour of method changes according to object.

## Constructors in Inheritance:

- Default constructor of superclass is called by the constructor available in the subclass.
- Calling of super class default constructor within subclass is done implicitly. (Compiler)
- If no other constructor of superclass is called explicitly.

Q) What is Constructor chaining?

- When a constructor of a class is executed it will automatically call the default constructor of the superclass (if no explicit call to any of the superclass constructor) till the root of the hierarchy.

Ex:  
class A

{ A() }

{ S.O.P ("Default Const of Super Class"); }

}

class InheritDemo6

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

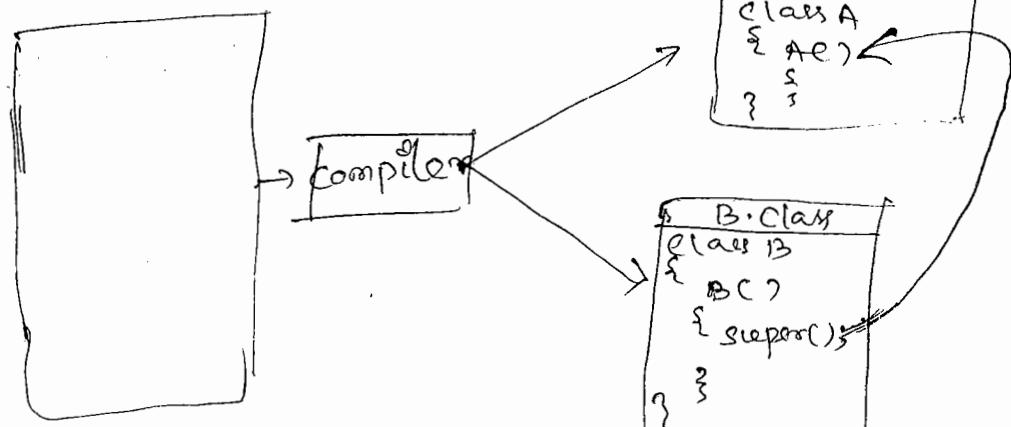
{ B obj = new B(); }

}

}

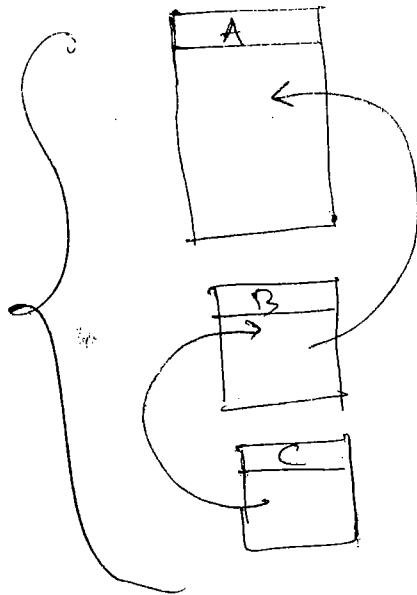
OIP      ↗ Default Const of Super Class  
                ↗ Default Const of Subclass

} Class B extends A  
{ B()  
{ S.O.P ("Default Const of Subclass"); }



Bobj1 =  
new B();

CO2001-ECDU-H02-01-TIA-H2025



Ex-2:-

Class A

```
{ private int x;  
private int y;  
A()  
{ x=10;  
y=20;  
}  
String getXy()  
{ return x+" "+y; }
```

Class InheritDemo

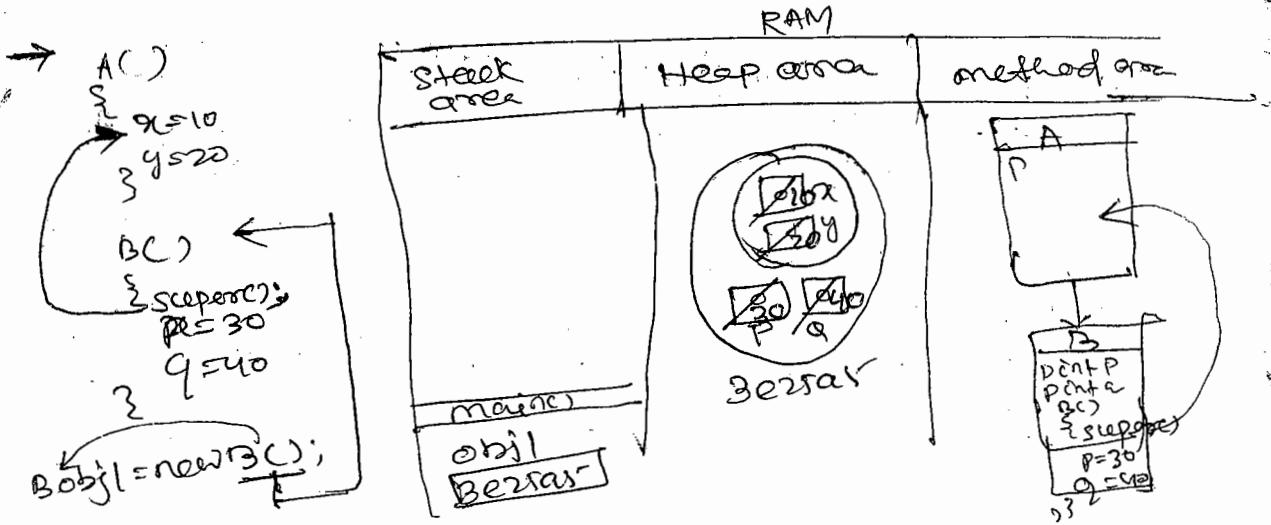
```
{ public static void main(String args[])
```

```
{ B obj1 = new B();  
System.out.println(obj1.getXy());  
System.out.println(obj1.getPQ()); }
```

O/P:- 10 20  
30 40

Class B extends A

```
{ private int P;  
private int Q;  
B()  
{ p=30;  
q=40;  
}  
String getPQ()  
{ return p+" "+q; }
```



ex-3:

Class A

{ A(int x)

{ S.O.P("inside paraclass");

}

Class InheritDemo8

{ public static void main(String args[])

{

    B obj1 = new B();

}

→ The above program display compilation error

becoz there is no default constructor in super class and no constructor of Super class is bind with subclass constructor.

Class is bind with subclass constructor.

→ In order to avoid this error, we must write one of the following things —

① Declare a default constructor

② make a super class call from subclass explicitly.

Class B extends A

{

    B()

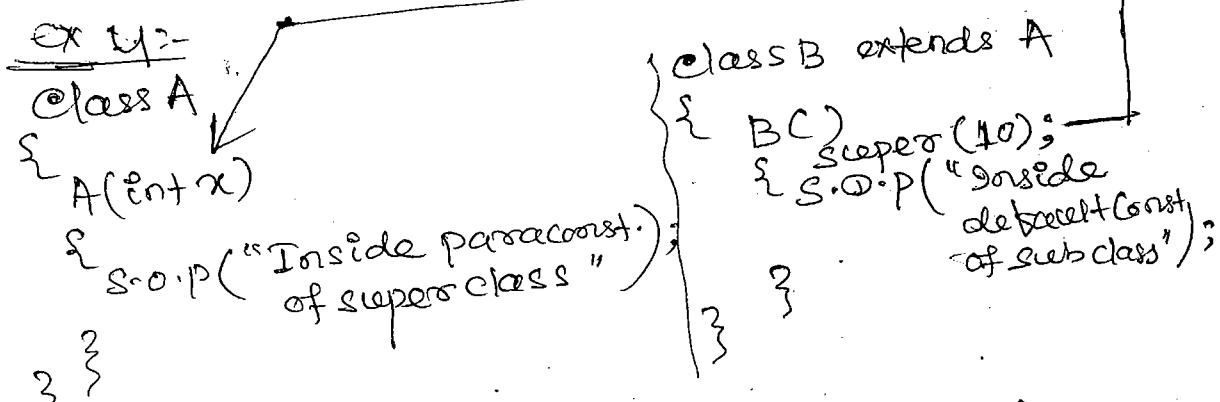
{

    S.O.P("Subclass const");

}

## "Super()" constructor call :-

- "super()" constructor call is used in order to call super class constructor within subclass explicitly.
- This constructor call must be first statement within sub class constructor.
- A constructor of subclass calls only one constructor of super class.
- It is used only inside subclass constructor but not inside method.



Class InheritDemo9

```

public static void main(String args[])
{
    B obj1 = new B();
}

```

O/P  
Inside param const of super class  
Inside default const of sub class.

Ex-5:-

Class A

```

A()
{
    A(int x)
    {
        S.O.P("Param of super class");
    }
}

```

Class B extends A

```

B()
{
    super();
    S.O.P("Param of super class");
    S.O.P("Param of sub class");
}

```

## Class Inheritance Demo 10

```
{ psvm (String args[ ])
```

```
{ Bobj1 = new B();
```

```
    Bobj2 = new B(20);
```

Q.P.:-

para of superclass

defact of subclass

defact of superclass

para of subclass

Date - 16/10/12

Q) what is the diff. bet "this" and "super()"  
constructor calls?

this()

super()

→ Calling the constructor of some class within another Super class within subclass  
constructor of same class.

Ex:-

Class Employee

```
{ private int empno;
```

```
private String ename;
```

```
Employee (int empno, String ename)
```

```
{ this.empno = empno;
```

```
    this.ename = ename;
```

```
}
```

```
int getEmpno()
```

```
{ return empno;
```

```
}
```

```
String getEname()
```

```
{ return ename;
```

```
}
```

}

-of  
"ss");

);

Class SelectedEmployee extends Employee

2 private block sculcerey;

Salaried Employee (int empno, String ename,  
float salary)

Super(empno, ename);

$$\text{thee} \cdot \text{salary} = \text{salary};$$

```
float getSalary()
```

research scarcely ;

## Class Inheritance

{ public static void main ( String args [ ] )

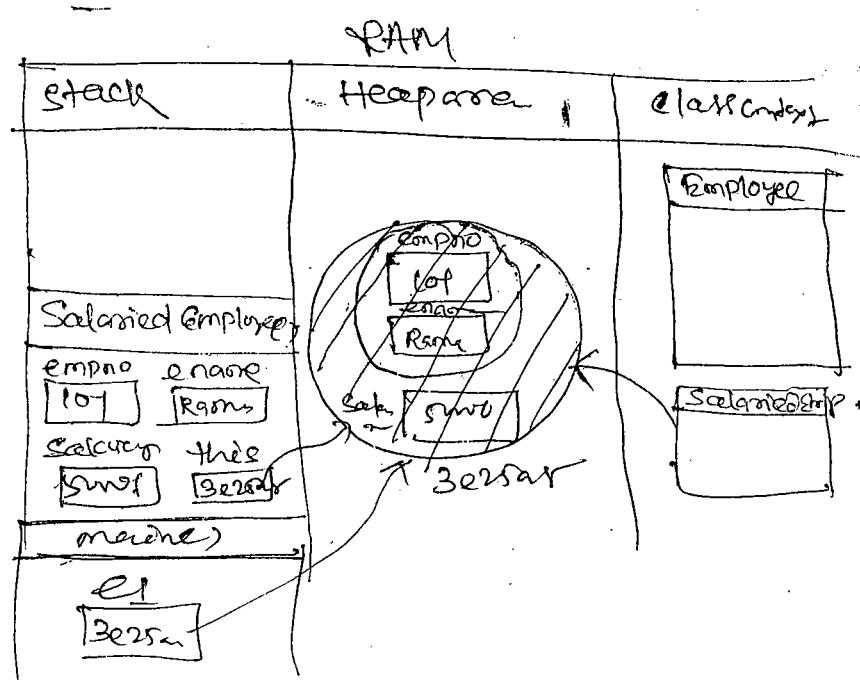
Salaried Employee ~~et~~ = new

Salaried Employee 101, "Rama", 5000/-;

```
S.o.p (es.getEmpno());
```

```
System.out.println(e1.getEname());
```

S.O.P (e1.getSalary());



ex:-  
Class Account

```
{ private int accno;
  private String ename;
  Account()
  {
    accno = 0;
    ename = null;
  }
  Account(int accno, String ename)
  {
    this.accno = accno;
    this.ename = ename;
  }
  int getAccno()
  {
    return accno;
  }
  String getName()
  {
    return ename;
  }
}
```

Class SavingAccount extends Account

```
{ private float balance;
  private boolean draftfac;
  SavingAccount(int accno, String ename,
                float balance, boolean draftfac)
  {
    super(accno, ename);
    this.balance = balance;
    this.draftfac = draftfac;
  }
  float getBalance()
  {
    return balance;
  }
  boolean getDraftfac()
  {
    return draftfac;
  }
}
```

## Class Inherit Demo 12

```
{ public static void main (String args[])
{
    SavingAccount sal = new SavingAccount
        (101, "xyz", 5000f, false);
    SavingAccount sal2 = new SavingAccount
        (102, "abc", 6000f, true);
    S.O.P (sal.getAcno());
    S.O.P (sal.getChname());
    S.O.P (sal.getBalance());
    S.O.P (sal.getDraftacc());
}}
```

\* A constructor can use "this()" & "super()" but not both.

Ex:-

### Class A

```
{ A()
{
    S.O.P ("Inside default constructor of
            super class");
}}
```

### A (int x)

```
{ this();
    S.O.P ("Inside Para Const. of superclass");
}}
```

Class B extends A

### B ( )

```
{ super(10);
    S.O.P ("Inside defaulet const. of subclass");
}}
```

B (int x)

{

    this();

    S.O.P("inside paraConst of subclass");

} ;

Class InheritDemo 23

{ public static void main (String args[])

{

    B obj1 = new B();

    B obj2 = new B(10);

? } ;

Obj1 { Inside default Const. of superclass  
        Inside paraConst. of superclass

    } Inside default const of subclass

    } Inside default const of superclass

    } Inside paraConst of superclass

    } Inside default const of subclass

    } Inside paraConst of subclass

Obj2.

Non static blocks inheritance :—

→ When object is created first non-static block is execute then constructor is invoked.

Class A

{ S.O.P ("inside non-static block of superclass");

    A()

{ S.O.P ("inside default Const of superclass");

? } ;

Class B extends A

{

{ S.O.P ("inside non-static block of subclass");

? }

    B()

{ S.O.P ("inside default const. of subclass");

? }

## Class Inherit Demo 14

```
{ public static void main (String args[])
{
    B obj = new B ();
}
```

O/P:

Inside non-static block of superclass  
Inside default Const. of superclass  
Inside non-static block of ~~super~~ class  
Inside default Const. of sub class.

## Static blocks in Inheritance

### Class A

```
{ static
{ S.o.p ("static block of superclass");
}
{ S.o.p ("non-static block of superclass");
}
A ()
{ S.o.p ("Default Const. of superclass");
}
```

Class B extends A

```
{ static
{ S.o.p ("static block of subclass");
}
{ S.o.p ("Non-static block of subclass");
}
B ()
{ S.o.p ("Default Const. of subclass");
}
```

## Class InheritDemo 15

```
{ public static void main (String args[])
    {
        B obj1 = new B();
    }
}
```

Ques:-

static block of Superclass

static block of subclass

Non-static block of Superclass

Default Const. of superclass

Non-static block of subclass

Default Const. of subclass.

What happen when object of subclass is created?

1. Superclass loaded

2. static variables of superclass memory is allocated.

3. static block of superclass.

4. Sub-class loaded.

5. static variable of subclass memory is allocated.

6. ~~allocate~~ the static block of subclass

7. allocate memory for non static variables of superclass & subclass.

8. invoke non static block of superclass.

9. invoke non-Constructor of superclass.

10. invoke non- static block of subclass.

11. invoke Constructor of subclass.

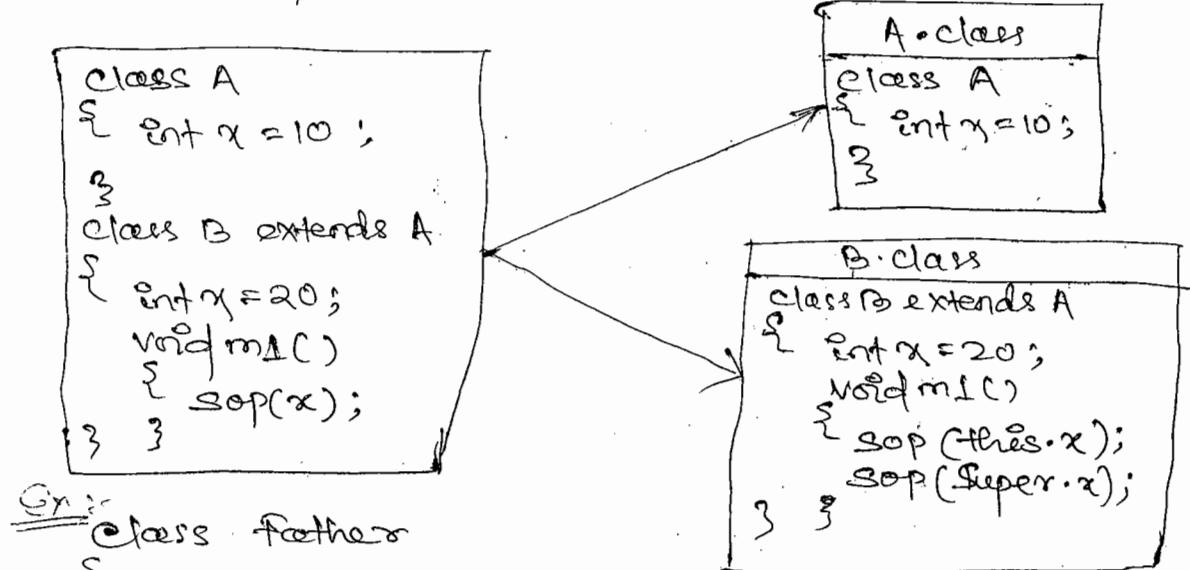
17/10/12

"Super" keyword or Reference:

→ Super keyword is used to refer members of superclass within subclass.

→ When superclass & subclass members are declared with same name, members of superclass refers within subclass using "super".

→ It is used within non-static method of subclass.



Ex :- class Father  
{ String name = "abc"; }  
class Son extends Father  
{ String name = "xyz";  
void printFatherName()  
{ System.out.println(Father.name); }  
void printName()  
{ System.out.println(name); } }

class InheritDemo6  
{ public static void main (String args[])  
{ Son son1 = new Son();  
son1.printName();  
son1.printFatherName(); } }

Output :- xyz , abc ,  
Ex :- class A  
{ int x = 10; }  
class B extends A  
{ int x = 20; → instance variable  
void m1()  
{ int x = 30; → local variable  
}

else,  
 SOP(x);  
 SOP(this.x);  
 SOP(Super.x);

} }

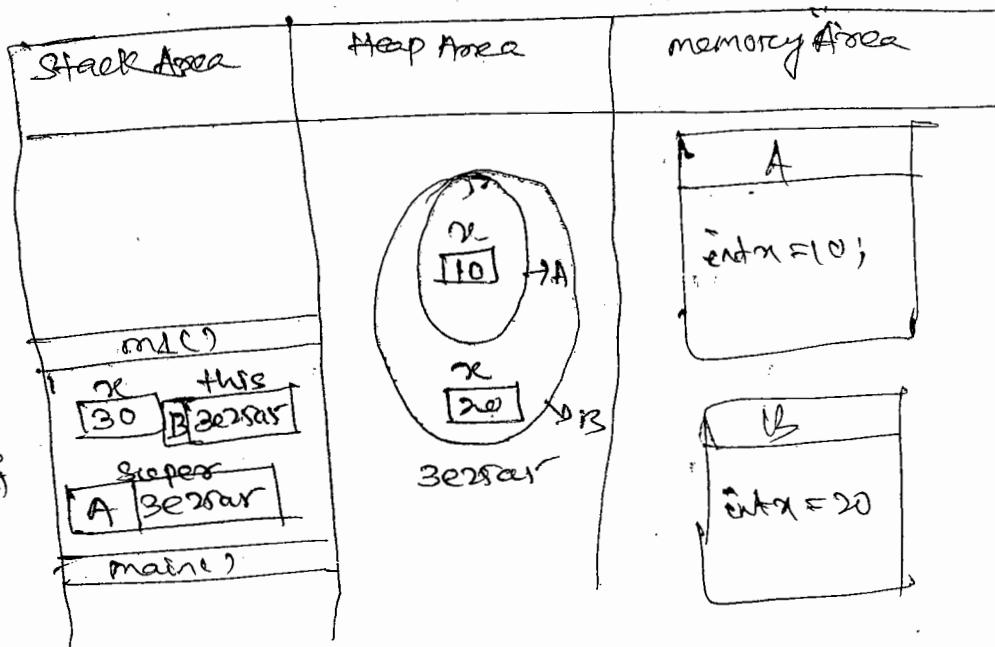
class InheritDemo17

{  
 p.s.v.m (String arr[ ]) }  
 {  
 - B obj1 = new B();  
 Obj1.m1(); } }

Off

30  
20  
10

RAM



this is the reference of B class.  
② Super is reference of A class.

Ex :-

class A

{ int x=10;

} class B extends A

{ int y=20;

void m1()

{

SOP(y); // this.y

SOP(x); // Super.x → Super bcz parent & sub class variables name are not same

class InheritDemo8

{ p.s.v.m (String arr[ ]) }

} } B obj1 = new B();

} } Obj1.m1();

Off

20  
10

### Ex:- Class A

```
{
    static int x = 10;
    int y = 20;
}
```

### Class B extends A

```
{
    static int x = 30;
    int y = 40;
    void m1()
}
```

\* If member is static  
we call it with class  
name or 'super'.  

```
SOP(x); → 30
SOP(y); → 40
SOP(super.x); → 10
SOP(A.x); → 10
SOP(super.y); → 20
```

### Class InheritDemo 9

```
{
    public static void main(String args[])
{
    B obj1 = new B();
    obj1.m1();
}
```

Op

```
30
40
10
10
20
```

### Ex:-

### Class A

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

### Class B extends A

```
{
    int y = 20;
    void m1()
}
SOP(this.y);
SOP(Super.x);
}
```

class Inherit Demo 20

{ p.s.v.m (String args []) }

{ Obj1 = new BC();

Obj1 . m1();

}

}

obj

20

10

next Copy.

Confineee...

:j)



## Method overriding:

- If a derived class needs to have a different implementation of a certain instance method from that of the super class override that instance method in the subclass.
- Note that the scheme of overriding applies only to instance methods.
- For static methods, it is called hiding methods.
- The overriding method has the same name, number and type of parameters and return type as the method it overrides.
- The overriding method can also known return a subtype of the type referred by the overridden method, this is called a covariant return type.
- Redefining of superclass method within subclass is called "method overriding".
- Method is override in order to have a diff. implementation of superclass method within subclass.

(i) Diff between method overloading and method overriding.

### Method overloading

(i) Defining more than one method with same name by changing -

- no. of parameters
- Type of parameters
- Order of parameters

(ii) Overloading is done in same class or in subclass.

### Method overriding

(i) Redefining superclass method in subclass is called method overriding.

(ii) Method is override in subclass.

(iii) method signature  
may not be same .      (iv) In this case signature  
                              must be same .

Modifications in the overriding methods:

- The access specifiers for an overriding method can allow more, but not less, access than the overridden method.
  - For example, a protected instance method in the super class can be made public, but not private in the subclass.
  - You will get a compile time error if you attempt to change an instance method in the superclass to a class method in the subclass & vice versa.

Ex-1

18/10/12

## Class Line

{ void draw(); }

### overridden method

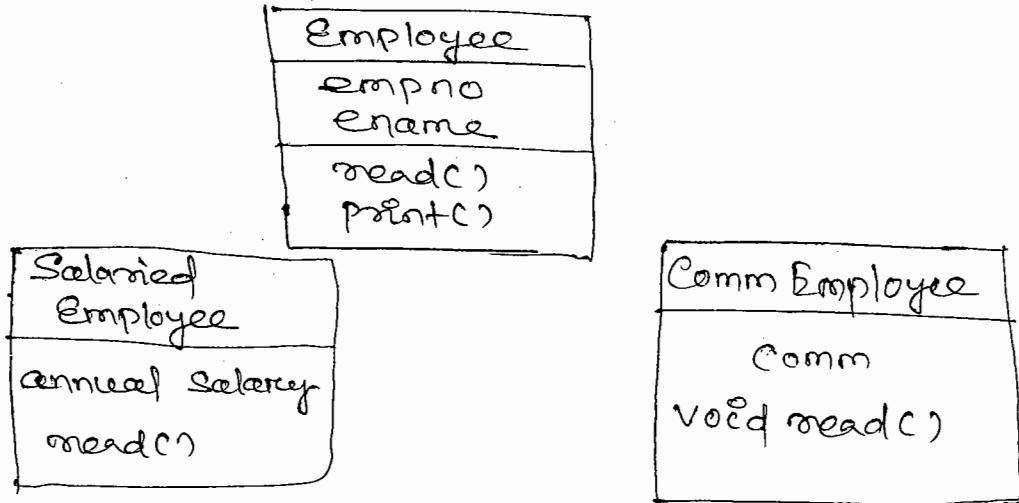
```
{ for (int i=1 ; i<40 ; i++)  
    sop("—");  
}
```

Class Starline extends Line

override  
method

Vocal drawing ← method

```
for (int i=1 ; i<40 ; i++)
    sop (" * ");
```



```

import java.util.*;
{
    private int empno;
    private String ename;
    Scanner scan = new Scanner (System.in);
    void read()
    {
        System.out.print("Input employee no ");
        empno = scan.nextInt();
        System.out.print(" Input employee name ");
        ename = scan.next();
    }
    void print()
    {
        System.out.println("Employee no " + empno);
        System.out.println("Employee Name " + ename);
    }
}
class SalariedEmployee extends Employee
{
    private float aSalary;
    void read()
    {
        super.read();
        aSalary = scan.nextFloat();
    }
    super.print();
    System.out.println("Salary " + aSalary);
}

```



## Class Inherit Demo 23

```
{    PS VM (String arrs [ ] )  
    { child C1 = new Child();  
        C1.walk(); // child walk  
        C1.eat(); // child eat  
    } } }
```

Ex:-

```
class A  
{ void m1()  
{  
    SOP("m1 of super class");  
}}  
  
class B extends A  
{ static void m1()  
{  
    SOP("overriding method");  
}}
```

→ Invalid method.

Ex:-

```
class A  
{ static void m1()  
{  
    SOP("m1 of super class");  
}}  
  
class B extends A  
{ void m1()  
{  
    SOP("overriding method");  
}}
```

→ Invalid method.

Rules in method overriding:

1. If a method is

1. Return type should be same as super class method.

2. Ex:- class A

```
{ void m1()  
{  
    SOP("m1 of super class");  
}}
```

class B extends A

```
{ void m1(){} } → overriding  
{ int m1(){} }
```

2. Static modifier should not be removed or added.

3. Access specifier should be same as super class or it can be increased, but should not be decreased.

Superclass method

private

Subclass method

→ It is not inherited.

(method would not be considered as overridden method)

default → default, protected, public.

protected → protected, public

public → public.

4. throws error should not be added, if super class method doesn't contain it.

Hierarchy of access specifiers :

public  
protected  
default  
private.

Ex :- (Rule - 2) .

Class A  
{  
    static void m1()  
    {  
        }  
    }  
}

class B extends A  
{  
    void m1() { } // not overriding  
    static void m1()  
    {  
        }  
    }  
}

Ex :- (Rule - 3) .

Class A  
{  
    void m1()  
    {  
        }  
    }  
}

Class A  
{  
    final void m1()  
    {  
        }  
    }  
}

Class B extends A  
{  
    public void m1()  
    {  
        }  
    }  
}

Class B extends A  
{  
    protected void m1()  
    {  
        }  
    }  
}

→ Valid overriding .

→ Invalid overriding

## What is Polymorphism?

- The function having single interface, and more than one form is called polymorphism.
- It is something like one name, many forms.
- Polymorphism is the term that describes a situation where one name may refer to different methods.
- Polymorphism is the ability of a programming language to process object differently depending upon their datatypes or class.
- This allows multiple objects of different subclasses to be treated as objects, of a single super class, while automatically selecting the proper methods to apply to a particular object based on the subclass it belongs to.

## Benefit of polymorphism:

### 1. Simplicity:-

- If you need to write code that deals with a family of types, this code can ignore type-specific details and just interact with the type of the family.
- Even though the code thinks it is using an object of the base class, the object class could be actually be the base class or any of its subclass.
- This makes your code easier for you to write and easier for others to understand.

### 2. Extensibility:-

Other subclasses could be added later to the family of types & objects of those new subclasses would also work with the existing code.

## 3 forms of polymorphism in java Program :—

### ① method overriding :—

method of a subclass , overrides <sup>the</sup> methods of super class .

### ② method overriding (implementation) of the abstract methods :—

method of subclass implement the abstract methods of an abstract class .

### ③ method overriding (implementation) through the java interface :—

- methods of concrete class implement the methods of the interface .
- Super class reference variable can hold object of subclass .
- Super class is called broader type & subclass is called narrower type .
- Inorder to write the code that deals with all subclass that code write with supertype and design assign subclass object .

### ④ what is dynamic method dispatching ?

→ The method which is bind at compile time <sup>at run time</sup>,  
is unbinding and bind ~~associated~~ with overriding method of  
Subclass during runtime is called  
"dynamic method dispatching".

Ex : Class A  
{  
    void m1()  
    {  
        System.out.println("Inside m1() of Super class");  
    }  
}

Class B extends A

{  
    void m1()  
    {  
        System.out.println("Inside overriding method of B");  
    }  
}

Class C extends A

```
{ void m1C()
  {
    SOP("Inside overriding method of C");
  }
}
```

Class InheritDemo24

```
{ P.S.V.m (String args[])
  {
    A obj1;
  }
}
```

```
Obj1 = new B();
Obj1. m1C();
Obj1 = new C();
Obj1. m1C();
}
}
```

O/P:- Inside overriding method of B

Inside overriding method of C

→ By assigning subclass object to superclass reference variable, it can invoke the method of superclass inherited in inherited is subclass, but not methods of subclass.

Ex:- Class A

```
{ void m1()
  {
    SOP("Inside m1 of A");
  }
}
```

```
void m2()
{
  SOP("Inside m2 of A");
}
}
```

Class B extends A

```
{ void m1()
  {
    SOP("Inside overriding method");
  }
}
```

```
void m3()
{
  SOP("Inside m3 of A");
}
}
```

Class InheritDemo25

```
{ P.S.V.m (String args[])
  {
    A obj1;
  }
}
```

```
Obj1 = new B();
Obj1. m1();
Obj1. m2();
Obj1. m3(); → // compilation error
}
}
```

O/P :- Inside overriding method  
inside m2 of A.

- Compiler bind method based on reference type.
- JVM bind method based on object type created during runtime.

Ex :-

Class CreditCard

{ void withdraw()

{ SOP("Inside withdraw()");

}

Class VisaCard extends CreditCard

{ void withdraw()

{ SOP("withdraw of VisaCard");

}

Class MasterCard extends CreditCard

{ void withdraw()

{ SOP("withdraw of masterCard");

}

Class Machine

{ static void swap(CreditCard c)

{ c.withdraw();

}

Class InheritDemo26

{ SVM (String args[])

VisaCard v1 = new VisaCard();

MasterCard m1 = new MasterCard();

Machine.swap(v1);

Machine.swap(m1); // m1 is assigning master.

O/P withdraw of VisaCard

    " masterCard.

## Polymorphism using method overriding:

```
import java.util.*;  
class Person  
{  
    private String name;  
    Scanner scan = new Scanner (System.in);  
    void read()  
    {  
        System.out.println("Input name");  
        name = scan.nextLine();  
    }  
    void print()  
    {  
        System.out.println("Name : " + name);  
    }  
}  
  
class Student extends Person  
{  
    private int rno;  
    private String course;  
    void read()  
    {  
        super.read();  
        System.out.println("Input rno");  
        rno = scan.nextInt();  
        System.out.println("Input course");  
        course = scan.nextLine();  
    }  
    void print()  
    {  
        super.print();  
        System.out.println("Rollno " + rno);  
        System.out.println("Course " + course);  
    }  
}  
  
class Lect extends Person  
{  
    private int lectID;  
    private float Salary;  
    void read()  
    {  
        super.read();  
        System.out.println("Input LectID");  
        lectID = scan.nextInt();  
        System.out.println("Input Salary");  
        Salary = scan.nextFloat();  
    }  
}
```

void point( )

{  
    Super. print();  
    SOP("lectID" + lectID);  
    SOP("Salary" + Salary);  
}

class PersonInfo

{  
    static void point(Person p)  
    {  
        p.print();  
    }  
}

class InheritDemo 27

{  
    P.S.V.m (String arg[ ])  
    {  
        Student s = new Student();  
        Lect l = new Lect();  
        s.read();  
        l.read();  
        PersonInfo.print(s);  
        PersonInfo.print(l);  
    }  
}

Q) what is the diff. bet<sup>n</sup> method overriding & method hiding.

method overriding

→ Redefining of super class instance method within subclass is called method overriding.

→ Overriding method participates in inheritance polymorphism.  
→ It doesn't participate even ability of reference to changes behaviour according to

method hiding

→ Redefining of superclass static method within subclass is called method hiding.

Ex:- Class A

```
{ static void main ( )  
{ System.out.println ("Inside superclass method");  
}}
```

Class B extends A

```
{ static void m1 ( )  
{ System.out.println ("Inside subclass method");  
}}
```

Class InheritDemo 28

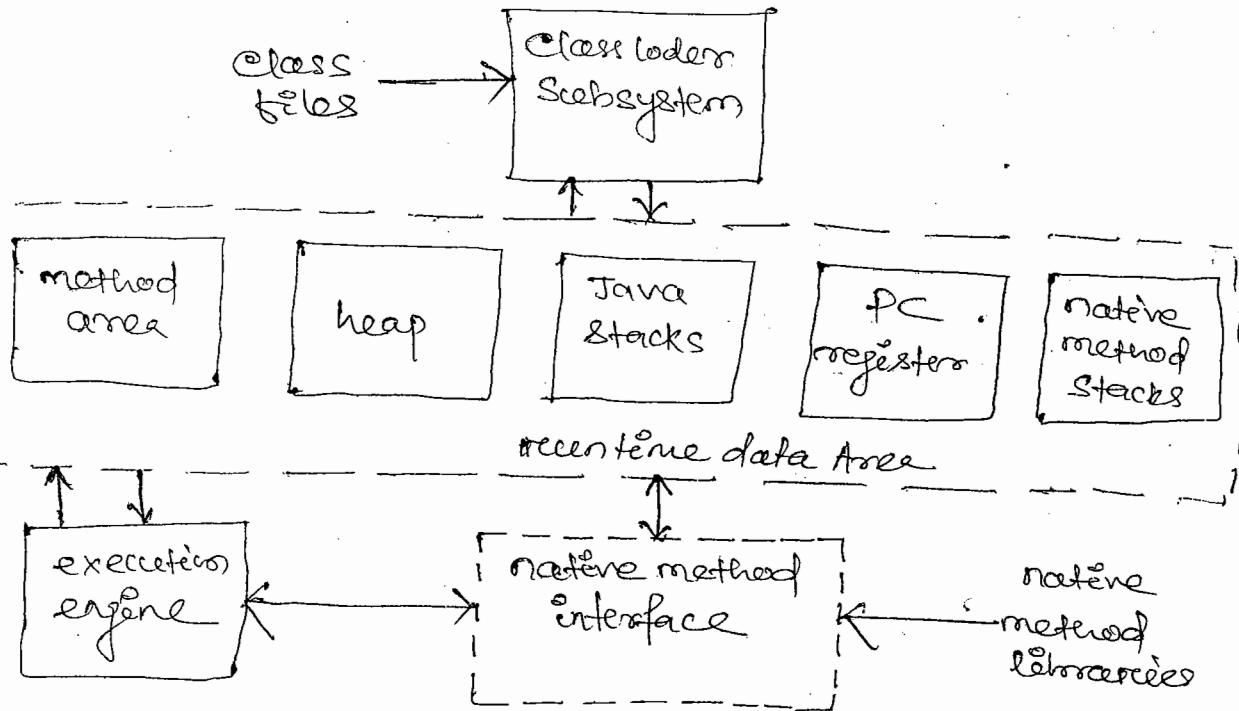
```
{ public static void main ( String args [] )  
{ A objA;  
  objA = new B();  
  objA.m1();  
}}
```

Ques: - Inside superclass method  
Inside subclass method.

JVM Architecture Specification

30/10/12

- JVM is a platform provided by Java to run the Java application or programs.
- JVM provides runtime environment for Java application.
- JVM is platform dependent program.
- JVM is a software developed by C & C++. JVM is a virtual becoz it doesn't have any hardware.
- JVM stands for Java Virtual machine. It is called virtual machine becoz it's build of software.
- For every Java program, JVM allocates memory by dividing into areas which is called runtime data area (RAM).

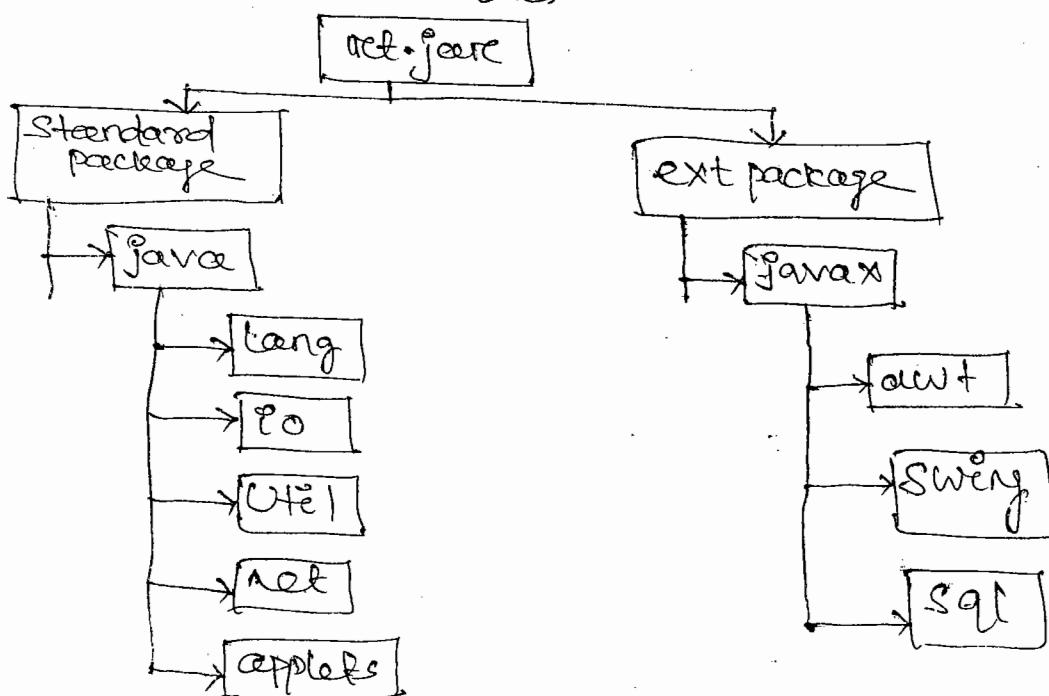


### 1) Class Loader :-

It's a program provided by JVM in order to load class files from secondary storage (hard disk) to primary storage (RAM).

#### types of class loader :-

- 1) Bootstrap class loaders.
- 2) System class loaders
- 3) Extension class loaders.
- 4) Userdefine class loaders.



## ① Bootstrap Class Loader :-

Loads classes from ... / jar / lib / ref.jare if it is the "root" in the class loader hierarchy.

## ② Extension Class Loader :-

Loads classes from ... / jre / lib / ext / \*.jare .

## ③ System Class Loader :-

It is responsible for ~~too~~ loading in the application, as well as for loading classes and resources in the application classpath.

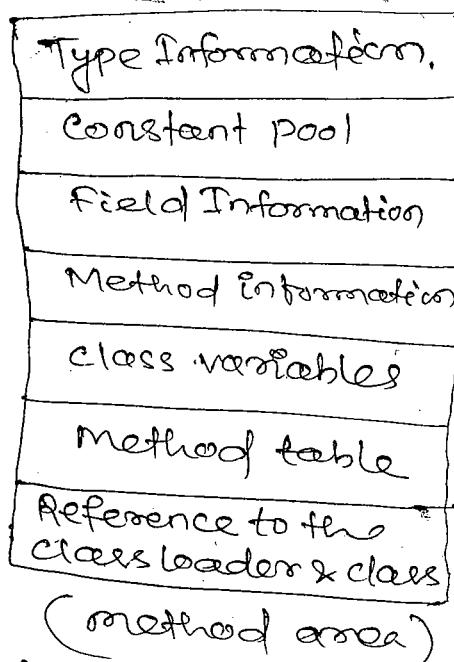
## ④ Method Area :-

→ This is class context area .

→ class loader loads the class from harddisk to method area ( RAM ).

→ Method area is shared area which is shared by more than one thread or method ( function ).

( M )



## Type Information :-

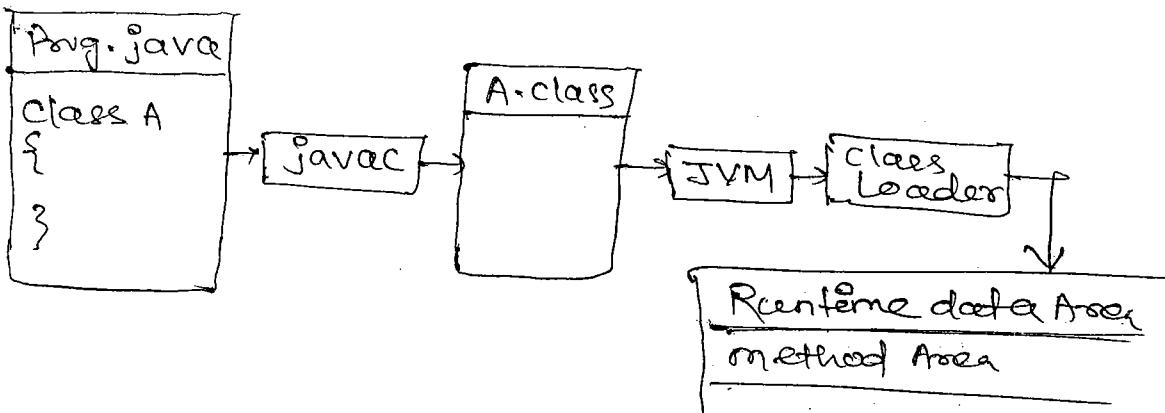
→ fully qualified type's name .

→ fully qualified direct super class name .

→ whether class or an interface .

→ type's modifiers .

→ list of fully qualified name of any direct super interface .



### Constant pool :-

- Ordered set of constants.
- ↳
  - String
  - Integer
  - Floating point
  - final variables
- Symbolic Reference to
  - Types
  - fields
  - methods

### Method area :-

- fields name
- fields type
- fields modifiers (subset)
  - public
  - private
  - protected
  - static
  - final
  - variable
  - transient

### Method Information :-

- method name
- method return type
- Number & type's of parameters.
- modifiers (subset)
  - public
  - private
  - protected
  - static
  - final
  - synchronized
  - native
  - abstract

## Class Variable :

Ordered set of classes, variables.  
Static variables.

## Method Table :

- Used for quick reference to method
- contain name to index in symbol  
reference ; array.

### Class ABC

```

{ public int a;
String str;
abc()
{
    a = 10;
    str = "String 1";
}
public void print()
{
    System.out.println(a + " " + str);
}
  
```

## Heap Area :

- It is shared area which is shared by more than one method or thread.
- It is an object context area.
- Object and array's are allocated in this area.
- Two different threads of the same application however, could not access to each other's heap data.

## Lock on Object :

- Object are associate with a lock (or monitor) to co-ordinate multiple-threaded access to the object.
- Only one thread at a time can "own" an object's lock.
- Once the thread owns a lock, it can request the same lock again multiple times, but then has to release the lock the same number of times before it is made available to other threads.

## Java Stack Area :

Java stack stores a thread's state in discrete form.

- Each

— Local variables area.

Class A

```
class A {  
    static void m3() {  
        int x = 10;  
    }  
    static void m2() {  
        int y = 20;  
    }  
    m3();  
    static void m1() {  
        int z = 30;  
        m2();  
    }  
    public static void main (String args[]) {  
        int p = 40;  
        m1();  
    }  
}
```

Native method Stack Area: —

Non java language method (C, C++), memory is allocated inside this area.

PC Register: —

- Registers it hold data temporary.
- Registers carry data from one place to another within JVM.

## Abstract method and Abstract classes:

- All method which doesn't have implementation (body).
- To create an abstract method just write the method declaration without the body and use the abstract keyword.
- No {} ?

→ Ex:-

// Note that there is no body

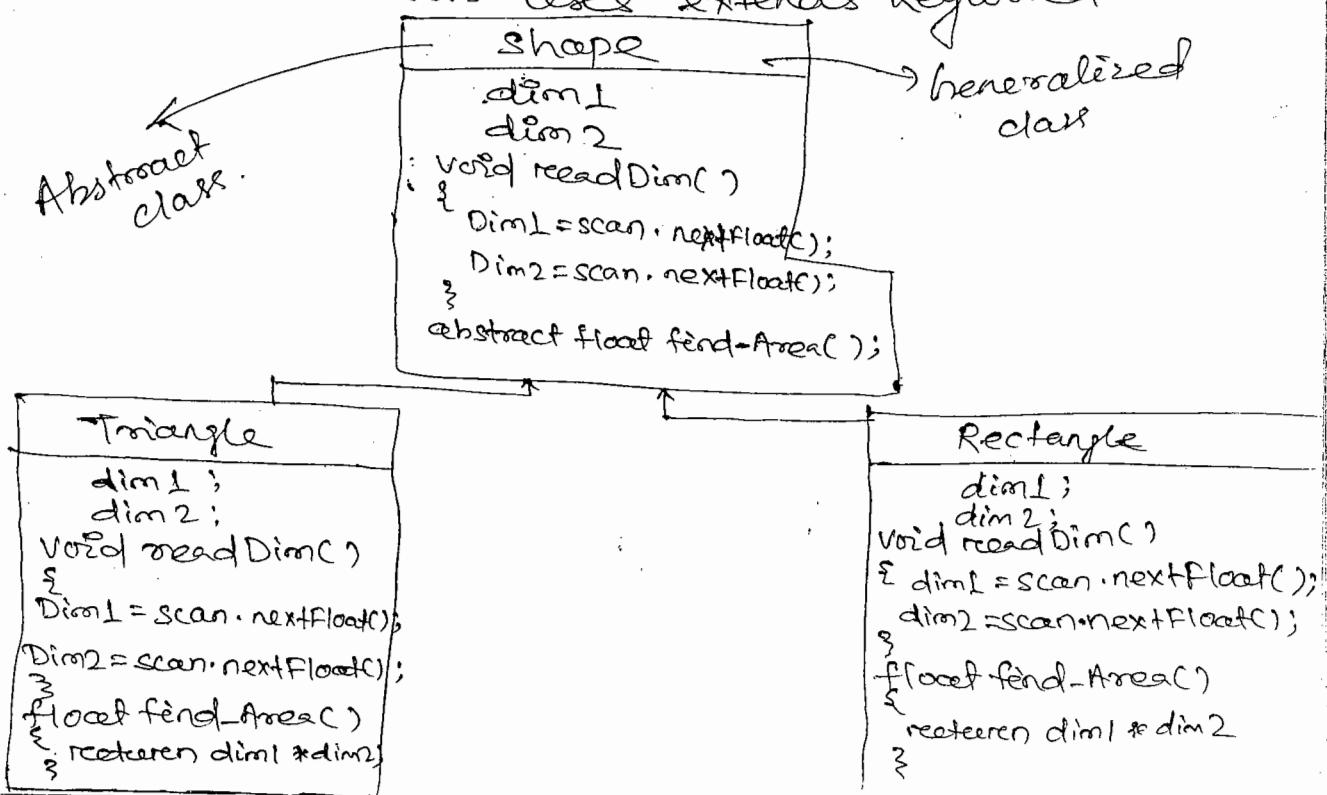
```
public abstract void somemethod();
```

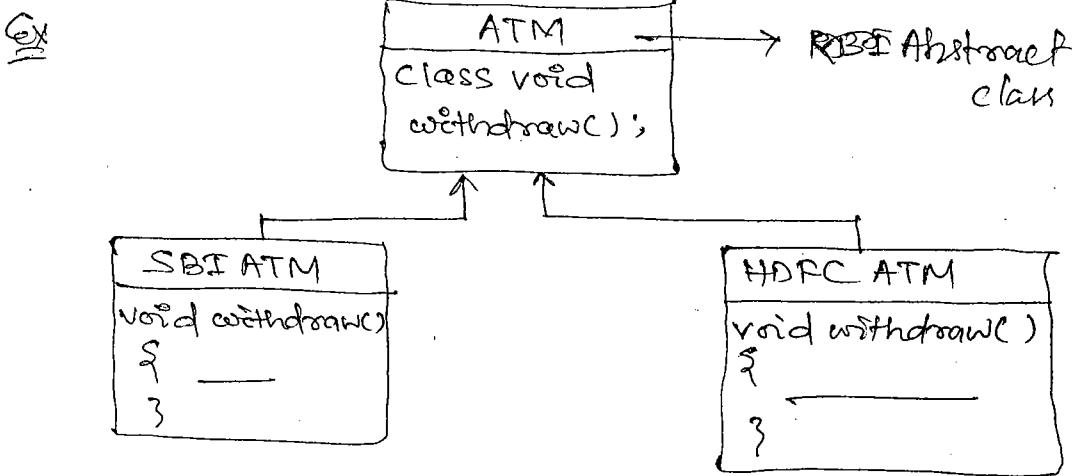
## Abstract Class:

- An abstract class is a class that contains zero or more abstract methods.
- An abstract class can't be instantiated.  
// you will get a compiled error on the following code.

```
MyAbstractClass a1 = new MyAbstractClass();
```

- Another class (concrete class) has to provide implementation of the abstract methods.
- Concrete class has to implement all abstract methods of the abstract class in order to be used for instantiation.
- Concrete class uses extends keyword.





Syntax of abstract class :-

abstract class class-type-name  
 {  
 non-abstract methods;  
 }  
 abstract methods;  
 }

Ex:-

```

abstract class LivingThings
{
    void breath()
    {
        SOP("LivingThings breath");
    }
    void eat()
    {
        SOP("LivingThings eat");
    }
    abstract void walk();
}
  
```

Class Human extends LivingThings  
 {  
 void walk()  
 {
 SOP("Human walk");
 }
}

Class AbstractDemo1  
 {  
 P.S.V.m (String args[])
 }

```

Human b1 = new Human();
b1.breath();
b1.eat();
b1.walk();
  
```

Output  
 LivingThings breath  
 LivingThings eat  
 Human walk

Q) When to use abstract methods and abstract class?

- Abstract methods are usually declared where two or more subclasses are expected to fulfill a similar role in different ways through different implementation (Polymorphism).
- These subclasses extends the same abstract class and provide different implementation for abstract methods.
- Use abstract classes to define broad types of behaviours at the top of an object-created program class hierarchy and uses its subclasses to provide implementation details of the abstract class.

Ex:-  
Abstract class A  
{  
    abstract void m1();  
}  
Class B extends A  
{  
    void m2()  
    {  
        System.out.println("m2");  
    }  
}

\*→ The above prog. displays CE becoz class which extends an abstract class must override abstract method.

Ex:-  
Abstract class A  
{  
    void m1()  
    {  
        System.out.println("m1");  
    }  
}  
Class Abstract Demo3  
{  
    public static void main(String args[])  
    {  
        A obj1 = new AC();  
    }  
}

\*→ A is abstract & can't be instantiated. The above prog. displays CE becoz A is an abstract class which can't be used for creating object.

Ex:-

```
import java.util.*;  
abstract class Shape  
{  
    float dim1, dim2;  
    Scanner scan = new Scanner (System.in);  
    void readDimc()  
    {  
        System.out.println("Input dim1, dim2");  
        dim1 = scan.nextFloat();  
        dim2 = scan.nextFloat();  
    }  
    abstract float findArea();  
}
```

Class Triangle extends shape

```
{  
    float findArea()  
    {  
        return 0.5f * dim1 * dim2;  
    }  
}
```

Class Rectangle extends shape

```
{  
    float findArea()  
    {  
        return dim1 * dim2;  
    }  
}
```

Class AbstractDemo4

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

```
    Triangle t = new Triangle();
```

```
    Rectangle r = new Rectangle();
```

```
    t.readDimc();
```

```
    r.readDimc();
```

```
    float area1 = t.findArea();
```

```
    float area2 = r.findArea();
```

```
    System.out.println("Area of triangle "+area1);
```

```
}      {  
    System.out.println("Area of rectangle "+area2);  
}
```

Date - 29/10/12

→ Abstract class can be used for creating reference variable, but abstract class cannot be used for creating object.

→ It is used for achieving multiple polymorphism.

Ex :-

```
import java.util.*;  
abstract class Employee  
{  
    private int empno;  
    private String ename;  
    Scanner scan = new Scanner (System.in);  
    void readEmployee()  
    {  
        System.out.println("Enter empno");  
        empno = scan.nextInt();  
        System.out.println("Enter name");  
        ename = scan.next();  
    }  
    String getEmployee()  
    {  
        return empno + " " + ename;  
    }  
    abstract float calcSalary();  
}
```

```
Class Salaryied Employee extends Employee  
{  
    private float salary;  
    void readSalary()  
    {  
        System.out.println("Enter salary");  
        salary = scan.nextFloat();  
    }  
    void calcSalary()  
    {  
        System.out.println("Salary + profit + awf");  
    }  
}
```

```
Class Worker extends Employee  
{  
    private float wage;  
    private int days;  
    void readWageDays()  
    {  
        System.out.println("Enter wage");  
    }  
}
```

```

wage = scan.nextfloat();
SOP("Input days");
days = scan.nextInt();
}
float calcSalary()
{
    return wage * days;
}
}

```

### Class Printer

```

static void printPayslip(Employee e)
{
    SOP(e.getEmployee());
    SOP(e.getCalcSalary());
}

```

### Class AbstractDemo

```

P.S.V.m(String args[])
{
    SalariedEmployee e1 = new
        SalariedEmployee();
}

```

```

Worker e2 = new Worker();
e1.readEmployee();
e1.readSalary();
e2.readEmployee();
e2.readWageDays();
Printer.printPayslip(e1);
}
}

```

→ If method having parameter of type abstract class it receives hashCode of subclass (Concrete Class).

→ If method having parameter of type abstract. It receives subclass object.

## Reference conversion:-

- Converting one reference variable type to another reference variable is called reference conversion.
- This conversion is between super and subtypes.

1. Widening reference conversion.

2. Narrowing reference conversion.

## Widening Reference Conversion:-

- Converting narrower reference type to broader reference type is called widening reference conversion.

→ This conversion is implicit conversion.

→ Subclass → Narrower type

Superclass → Broader type

Ex:-

Class A

```
{ void m1()
  {
    sop("m1");
  }
  void m2()
  {
    sop("m2");
  }
```

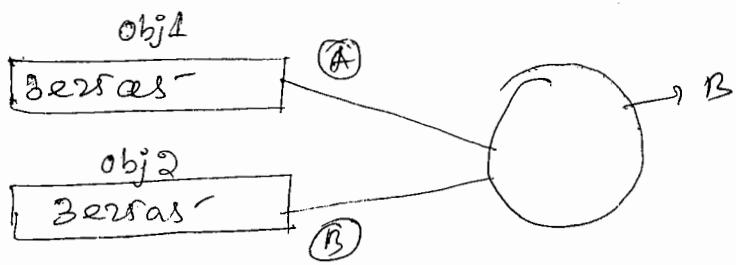
Class B extends A

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

Class RefDemo1

```
{ P.S.V.m(String args[])
  {
    A obj1;
    B obj2 = new B();
    obj1 = obj2; // widening conversion
    obj1.m1();
    obj2.m2();
    obj1.m3(); → Error
  }
```

O/P



Ex:-

Class A

```

    void m1()
    {
        sop("m1 of A");
    }

    void m2()
    {
        sop("m2 of A");
    }
  
```

Class B extends A

```

    void m1()
    {
        sop("m1 of B");
    }

    void m3()
    {
        sop("m3 of B");
    }
  
```

Class RefDemo2

```

P.S.V.m(new B());
A obj1;
B obj2 = new B();
obj1 = obj2;
obj1.m1();
obj1.m2();
  
```

O/P:

m1 of B

m2 of A

Date - 30/10/12

## Narrowing Reference Conversion:-

- Converting of broader reference type to narrow reference type is called narrowing reference conversion.
- This conversion is explicit. It has to be done using typecasting.

Ex:- Class A

```
{ void m1()
  {
    { sop ("m1 of A");
    }
  void m2()
  {
    { sop ("m2 of A");
    }
  }
```

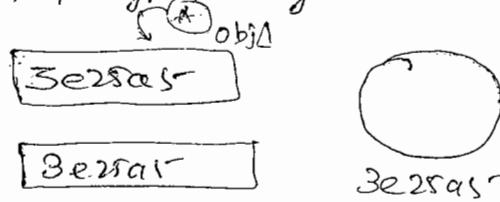
Class B extends A

```
{ void m3()
  {
    { sop ("m3 of B");
    }
  }
```

Class RefDemo3

```
psvm (String arr[ ])
```

```
A obj1 = new B();
obj1.m1();
obj1.m2();
obj1.m3(); → // Invalid
B obj2 = B obj1; → // Typecasting
obj2.m1();
obj2.m2();
} } obj3.m3();
```



- Conversion is always between the references not between the objects.

Ex-2

Class Account

```
{ void m1()
{ sop("m1 of Account");
}
```

Class SavingAccount extends Account

```
{ void m2()
{ sop("m2 of SavingAccount");
}
```

Class CurrentAccount extends Account

```
{ void m3()
{ sop("m3 of CurrentAccount");
}
```

```
public static void main(String args[])
{
    Account acc = new SavingAccount();
    acc.m1();
}
```

Saving Account sacc = (Saving Account) acc;

sacc.m1();

sacc.m2();

acc = new CurrentAccount();

acc.m1();

Current Account caacc = (Current Account) acc;

caacc.m1();

} caacc.m3();

→ This conversion is explicit.

Ex-3

Class A

```
{ int x;
}
```

Class RefDemo3

```
static boolean compare(Object obj1, Object obj2)
```

```
A @1 = (A) obj1;
```

```
A @2 = (A) obj2;
```

```
if ( $\text{@1} \cdot x == \text{@2} \cdot x$ )
    reteeren true;
else
    reteeren false;
}
P.S.V.m ( Strong args[] )
Aobj1 = new AC();
Aobj2 = new AC();
Obj1.x = 10;
Obj2.x = 20;
if (compare (obj1, obj2))
    sop ("equal");
else
    sop ("not equal");
}
OP not equal
```

2)

# Interfaces

- It defines a standard and public way of specifying the behaviour of classes.
- Defines a contract.
- All methods of an interface are abstract method.
- Defines the signatures of a set of methods without body (implementation of the methods).
- A concrete class ~~cannot~~ must implement the interface (all the abstract methods of interface).
- It allows classes regardless of their location in the class hierarchy to implement common behaviours.
- It is a call of abstract methods and static final variables (constants).
- Interface defines the specifications.

## Why do we use Interfaces?

### Reason - 1:-

- To revel on object programming interface (functionality of the object) without revealing its implementation.
- This is the concept of encapsulation.
- The implementation can change without affecting the caller of the interface.
- The caller doesn't need the implementation at the compile time.
- It needs only the interface at the compile time.
- During runtime, actual object instance is associated with the interface type.

### Reason - 2 :-

- To have unrelated classes implement similar methods.
- One class is not a sub-class of another.

Ex:-

Class Line and Class MyInteger

- They are not related through inheritance.
- We want both to implement comparison methods.
  - checkGreater (Object x, Object y)
  - checkLess (Object x, Object y)
  - checkEqual (Object x, Object y)

Q) Define comparison interface which has the 3 abstract methods above?

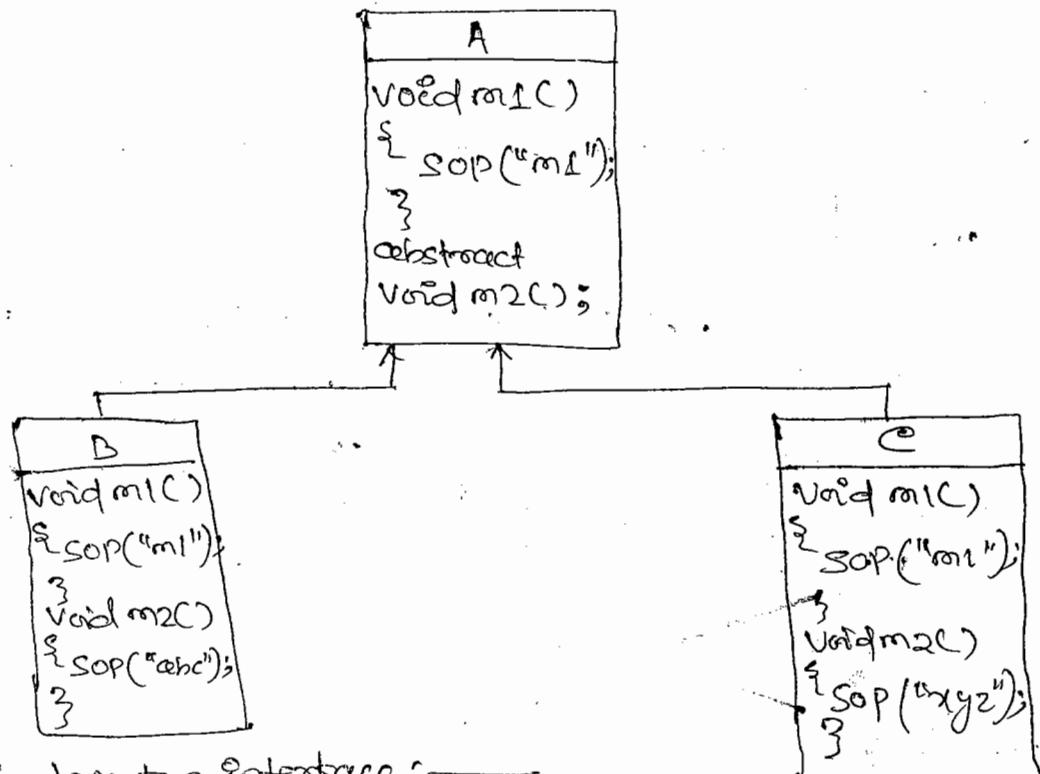
### Reason - 3 :-

- To model multiple inheritance - we want to impose multiple set of behaviors to your class.
- Each set is defined as an interface.
- A class can implement multiple inheritance interfaces while it can extend only one class.
- When more than one subclasses having similar forms of implementation, then we use interfaces else abstract class.

Q) When to use an abstract class over interface?

- for non-abstract methods, we want to use them when we want to provide common implementation code for all subclasses.
- Reducing the application.
- for abstract methods the motivation is some with the ones in the interface to impose a common behavior for all subclasses without defining how to implement it.
- Remember a concrete class can extend only one super class whether that class is in the form of concrete class or abstract class.

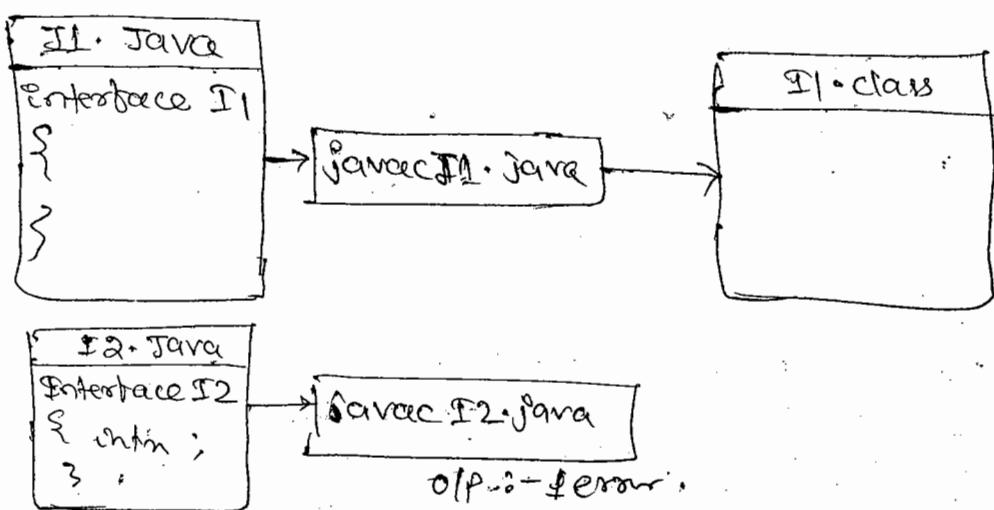
Ex  
==  
be



### Syntax for Interface:

```
interface Interface-type-name
{
    constants;
    abstract methods;
}
```

- Interface is one type of class which is inherited but not instantiated.
- We can't create object of Interface.



### Exp:-

The above program displays compilation error  
becoz interface doesn't allow variables

Interface I3 :

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

I3.java

```
interface I3
{ int x = 10;
}
```

javac I3.java

I3.class

```
interface I3
{ public static final int x = 10;
}
```

I4.java

```
interface I4
{ interface
  int x = 10;
}
```

javac I4.java

error

becoz interface allows only one  
access specifier i.e public.

I4.java

```
interface I4
{ private int x = 10;
}
```

javac I4.java

error

I5.java

```
interface I5
{ static int x = 10;
}
```

javac I5.java

I5.class

```
interface I5
{ public static final int x = 10;
}
```

I6.java

```
interface I6
{ void m1();
}
```

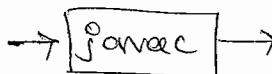
javac I6.java

{ error becoz interface  
doesn't allow concrete  
method with body.

## I7.java

### Interface I7

```
{
    void m1();
    abstract
    void m2();
    public abstract
    void m3();
}
```



## I7.class

### Interface I7

```
{
    public abstract
    void m1();
    public abstract
    void m2();
    public abstract
    void m3();
}
```

Ex

## Q. How to inherit Interface?

- An Interface is inherited by a class using "implements" keyword.
- A class implements interface but does not extends.
- A class extends class and implements one or more than one interface.
- A class which implements interface is called "concrete class".
- Concrete class must provide implementation of abstract methods (ore) concrete class must override abstract methods.

### Syntax:

```
class class_name implements
    interface_name1, interface_name2, ...
{
    fields;
    methods;
}
```

(or)

```
class class_name extends Super_classname
    implements interface_name1, interface_name2 ...
{
    fields;
    methods;
}
```

C

→ T

C

Eg:

0/1

→ 1

8

+

6/5

Ex 2:-  
Interface I1

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

Class C1 extends I1

```
{
```

```
}
```

O/P :- Error

→ This above program displays CE becoz class cannot extends interfaces.

Eg:

Interface I1

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

Class C1 implements I1

```
{
```

void m1() ← override

```
{ sop("m1");  
}
```

```
}
```

O/P :- Error

→ This above program displays CE becoz overriding method allow higher or equal access specifiers than overridden method.

Eg:- Interface I1

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

Class C1 implements I1

```
public void m1()
```

```
{
```

```
 sop("m1");  
}
```

```
}
```

## Opp! - Error .

→ The above program displays an CE becoz  
the class which implements the interface,  
that class must be provides the method  
body of all the ~~an~~ abstract methods .

Diffs. bet<sup>n</sup> Interface and abstract class.

### Interface

- All methods ~~are~~ in interface  
are abstract methods.
- allows only constants.

- Members of interfaces.  
are public

- This is developed  
irrespective of subclasses.  
(i.e. no direct relationship  
with subclasses.)

Ex: interface Compare  
{ boolean gt;  
boolean lt;

{  
class Triangle implements  
Compare  
{}  
  
}  
no relation  
bet<sup>n</sup> {  
class Rectangle implements  
Compare  
{}  
}  
Sub classes

### Abstract class

- Allows abstract method &  
non-abstract methods.

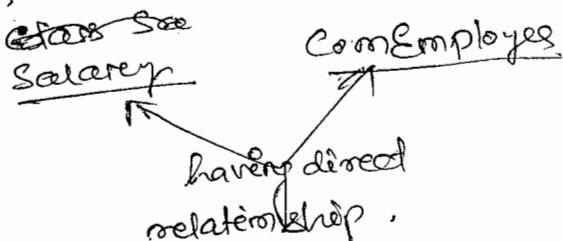
- Allows fields and  
constants.

- members of this class  
can be private, public,  
protected or default.

- This class is having  
direct inheritance  
relationship with subclasses.

Ex: abstract class Employee

{  
empno;  
empname;  
}



a) Why interface?

Interfaces are used for achieving —

(1) Multiple Inheritance.

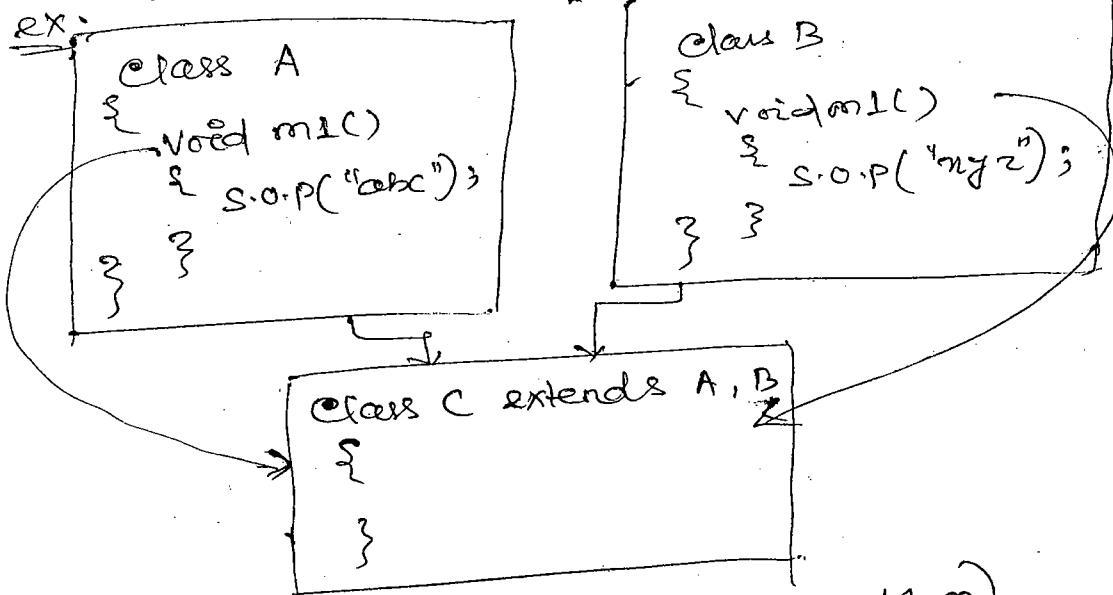
(2) Interface is developed irrespective of subclasses,  
where one class does not share the properties of another  
class.

(3) Runtime polymorphism.

## Multiple inheritance :-

→ Java does not support multiple inheritance using class becoz ambiguity problem.

ex:-



(ambiguity / duplication problem)

- This problem can be avoided by interfaces.
- Interfaces can be used to achieve multiple inheritance.
- A class implements more than one interface.
- A class extends only one class.
- A class extends one class and implements more than one interface.

ex:-

Interface I1

{ void m1();  
}

Interface I2

{ void m1();  
}

Class C implements I1, I2

{ public void m1()

{ S.O.P ("Implementation of I1 method");  
}

\* Methods of interfaces are by default abstract.

## Class InterfaceDemo4

```
{ public static void main (String args[])
{
    C1 obj1 = new C1();
    obj1.m1();
}
```

Q1 Implementation of I1 method.

Q2 What is the diff. b/w extends and implements?

extends

(1) It allows reusability and extensibility.

implements

(1) It allows reusability but does not allow extensibility.

(2) A class extends another class and can implement one or more than one interface.

extends another interface.

(2) A class implements "is-a" and "has-a" relationship.

(3) It creates "is-a" and "has-a" relationship.

Ex:-

### Interface I1

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

### Class C1

```
{ public void m1()
```

```
{     System.out.println("abc");
```

Class C2 extends C1 implements I1

```
{ void m2()
```

```
{     System.out.println("xyz");
```

## Class Interface Demo's

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

{  
    C2 obj1 = new C2();  
    obj1.m1();  
}

3  
3

O/P: abc

Ex

class LogonWindow extends Window implements  
WindowListener,

multiple inheritance,

→ A class inherit more than one superclass  
or types is called "multiple inheritance".

2. Interface is developed irrespective of subclass  
where one class doesn't share the  
properties of another class.

Ex:

interface Operations

{  
    void moveLeft();  
    void moveRight();  
}

Class Dog implements Operations

{  
    public void moveLeft()  
    {  
        S.O.P("MoveLeft of Dog");  
    }  
    public void moveRight()  
    {  
        S.O.P("MoveRight of Dog");  
    }  
}

Class Cane implements Operations

{  
    public void moveLeft()  
    {  
        S.O.P("MoveLeft of Cane");  
    }  
}

```

public void moveRight()
{
    s.o.p("moveRight of circle");
}

```

### Class Interface Demo 6

```

public static void play (Operations S)

```

```

{
    S.moveLeft();
    S.moveRight();
}

```

public static void main (String args[])

```

{
    play (new Circle());
    play (new Dog());
}

```

O/P  
MoveLeft of Circle  
MoveRight of Circle  
MoveLeft of Dog  
MoveRight of Dog

21/11/12

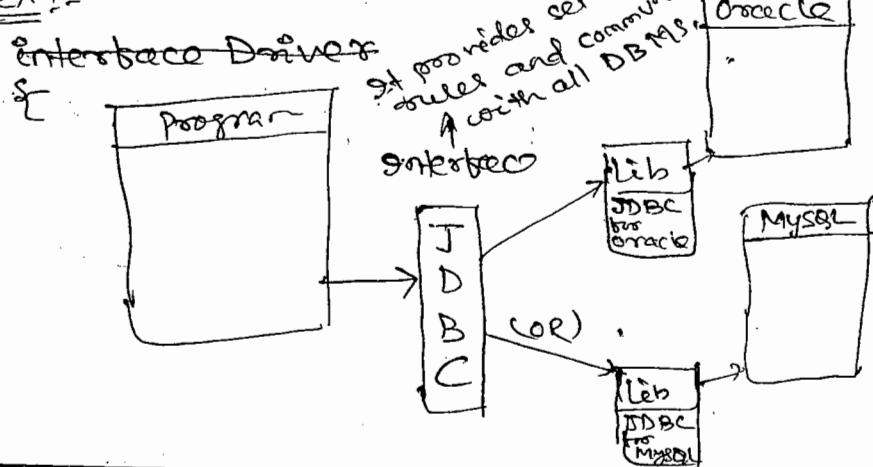
### ③ Reason # 1 (Reentame polymorphism)

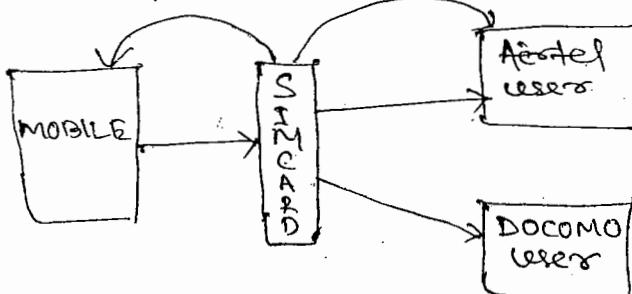
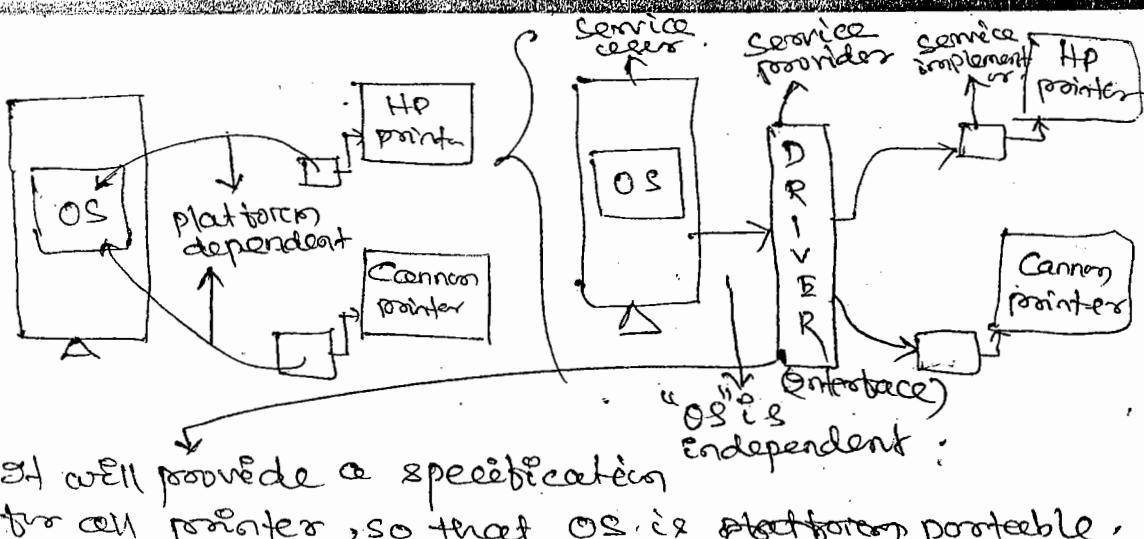
→ To control an object's programming interface (functionality of the object) without revealing its implementations

- This is the concept of encapsulation.

- The implementation can change without affecting the callers of the interface

Ex:- See previous note (page)





ex:-

### Interface Driver

```
{
    void connect();
}
```

Class HPDriver implements Driver

```
{
    public void Connect()
    {
        System.out.println ("Connect to HP Printer");
    }
}
```

Class CannonDriver implements Driver

```
{
    public void connect()
    {
        S.O.P ("Connect to Cannon driver");
    }
}
```

### Class OS

```
{
    static void install (Driver d) // OS is the caller
                                    of the interface
}
```

```
{
    d.connect();
}
```

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

HP  
Printer

install(newCanonDriver());  
} } .

Q.P.: Connect to HP Printer

Note: Connect to Canon printer

\* → It's method having parameters of type interface,  
it receives an object of concrete class

\* → It's method having return type as an  
interface it returns hashCode of concrete  
class object.

Ex: ① void service(ServletRequest re1, ServletResponse re2)  
{  
} } .  
② ~~Driver d = new HPDriver();~~

\* → We cannot create an object of interface.

\* → We can create reference variable of type  
reference in order to achieve Runtime  
Polymorphism.

\* → Reference variable of type Interface hold an object  
of Concrete class.

Ex:-

```
import java.util.*;  
interface PersonInfo  
{  
    void read();  
    void print();  
}
```

Class Employee implements PersonInfo

```
{  
    private int empno;
```

```
    private String name;
```

```
    Scanner scan = new Scanner(System.in);
```

```
    public void read()
```

```
{  
    S.O.P("Input empno");
```

```
    empno = scan.nextInt();
```

```
    S.O.P("Input name");
```

```
    name = scan.next();
```

public void point()

{  
    S.O.P ("Employee No" + empno);  
    S.O.P ("Employee Name" + name);  
}

Class Supplier implements PersonInfo

{  
    private int supplId;  
    private String name;  
    Scanner scan = new Scanner (System.in);  
    public void read()  
    {  
        S.O.P ("Input SupplierId");  
        supplId = scan.nextInt();  
        S.O.P ("Input SupplierName");  
        name = scan.next();  
    }  
}

public void point()

{  
    S.O.P ("SupplierId" + supplId);  
    S.O.P ("SupplierName" + name);  
}

Class PersonOperations

{  
    static void read (PersonInfo p)  
    {  
        p.read();  
    }

    static void point (PersonInfo p)  
    {  
        p.print();  
    }

// read() & point() are caller of the employee.

Class InterfaceDemo8

{  
    public static void main (String args[])  
    {  
        Employee e = new Employee();  
        PersonOperations.read (e);  
        PersonOperations.point (e);  
    }

```

Supplier s = new Supplier();
PersonOperations.read(s);
PersonOperations.print(s);
}
}

```

Problem of Rewriting an Existing Interface:

→ Consider an interface that you have developed called Dolt:

```

public interface Dolt
{
    void doSomething(int i, double x);
    int doSomethingElse(String s);
}

```

→ Suppose that, after some time, you want to add a third method to Dolt, so that the interface now becomes:

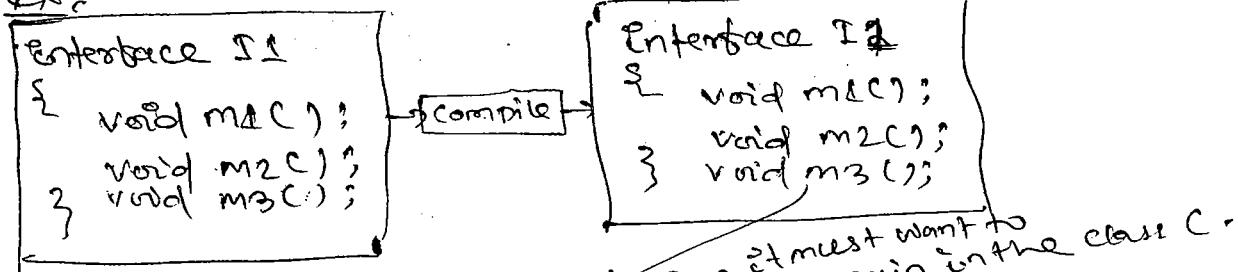
```

public interface Dolt
{
    void doSomething(int i, double x);
    void doSomethingElse(String s);
    boolean didItWork(int i, double x, String s);
}

```

→ If you make this change, all classes that implements the old Dolt interface will break bcoz they didn't implement all methods of the interface anymore.

Ex:



```

class C implements I1
{
    public void m1()
    {
        System.out.println("ABC");
    }
    public void m2()
    {
        System.out.println("XYZ");
    }
}

```

```

c.m1();
c.m2();

```

## Solution of Rewriting an Existing Interface:

11/12

- Create more interface later.
- for example, you could create a Dolt plus interface that extends Dolt:

public interface DoltPlus extends Dolt

{

    boolean didItWork(int i, doable x, String s);

}

- Now users of your code can choose to continue to use the old interface or to upgrade to the new interface.

3/11/12

## Syntax for extending Interface:

```
interface Interface-type-name extends
    Interface-type-name1, Interface-type-name2, ...  

    {  

        Constants;  

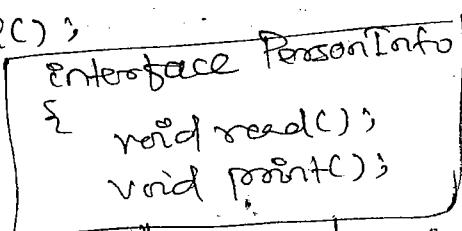
        abstract methods;  

    }
```

→  
C  
M  
C  
SC

Ex:-  
interface StudentInfo

{  
    void read();



interface StudentInfo  
extends

{  
    // void read();  
    // void print();  
    void float Result();  
}

3

interface EmployeeInfo  
extends

{  
    // void read();  
    // void print();  
    void float Salary();  
}

3

- StudentInfo & EmployeeInfo interfaces are extends PersonInfo interface. So common methods are not required in subclass child interfaces.

Ex:-

Interface A

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

}

Interface B extends A

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

}

Interface C extends A

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

}

Class C1 implements B

```
{ public void m1()
```

```
{ s.o.p("m1 of c1");
```

}

```
public void m2()
```

```
{ s.o.p("m2 of c1");
```

}

Class C2 implements C

```
{ public void m1()
```

```
{ s.o.p("m1 of c2");
```

public void m3()

```
{ s.o.p("m3 of c2");
```

}

When more than one interface has same behaviour  
 at that time we write the method one time & used  
 more than one time. i.e. we create a super interface &  
 write similar methods and extends it to child type.

Class InterfaceDemo

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

```
{ C1 obj1 = new C1();
```

```
    C2 obj2 = new C2();
```

```
    obj1.m1();
```

```
    obj1.m2();
```

```
    obj2.m1();
```

```
    obj2.m3();
```

}

~~obj~~  
 {  
 m1 of c1  
 m2 of c1  
 m1 of c2  
 m3 of c2 .

→ An Interface can extends more than one interface.

→ A class which implements the interface must be

override all the abstract methods.

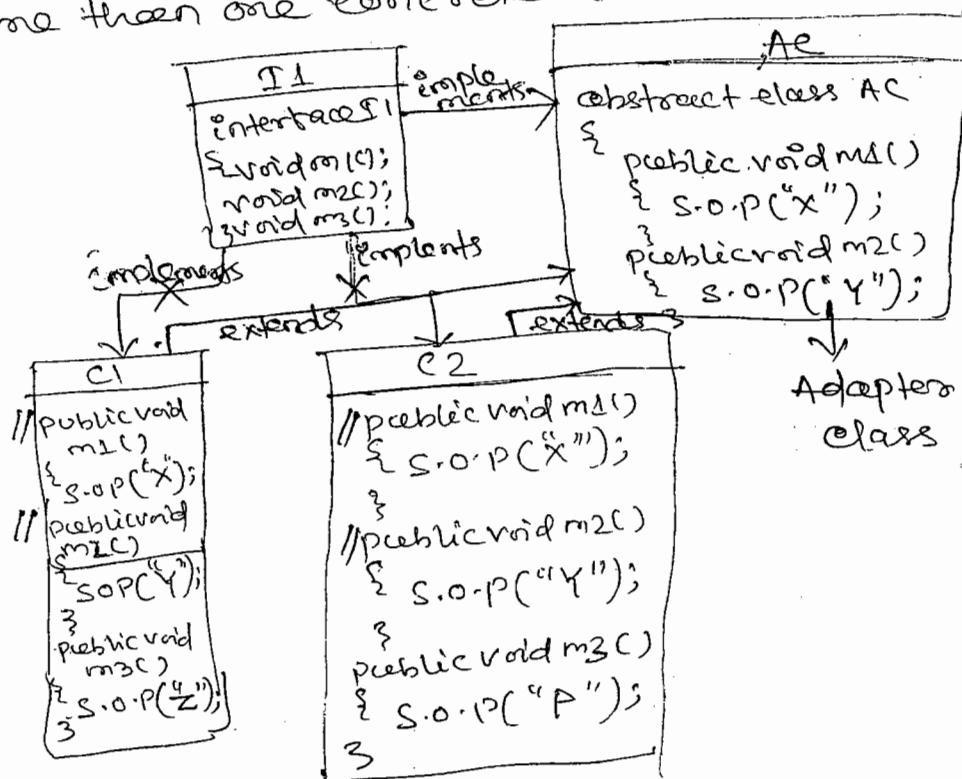
## Adapter class:

→ Adapter class is an abstract class, which provides a partial implementation of abstract method.

→ Adapter class implements one or more than one interface.

→ Adapter class provide common implementation of more than one concrete class.

Ex:-

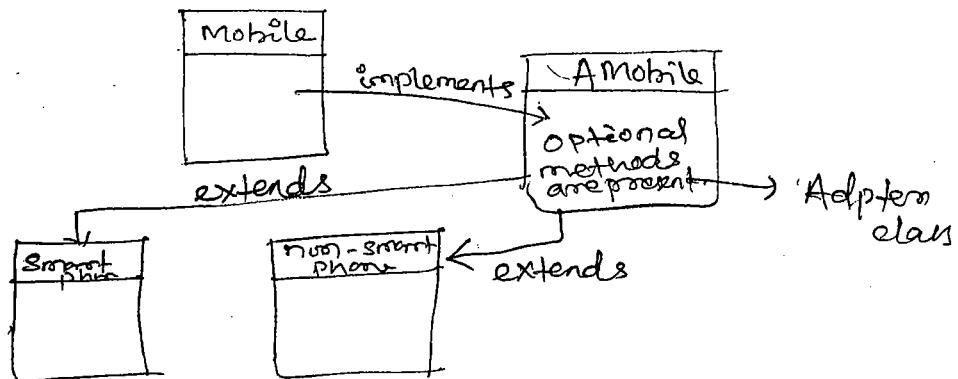


→ Here `m1()` & `m2()` methods are same implementation in **C1** & **C2** class. So it leads to code redundancy. To avoid it we will provide an Adapter class (AC). and define the common methods of **C1** & **C2** at that class.

→ If class implements interface called Concrete class and it must override all abstract methods.

→ If abstract class implements interface, not required to override all abstract methods.

Ex:-



### Marker Interface:

- An interface which does not have any methods that is called "marker interface".
- Marker Interface work as a role or eligibility to perform a particular type of operation.

### Instance of operator:

- This operator returns boolean value.
- This operator check a reference hold, which type of object.

Ex:-

interface DBA

{

}

class DBOperations

{

    static void create(Object o)

    if (o instanceof DBA)

        S.O.P("Object created");

    else

        S.O.P("not eligible to create");

}

}

class A

{

    int x;

}

Class B implements DBA

```
{   entry; // Eligible to create object bcoz  
    B is marked.  
}
```

class InterfaceDemo10

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

```
{   A obj1 = new A();
```

```
    B obj2 = new B();
```

```
    DBOperations.create(obj1);
```

```
    DBOperations.create(obj2);
```

```
{ }
```

O/P:- Not elig to create.  
object created.

a) what is tag interface?

→ Alternative name of marker interface is

Taginterface.

4/11/12

eg:- The example of marker interface is

Serializable, cloneable.

→ Both Serializable & cloneable are pre-defined  
marker interfaces provided by java.

ex:- interface VoterCard

```
{
```

```
}
```

class Employee implements VoterCard

```
{   int empno;
```

```
    String name;
```

```
}
```

class Student

```
{   int rno;
```

```
    String name;
```

```
}
```

Q)

(a)

(b)

(c)

(d)

(e)

(f)

## Class Interface Demo 12

```
{ void static vote ( Object o )
  { if ( o instanceof VoterCard )
      s.o.p( "Elg to Vote" );
    else
      s.o.p( "Not-Elg to Vote" );
  }
  public static void main ( String args[] )
  {
    Student stud1 = new Student();
    Employee emp1 = new Employee();
    Vote(stud1);
    Vote(emp1);
  }
}
```

O/P: Not-Elg to Vote  
Elg to Vote.

Q) Choose which is correct?

- (a) class extends a class (true)
- (b) interface extends one or more than one interface (T)
- (c) interface implements interface (F)
- (d) class implements one or more than one interface (T)
- (e) class extends ~~one~~ more than one class (F)
- (f) class extends a class and implements one or more than one interface (T)

## Inner Classes:

- A class within a class called "Inner class".
- Before we write inner class which provides avoiding
  - ① Redundancy
  - ② Modularity

ex:-

Class Person

{ String name;

Replace it

|                              |
|------------------------------|
| String tno, tstreet, tcity;  |
| String phno, pstreet, pcity. |

Inner class  
class Address  
{ String hno;  
String street;  
String city;

Avoids Redundancy.

Address - add = new Address();

Address - Padd = new Address();

Q) what are the different inner classes types ?

→ These inner classes are 4 types -

- ① Member class / Nested class
- ② Nested local level class
- ③ Local class
- ④ Anonymous class.

## Member / Nested class:

→ Non-static inner class is called member / Nested class.

→ This class cannot access outside outer class.  
→ An object of this class is created within outer class but not outside the class.

Q) what are the member class ?

→ A class declared inside a class without static modifier is called member class.

→ Member classes are just like any other member methods or member variables.

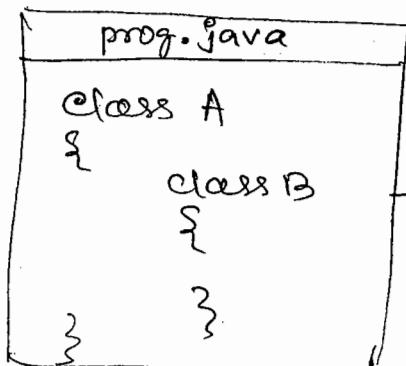
## Modifiers used with Top-level class

- ① public
- ② default
- ③ abstract
- ④ final

ex

```
public class A {} ✓
private class A {} X
class A {} ✓
protected class A {} X
```

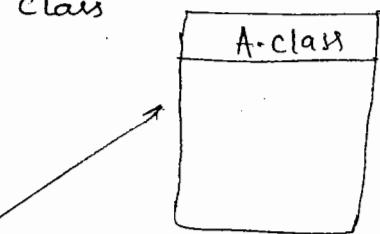
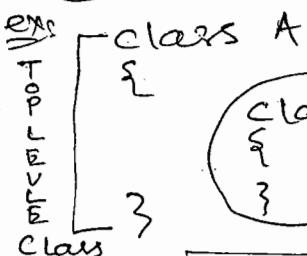
example:



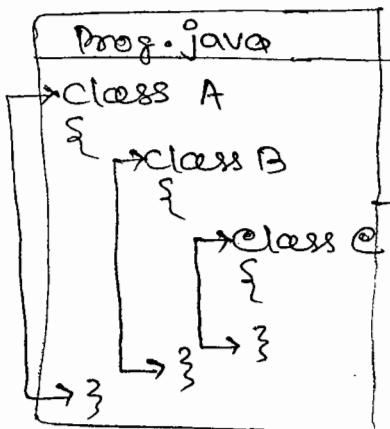
## Modifiers used with inner class

- ① private
- ② protected
- ③ default
- ④ public
- ⑤ abstract
- ⑥ final
- ⑦ static

ex



{ innerclass always comes with Toplevel class by (\$) sign }



`A.B obj1 = new A.B(); → X C.R.E`

here B class is not static, so we cannot create object outside the class.

```
Class A
{
    int x;
    Class B
    {
        int y;
    }
}
```

```

→ Class A
{
    int x;
}
Class B
{
    int y;
}
B obj = new B();
// → No error bcoz object
      is created within the
      Top level class.
}

```

→ Members of outer class can be access by inner class directly.

Ex :-

```

Class A
{
    int x;
}
Class B
{
    void m1()
    {
        System.out.println(x);
    }
}

```

→ Inner class members cannot access by outer class members directly.

Ex :-

```

Class A
{
    Class B
    {
        int x;
        void m1()
        {
            System.out.println(x);
        }
    }
}

```

\* we can use the  
members by  
creating object

Ex :-

```

import java.util.*;
class Person
{
    private String name;
    Scanner scan = new Scanner(System.in);
}
class Address
{
    private String street;
    private String city;
}

```

```
void readAddress()
{
    S.O.P("Input Street");
    Street = scan.next();
    S.O.P("Input City");
    City = scan.next();
}
```

```
void pointAddress()
```

```
{
    S.O.P("Street" + Street);
    S.O.P("City" + City);
}
```

```
Address add1 = new Address();
Address add2 = new Address();
```

```
void readDetails()
```

```
{
    S.O.P("Input name");
    name = scan.next();
    S.O.P("Input Address1");
    add1.readAddress();
    S.O.P("Input Address2");
    add2.readAddress();
}
```

```
void pointDetails()
```

```
{
    S.O.P("Name" + name);
    S.O.P("Address 1");
    add1.pointAddress();
    S.O.P("Address 2");
    add2.pointAddress();
}
```

```
Class InnerDemo1
```

```
{
    public static void main(String args[])
    {
        Person person1 = new Person();
        person1.readDetails();
        person1.pointDetails();
    }
}
```

5/11/12

Ex  
of  
C  
L

## Nested Top Level class :-

A class declared inside a class with static modifier is called "nested top level class".

Q) What is Nested Top Level class?

→ A class declared inside a class with static modifier is called nested Top Level class.

→ Any class outside the declaring class can access the nested top level class. Top-level inner classes have access to static variable only.

## Syntax :-

Class OuterClass-typeName

{

    → static class -> InnerClass-typeName

{

    fields;

    constants;  
    methods;

}

nested  
top  
level  
class

Q) What is the diff. between member class & Top level class?

### Member class

1) It is a non-static inner class.

2) This class can access static & non-static members of outer class.

3) This class cannot access outside outer class.

### Nested top level class

1) It is a static inner class.

2) This class can access static members of outer class.

3) This class can access outside the outer class.

Ex:-

```
Class A
{
    static class B
    {
        void m1()
        {
            System.out.println("m1");
        }
    }
}
```

Class InnerDemo2

```
{
    public static void main (String args[])
    {
        A.B obj1 = new A.B(); // B is static so
        obj1.m1();           bind with class
                            name.
    }
}
```

Ex:-

```
import java.util.*;  
class Complex  
{  
    private float real;  
    private float img;  
    private Scanner scan = new Scanner(System.in)  
    void read()  
{  
        System.out.print("Input method real");  
        real = scan.nextFloat();  
        System.out.print("Input img");  
        img = scan.nextFloat();  
    }  
    void print()  
{  
        System.out.println("Real" + real);  
        System.out.println("img" + img);  
    }  
}
```

Static class ComplexOperations

```
Static class @  
Static Complex addComplex(Complex c1, Complex c2)  
{  
    Complex c3 = new Complex();  
    c3.real = c1.real + c2.real;  
    c3.img = c1.img + c2.img;  
    return c3;  
}
```

Class InnerDemo3

```
PSVM(String args[]){  
    Complex comp1 = new Complex();  
    Complex comp2 = new Complex();  
    comp1.read();  
    comp2.read();  
    comp1.print();  
    comp2.print();  
}
```

Ex:-

class A

{

int n=10;

static class B

{

void m1()

{

sop(x);

}

}

}

CE

Explanation:-

Above program display compile time error,  
beoz nested top level class cannot access  
non-static members of outer class.

Local Class :-

Q) What is local inner classes?

Local inner classes are classes declared  
inside a block of code. They are visible  
only within the block of their declaration.

{

static

{

void m1()

{

}

}

}

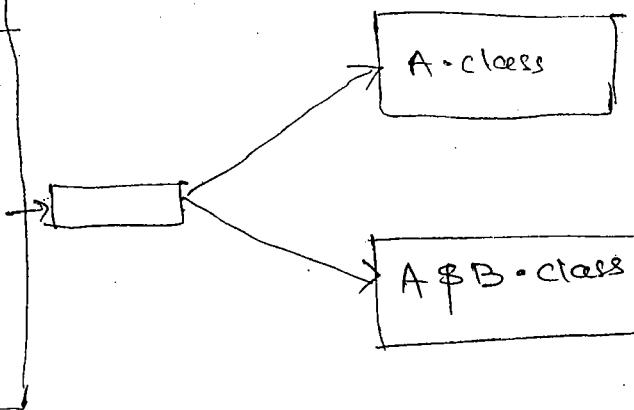
→ A class declared inside only one of above  
blocks are called local classes.

### Prog. Java

```
class A  
{ void m1()
```

```
{ class B  
{
```

```
} }
```



- local class can access the members of outer class directly.
- local class cannot access the local variables of method but can access local final variables (like constants).

Ex:-

Class A

```
{ int x=10; (instance)
```

```
void m1()
```

```
{ int y=20; (local)
```

```
final int z=30; // local final
```

Class B

```
{ void m2()
```

```
{ S.O.P(x); ✓
```

```
S.O.P(y); ✗
```

```
S.O.P(z); ✓
```

```
}
```

```
{ }
```

```
}
```

- due to scope of local variable with in the method. if it goes to any other method then that value can be change.

Ex:-

```
Interface I1
{
    void m1();
}

Class C1
{
    void m2()
    {
        Class C2 implements I1
        {
            public void m1()
            {
                System.out.println("m1");
            }

            C2 obj = new C2();
            obj.m1();
        }
    }
}

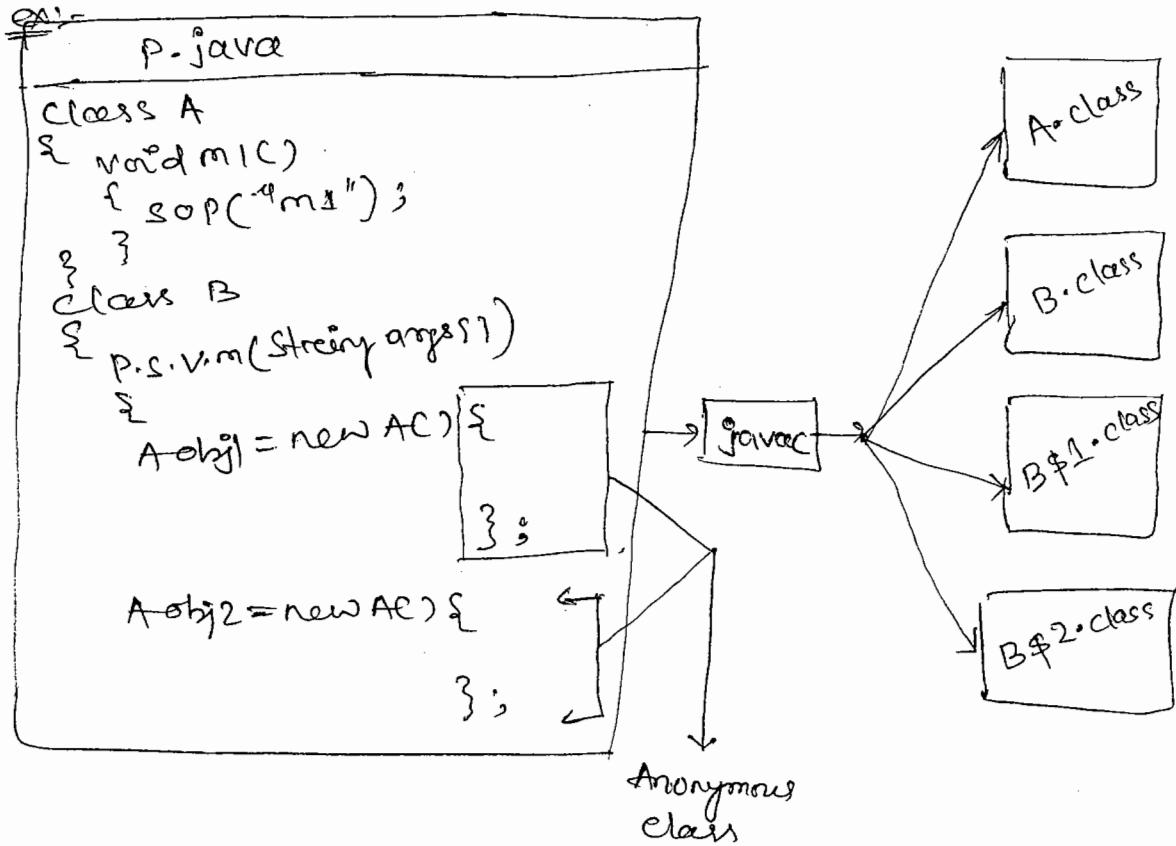
Class InnerDemo
{
    public static void main(String args[])
    {
        C1 obj1 = new C1();
        obj1.m2();
    }
}
```

→ Abstract class & Interface have no object, by  
extend & implement the object which are created  
are object of concrete class.

## Anonymous Class:

6/11/12

- A class which does not have any name is called anonymous class.
- Anonymous class is a subclass which extends a class or implements interface.
- Q) What is anonymous class?
  - Anonymous class is a type of inner class that don't have any name.
- Q) Can we declare an anonymous class as both extending a class and implementing an interface?
  - No, An anonymous class can extends a class or implement an interface, but it cannot be declared to do both.
- This class can be local class or member class.
- This class whose implementation varies from one object to another object is defined using anonymous class.



example : (Complement Interface)

interface A

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

class AnonymousDemo1

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

```
{ A obj1 = new A(); // interface can be  
// implemented but not  
// instantiated.
```

```
A obj1 = new A(); }
```

```
public void m1()
```

```
{ S.O.P("Impl of Obj1"); }
```

```
public void m2()
```

```
{ S.O.P("Impl of Obj1"); }
```

```
}
```

```
}; // anonymous class
```

```
A obj2 = new A(); }
```

```
public void m1()
```

```
{ S.O.P("Impl of Obj2"); }
```

```
public void m2()
```

```
{ S.O.P("Impl of Obj2"); }
```

```
}; // anonymous class
```

```
obj1.m1();
```

```
obj1.m2();
```

```
obj2.m1();
```

```
obj2.m2();
```

```
}
```

```
OP:
```

Impl of Obj1

Impl of Obj1

Impl of Obj2

Impl of Obj2

> java AnonymousDemo1 \$ 1

> java AnonymousDemo1 \$ 2

Compiled from "AnonymousDemo1.java"

Final version.

Therefore:  
one  
one  
statement.  
so

Ex :- (Extends a class)

abstract class A

```
{
    void m1()
    {
        S.O.P("m1 of A");
    }
}
```

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

Class AnonymousDemo2

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

```
{
    // A obj1 = new AC(); → abstract class can be extended  

    // but not be instantiated.
    A obj1 = new AC()
}
```

```
void m2()
```

```
{ S.O.P("Impl of obj1"); }
```

```
A obj2 = new AC()
```

```
void m1()
```

```
{ S.O.P("Impl of obj2"); }
```

```
void m2()
```

```
{ S.O.P("Impl of obj2"); }
```

```
}
```

```
obj1.m1();
```

```
obj1.m2();
```

```
obj2.m2();
```

```
obj2.m2();
```

```
}
```

Ex :-

Class A

```
{
    void m1()
    {
        S.O.P("m1");
    }
}
```

Class AnonymousDemo3

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

```
{ A obj1 = new AC();
```

```
obj1.m1();
```

```

A obj2 = new A();
{
    void m2()
    {
        S.O.P("m2 of obj2");
    }
    Obj2.m1();
}
//obj2.m2(); → CE
}

```

- Anonymous class can have its own methods, which cannot bind with super class reference.
- These methods can be called with the help of Super class overriding methods.

Ex:-

```

class A
{
    void m1()
    {
        S.O.P("m1 of A");
    }
}

class AnonymousDemo4
{
    public static void main (String args[])
    {
        A obj1 = new A();
        {
            void m2()
            {
                S.O.P("Anonymous class method");
            }
            void m1()
            {
                super.m1();
                m2();
            }
        };
        obj1.m1();
    }
}

```

Output

m1 of A  
Anonymous class method

- Generally we always override super class methods in anonymous class.

## Final classes and final methods :-

### Final

#### Final methods :-

- A method of class can be defined as final.
- Final method is not overridden.
- Final method is prevented from overriding by declaring that method as final.
- A method which has final keyword in its declaration is called final method.

#### Syntax :-

final return-type method-name (Parameters)

{

Statements;

}

Ex:- 1. final void m1()

{  
    System.out.println("m1");

}

→ Combination is not allowed.

2. abstract final void m1(); // error

3. static final void m1() {} // valid

4. final void m1() {} // valid.

#### Ex:-

import java.util.\*;

abstract class Shape

{  
    float dim1;

    float dim2;

    Scanner scan = new Scanner(System.in);

    final void readDim()

{  
    System.out.println("Enter two dim");

    dim1 = scan.nextFloat();

    dim2 = scan.nextFloat();

}

abstract float findArea();

}

Class Triangle extends Shape

```
{  
    float findArea()  
{  
    }  
    }  
    referen 0.5f * dim1 * dim2;  
}
```

Class Final Demo

```
{  
    public static void main (String args[])  
{  
        Triangle t1 = new Triangle();  
        t1.readDim();  
        float area1 = t1.findArea();  
        S.O.P ("Area of triangle is " + area1);  
    }  
}
```

Final Class :-

→ A class can be defined as final if keyword is called

Final class.

→ These class cannot be inherited.

→ A class can prevented from inheriting by declaring it as final.

Syntax :-

```
final class class-type-name  
{  
    variables;  
    methods;  
}
```

Ex:-

```
final class A  
{  
    void m1()  
    {  
        S.O.P ("m1 of A");  
    }  
}
```

Class B extends A

```
{  
}
```

```
}
```

O/P CE Cannot

→ The above program displays CE becoz class A is final which cannot be inherited.

# Package

Date - 7/11/12

## Packages:-

### Q.What are Packages?

→ A package is a grouping of related types providing access protection and namespace management.

Note:-

→ That types refer to classes, interface, enumeration, and annotation types.

→ Types are often referred to simply as classes and interfaces, since enumeration & annotation types are special types kind of classes & interfaces respectively. So types are often referred to simply as classes & interface.

→ Package is a directory which contain "class" files, these class file cannot access outside the package without the package name.

→ Package may be predefined or user-defined.  
It provides encapsulation.

## Advantage of Packages:-

→ You and other programmer can easily determine that these classes and interfaces are related.  
i.e Simplicity. (Simplicity)

Ex:-

You and other programmers know where to find classes and interfaces that can provide graphics-related functions.

→ The names of your classes and interfaces won't conflict with the names in other package bcoz the package create the new namespace.  
i.e It provides namespace management.

→ You can allow classes within the package to have unrestricted access to one another yet still restrict the access for types outside the package.  
i.e it provides access protection (security).

### Syntax for Package:

```
package package_name;  
classes;  
interfaces;  
enums;
```

→ Package statement must be first statement within source program.

→ One source program allows only one package statement.

→ Contents of the package can be default or public. (becoz they contain top-level class)

→ Default members of package cannot access outside the package.

→ Public members ~~can~~ <sup>can</sup> access outside the package.

ex:-

```
package package1;  
class A  
{  
    void m1()  
    {  
        sop("inside m1 of class A of pck1");  
    }  
}
```

→ Save with any name as class is default.

→ We can save the package with any name entell  
and unless the ~~or~~ class is not public access specifier.

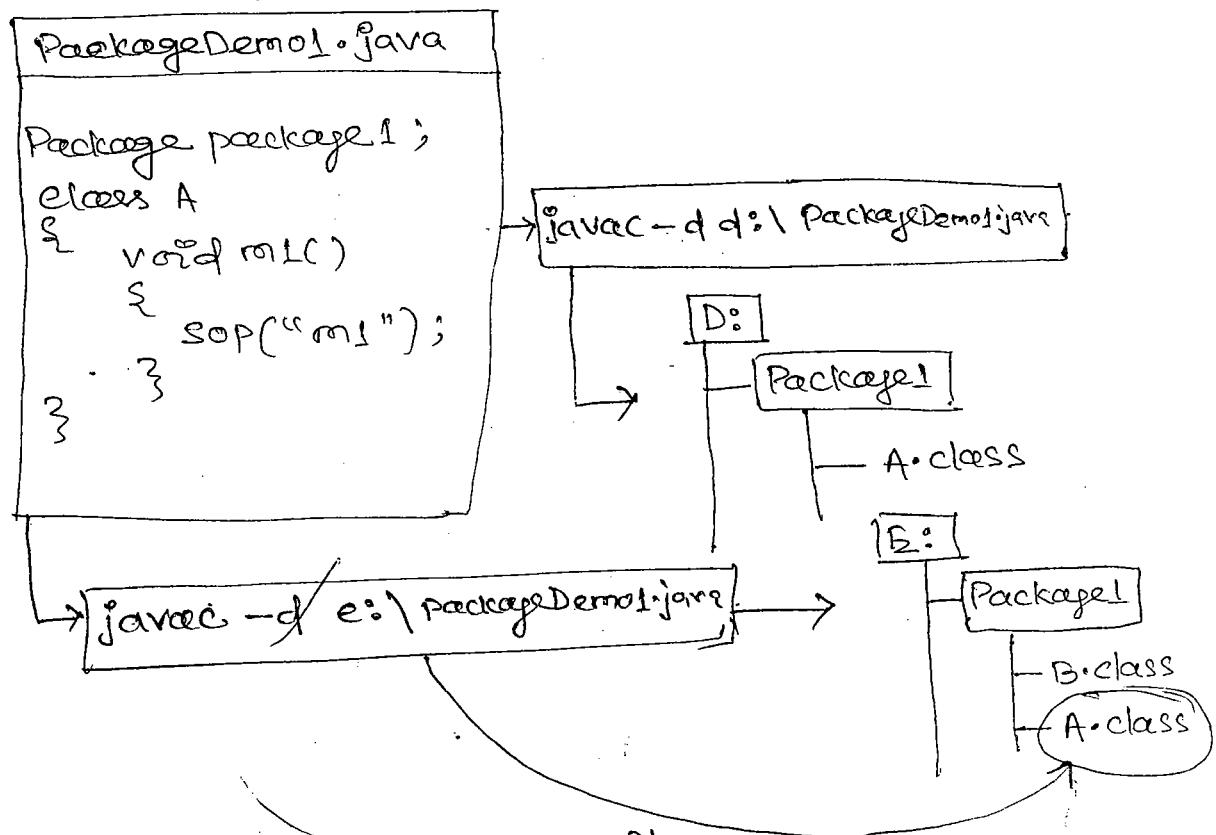
### Compiling Package :-

Syntax :-

```
javac -d <location> {SourceName}
```

Ex:-  
javac -d c:\ PackageDemo1.java  
javac -d c:\ net PackageDemo1.java

- On compiling package, java compiler creates a directory with package name in which it generates .class files.
- These .class files are generated with reference of package name.



It just add the  
class A in the existing  
package1 of e drive  
which previously contain  
B.class.

## Using classes from other Packages:-

- To use a public package member (classes or interfaces) from outside its package, we must do one of the following
  - Import the package statement using import Statement.
  - Import the member's entire package using import statement.
  - Refer to the member by its fully qualified name (without using import statement)

Ex:- (Adding new class to existing package)  
package package1;

public class B

```
{ void m1()
    {
        System.out.println("Inside m1");
    }
}
```

## Syntax of Import Statement:-

1. `import package-name.member-name;`
2. `import package-name.*;`

2. \* → Import the member's entire package  
i.e access can be used all members.

1. → Import package member.

→ Importing is not including the class but  
adding fully qualified name to the class.

→ Loading of classes is done during runtime.

Ex:- `import package1.B;`

Class PackageDemo2

```
{ public static void main(String args[])
    {
        B obj1 = new B();
    }
}
```

## Q18 CB

→ The above program display compilation error "package package1 does not exist".

Compiler always search for classes and packages inside working directory if not available it Search in given "classpath".

Class path:-

→ Class path is environment variable used by Java Compiler and JVM for locating packages and classes.

Syntax:-

`Classpath = <location1>;<location2>;<location3>;`

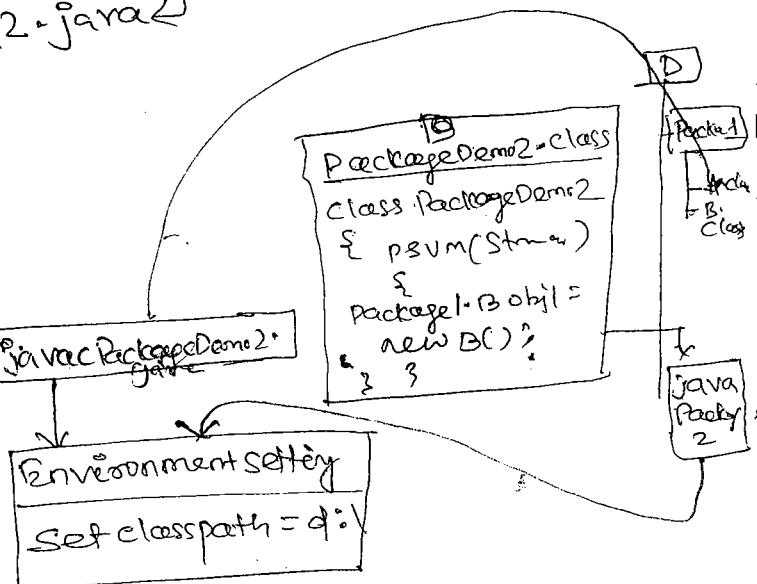
ex:- `Classpath = d:\; // Valid`

`Classpath = d:\ package1; // Invalid  
(Should not give the)  
package name`

➤ Set classpath = d:\;

➤ javac PackageDemo2.java

C:\  
+-- batchdam  
    +-- PackageDemo2.java  
        import Package1.B;  
        class Package2  
        {  
            psvm(String args)  
            {  
                B obj1 = new B();  
            }  
        }



Date - 8/11/12

Using classes outside package without using import statement :-

→ This is done using fully qualified name

↓  
(Class name along with package name)

Ex :-  
Ex (package1.B)

Class PackageDemo3  
{ public static void main (String args[] )  
{ package1.B obj1 = new package1.B();  
}}

DOS/Windows :-

Set classpath = C:\init;  
Set classpath = %classpath%; d:\ → C:\init; d:\  
refers to existing current path  
class path. new concatenation

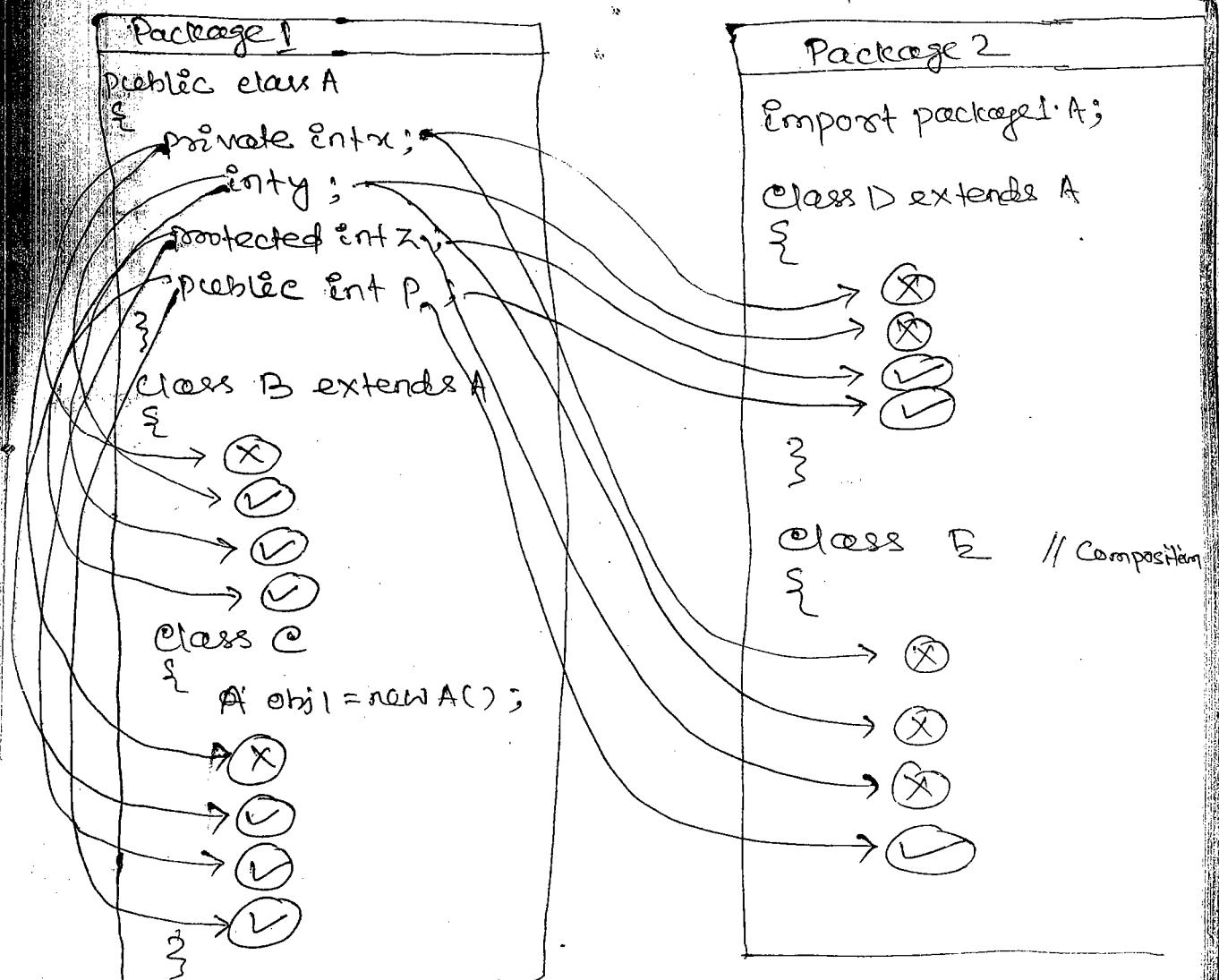
Linux/Unix :-

export classpath = location1, location2.

Members Access :-

→ Members of a class can be private, public,  
protected & default. → (Package Scope)

| SCOPE<br>Access Specifier | Within the Class | Within NonSubclass of same package | Within Subclass of same package | Within non subclass Subclass of outside package | Within outside package |
|---------------------------|------------------|------------------------------------|---------------------------------|-------------------------------------------------|------------------------|
| Private                   | Yes              | No                                 | No                              | No                                              | No                     |
| default                   | Yes              | Yes                                | Yes                             | No                                              | No                     |
| protected                 | Yes              | Yes                                | Yes                             | No                                              | Yes                    |
| public                    | Yes              | Yes                                | Yes                             | Yes                                             | Yes                    |



Ex:-

```

package package2;
public class Class1
{
    protected void m1()
    {
        sop("inside protected method");
    }
    public void m2()
    {
        sop("inside public method");
    }
}

```

ex:-

```

package package3;
import package2.class1;
import package1.B;
public class Class2
{
    Class1 obj = new Class1();
    public void m2()
    {
        obj.m1(); // error → becoz m1() is protected
        obj.m2(); // in package2 class1
    }
}

```

(within non subclass outside package).

ex:-

```

package package3;
import package2.class1;
public class Class3 extends Class1
{
    public void m3()
}

```

```

    {
        m1(); // within the subclass outside the
        m2(); // package. (protected)
    }
}

```

m3(); // public can accessible by all

ex:-

```

package package3;
import package1.class1;
public class Class4 extends Class1
{
    public void m4()
}

```

```

    {
        System.out.println("inside m4 methods");
    }
}

```

ex:-

```

import package3.Class4;

```

```

class PackageDemo4
{
    public static void main(String args[])
}

```

```

    {
        Class4 obj = new Class4();
    }
}

```

//obj.m1(); //E<sup>P</sup>rotected cannot access outside  
//obj2.m2(); ✓ the package of non subclass

Q) Can we use both import & fully qualified name?

→ Yes

→ If a class is available in two packages with same name, and while using it another class, if that class adds both import statements, Compiler throws ambiguity error in accessing that class. To solve this CR we must access it with fully qualified name.

Ex:- package P1;

public class X

{ public void m1()

{ SOP("X of m1");

Package P2;

public class X

{ public X()

{ SOP("P2.X constructor");

Package P3;

import P1.\*;

import P3.\*;

public class Test

{ SVM(String args[])

{ X x = new X();

P1.X x = new P1.X();

P2.X x = new P2.X();

}

09/11/12

Siebpackages :-

→ A package within package is called Sub package.

### Syntax :-

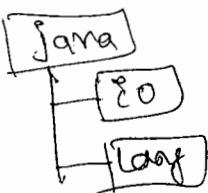
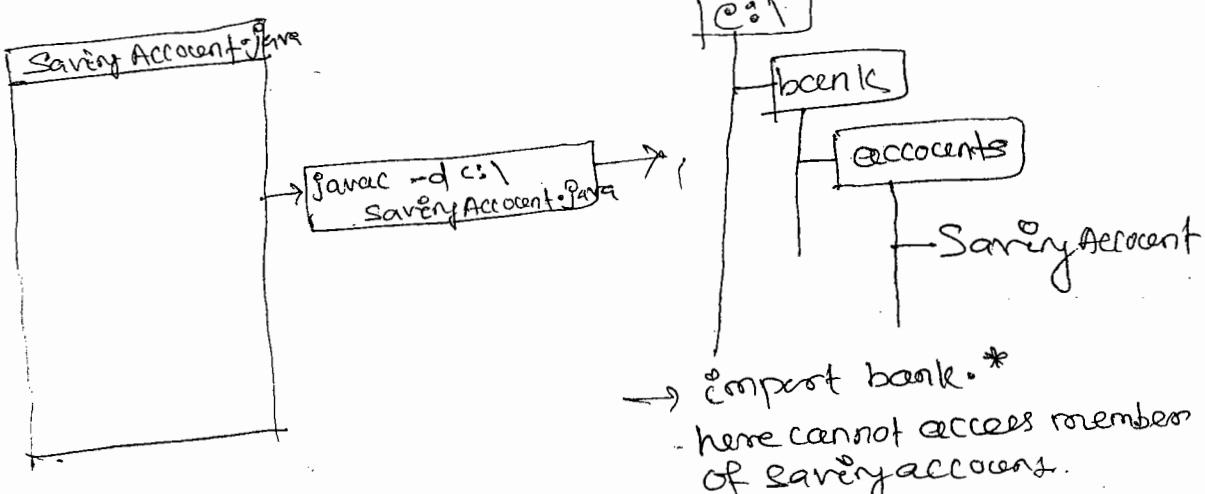
package package-name • sub-package1 • sub-package2 ... ;  
(OR)

~~Ex?~~

Ex-1  
package bank.accents;  
public class SavingAcccent  
{  
    ...  
}

public void m1()

3. Inside Safety Account of accounts  
SOP ("Inside Safety Account of accounts  
package");



`import java.*;` → cannot access the  
class `*`

earliest Java. 80.\* → here we can access  
the classes present  
inside it directly.

→ "import" is used by java compiler to locate by JVM, becoz import is present classes but not by outside the class.

Importing / Using members of Subpackage :-

- Members of subpackage is used along with main package name & sub package-name.
- By importing main package / root package we cannot access members of subpackage (above example).

Ex:-

```
import bank.account.SavingAccount;
```

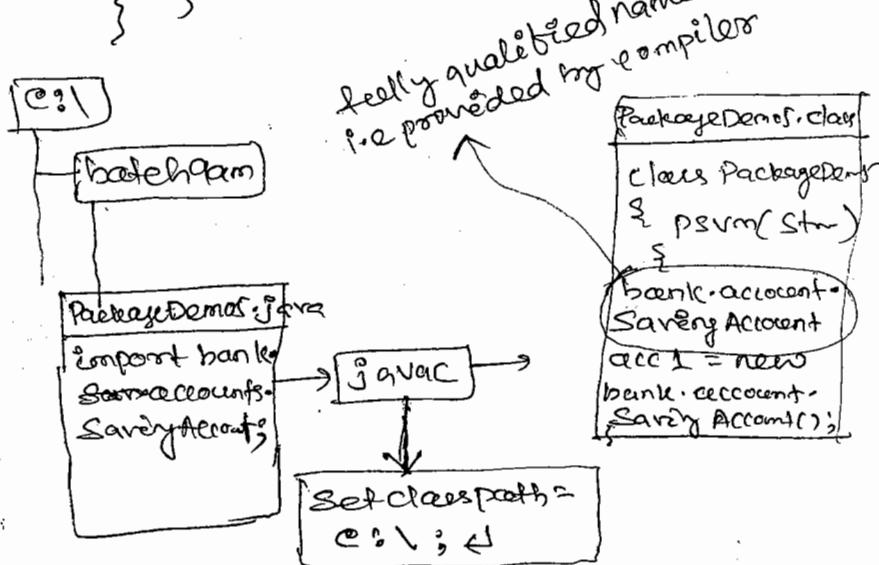
```
class PackageDemo5
```

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

```
{ }
```

```
    SavingAccount acc1 = new SavingAccount();
```

```
} } acc1.m1();
```



> del SavingAccount;

Static Import :-

mmmmmm This is a new feature added in Java 5.0

→ This is used to import static data

→ This is used to import static members of classes outside the package.

Syntax :-

```
import static package-name . class-name . static-data-members;
```

(OR)

```
import static package-name . class-name *;
```

→ It is useful to access more than one static member of the class.

Ex:-

```

package package4;
public class class1
{
    public static int x = 10;
    public static int y = 20;
}

```

Ex:-

// before S.O  
import package4.class1;

class PackageDemo6

{ public static void main (String args[])

{
 System.out.println(class1.x);
 System.out.println(class1.y);
 System.out.println(class1.x);
 }

// after S.O concerns

import static package4.class1.x;

import static package4.class1.y;

class PackageDemo7

{ public static void main (String args[])

{
 System.out.println(x); // 10
 System.out.println(y); // 20
 }

.jar → java archive file.

→ Compressed all related files into one file.

.war → web archive file

.ear → Enterprise archive file

→

→

b) 1

1

2.

→

3

4

→ 5

ce

)

→ 6

## java Utility / command :-

Date - 10/11/12

→ This utility is used for compressed file which is called as "Zipfile".

→ There are 3 types of Zipfile created in Java application environment development.

- 1) .jar (Java archive file)
- 2) .war (Web archive file)
- 3) .ear (Enterprise archive file)

### How to create jar file :-

#### Syntax :-

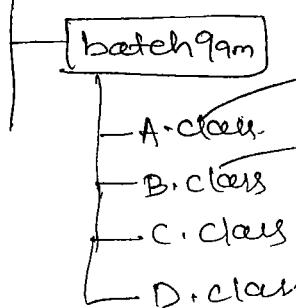
1. jar -cvf jar-file-name include-files

- c → Create
- v → Verbose
- f → filename

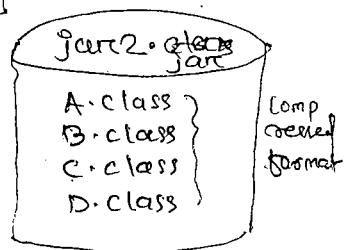
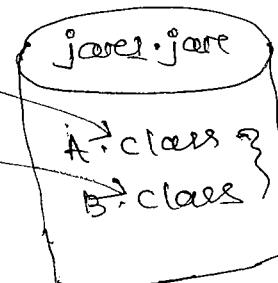
2. jar -cf jar1.jar A.class B.class

→ The above command creates jar file with A.class & B.class.

File:



> jar -cf jar1.jar A.class B.class



> jar -cf jar2.jar \*.class

→ If we want to see output then we write -

> jar -cvf jar1.jar \*.class ↳

→ Here we see all output.

In → actual class size.

Out → Compressed class file size.

> jar -cvf jarfile package1 package2 ↳

→ The contents of jar is set by setting the classpath :

classpath = C:\batch9am\jar2.jar ; C:\batch9am\jar1.jar

> else ↳

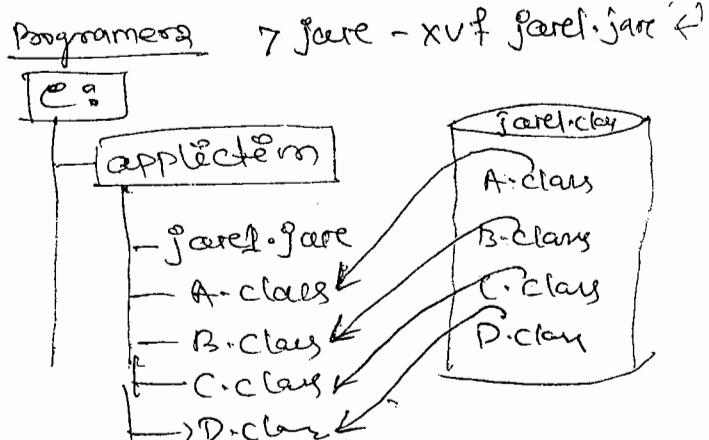
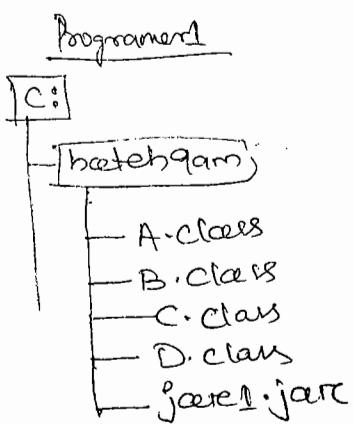
Extracting Content of jarfile: —

jarx -xvf jarfile-name

-x → extract files from jar

-v → verbose

-f → filename



Displaying Contents of jar file: —

{ jar -tvf jarfile-name ↳

{ -t → table of Contents ↳

Upgrading jar file: —

jar -uvf jarfile-name & include files ↳

jar -uvf jarfile-name & update existing jar.

-u → updating existing jar.

## Creating executable jar file :-

### Step-1 :-

- Create a manifest file.
- manifest file having information about
- It is a text file. It is have extention ".mf".
- manifest file contain 2 attributes —
  1. Version Manifest Version:
  2. Main-class: class name which contain main method.

### Step-2 : (Creating an executable jar)

jar -cvf from jar-file-name manifest-file-name  
 < included classes >

### Step-3 :-

If it is a GUI — (we cannot doubleclick)

java -jar jar-file-name

### example :-

```

import java.awt.*;
class Window1 extends Frame
{
    public static void main(String arg[])
    {
        Window1 w1 = new Window1();
        w1.setSize(500,500);
        w1.setVisible(true);
    }
}
  
```

Create Manifest file in notepad -

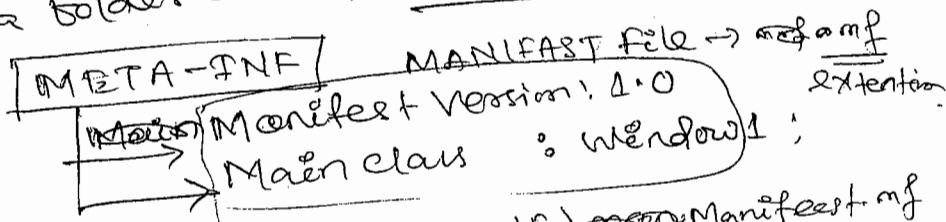
Manifest MainClass : Window1

Manifest Version : 1.0

> java -cvfm jar3.jar manifest.mf Window1.class

> When we create a jar file jar command will

create a folder called "META-INF".



> java -cvfm app2.jar META-INF \manifest.mf  
window1.class

jarfile  
name

Ctrl+Alt

> java -jar app2.jar

→ then copy files

[app]

window1.java  
window1.class

META-INF

→ MANIFEST.MF



→ If it is not execute, then change the property.

→ If it is not executable, then change the property.

→ If it is not executable, then change the property.

→ If it is not executable, then change the property.

→ If it is not executable, then change the property.

→ If it is not executable, then change the property.

# Java Strings

12/11/12

- String is a collection of characters.  
→ In java, String is represented in 3 ways -

1. String class
2. StringBuffer class
3. StringBuilder class

- String which contain number and alphabets called "alphphanumeric string".  
→ Whenever we create a string it is internally created as character array.

- String class is available at `java.lang` package and the default package imported by `java` is `java.lang` package.

## String Class:

- The String class represents a character strings. All String literals in java programs, such as "abc", are implemented as instance of this class.
- String are constant; their values cannot be changed after they are created. String buffers support mutable strings, because String objects

## Ans

### Constructors:

- `String()` :- initializes a newly created String object so that it represents an empty ch. sequence

- `String(byte[] bytes)` :- Construct a new String by decoding the specified array of bytes using the platform's default character charset.

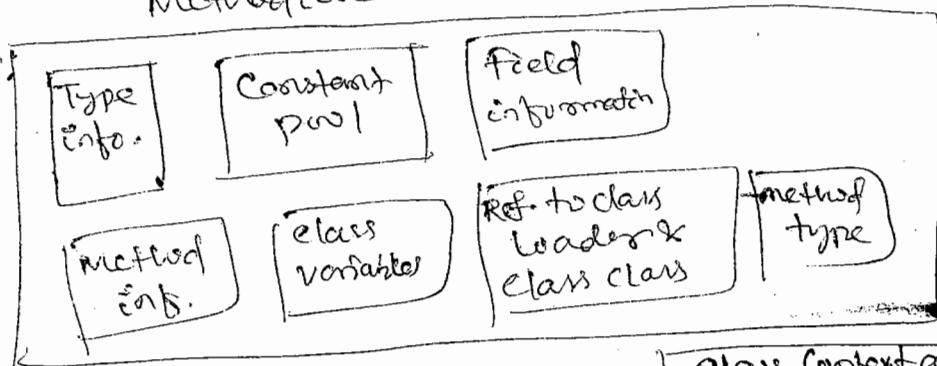
`String(char[] value)`:— Allocates a new String so that it represents the sequence of characters sequentially contained in the character array argument.

ex:-

- `10` → Integer Constant
- `1.5f` → Float Constant
- `1.5` → Double "
- `true` → Boolean "
- `'A'` → Character "
- `"java"` → String "

→ String constant is stored in `method Constant pool` of method area (class context area).

Method area



`String s1 = "java";`

`SOP(s1); → java`

`s1 = "oracle";`

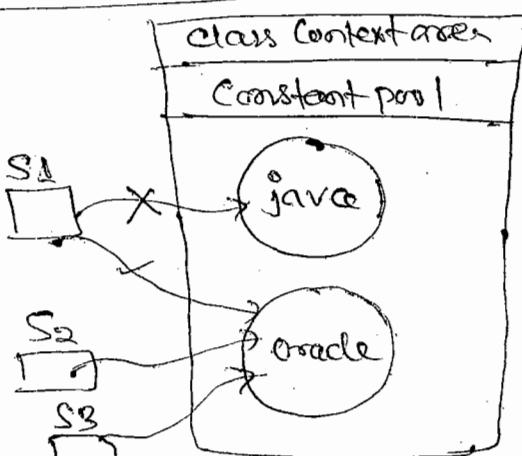
`SOP(s1); → oracle`

`String s2 = "Oracle";`

`String s3 = "Oracle";`

here it only creates reference variable but not again String constant as it previously present in the Constant pool.

class Context area



`SOP(s1);`

Compiler interprets it

it is a method of object class.

ex : // With ref override to String method )

```

Class A
{
    int x = 10;
}

```

public String toString()  
referen. A@3c28a1

```

Class B
{
    psvm()
    {
        A obj1 = new A();
        sop(obj1); → obj1.toString()
        @IP → A@3ses → (referen hashcode)
    }
}

```

- Reference variable is pointed by invoking toString() method of object class.
- to String method return textual representation of object.
- This method can be override in order to print contents of object or important information of object.

ex : (Overriding toString() method of Object class)

Class A

```

{
    int x = 10;
}
public String toString()
{
    referen String.valueof(x);
}

```

Class Demo2

```

psvm(String args)
{
    A obj = new A();
    sop(obj.x); →
    sop(obj.toString());
}

```

- valueOf() method is a method of String class which converts String value to integer value.

Ex :-

Q||

Class StringDemo1

A

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

R

    String s1 = "java";

    String s2 = "oracle";

    System.out.println(s1); → java

    System.out.println(s2); → oracle

    System.out.println(s1.toString()); → java

    System.out.println(s2.toString()); → oracle

}

→ toString() method give textual information.

→ If we want to give hashCode then we must override  
toString().

Q How many String objects are created?

Class StringDemo2

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

    String s1 = "java";

    String s2 = "java";

    String s3 = "oracle";

    String s4 = "oracle";

    System.out.println(s1);

    System.out.println(s2);

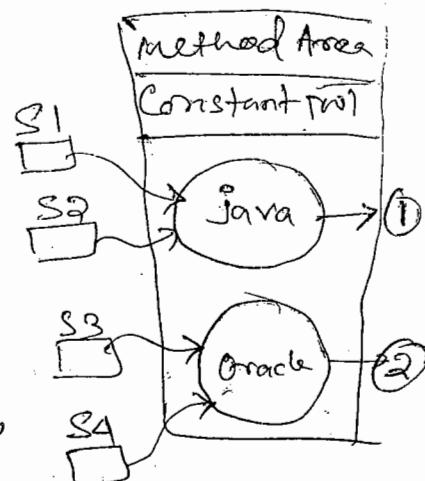
    System.out.println(s3);

    System.out.println(s4);

}

→ O/P Ans:-

→ 2 String objects are created in  
Constant pool area.



Q) How to get hashCode of the String.

Ans:- By using hashCode() method.

Ex:- Class StringDemo2

```
{ public static void main(String args[])
{
    String s1 = "java";
    String s2 = "java";
    String s3 = "Oracle";
    String s4 = "oracle";
    System.out.println(s1);
    System.out.println(s2);
    System.out.println(s3);
    System.out.println(s4);
    System.out.println(s1.hashCode());
    System.out.println(s2.hashCode());
    System.out.println(s3.hashCode());
    System.out.println(s4.hashCode());
}}
```

- hashCode() is a method of Object class, which refers hashCode of current object.
- hashCode is a 32 bit integer.

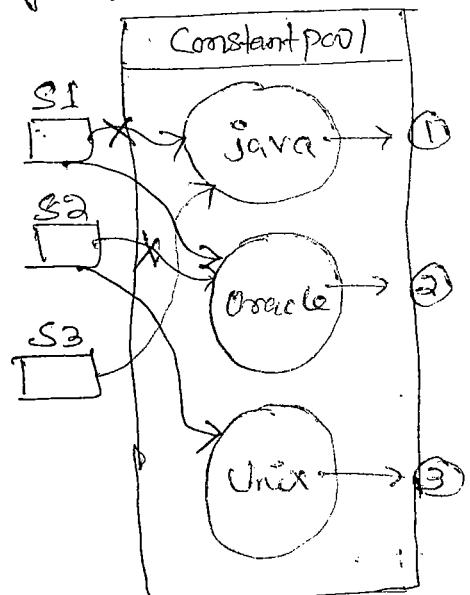
Q) How many String Objects are created?

Ex:- Class StringDemo3

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

```

    String s1 = "java";
    String s2 = "Oracle";
    String s3 = "java";
    s1 = "Oracle";
    s2 = "Unix";
    System.out.println(s1); → Oracle
    System.out.println(s2); → Unix
    System.out.println(s3); → java
    System.out.println(s1.hashCode());
    System.out.println(s2.hashCode());
    System.out.println(s3.hashCode());
}}
```



Comparing Strings :-

→ Comparing of String are done using —

1. equals()
2. equalsIgnoreCase()
3. compareTo()
4. compareToIgnoreCase()
5. == operator

equals() :-

\* boolean equals (Object an Object) { }

→ Compares this String to the specified object.

→ equals() is a method of Object class and overridden in String class in order to compare contents.

Ex:- important java.util.\*;

class Login

{ public void main (String args[])

{ Scanner scan = new Scanner (System.in);

System.out.println ("Input Username");

String cname = scan.next();

System.out.println ("Input password");

String password = scan.next();

String password = scan.next();

if (cname.equals ("PRITAM") &&

password.equals ("PRITAM"))

{ System.out.println ("Welcome to my application");

}

else

{ System.out.println ("Invalid username & password");

}

}

→ equals() method is case sensitive.

## (2) equalsIgnoreCase() :-

\* boolean equalsIgnoreCase (String anotherString)

→ Compares this string to another string, ignoring case consideration. (i.e. case insensitive)

Ex 2

class StringDemo2

{ public static void main (String args[])

{

String s1 = "nit";

String s2 = "Nit";

String s3 = "NIT";

boolean b1 = s1.equals(s2);

boolean b2 = s1.equals(s3);

boolean b3 = s1.equalsIgnoreCase(s3);

s.o.p(b1); → True

s.o.p(b2); → False

s.o.p(b3); → True

}

}

## (3) CompareTo() :-

\* int compareTo (String anotherString)

→ Compares two strings lexicographically.

→ This method has a return type which reflects —

0 → if String1 == String2

>0 → if String1 > String2

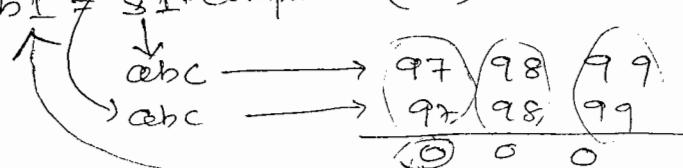
<0 → if String1 < String2

Ex 1 String s1 = "abc"

String s2 = "abc"

int b1 = s1.compareTo(s2);

sop(b1); → 0



Ex-2 String s1 = "abc";  
String s2 = "aBc";

int b1 = s1.compareTo(s2);

|     |   |    |    |     |
|-----|---|----|----|-----|
| abc | → | 97 | 98 | 99* |
| aBc | → | 97 | 66 | 99  |
|     |   | 0  | 32 |     |

SOP(b1); → 32

→ If first character is same then goto 2nd character  
otherwise not.

Ex-3 :-

String s1 = "Abc";

String s2 = "aBc";

int b2 = s1.compareTo(s2);

|     |   |    |    |     |
|-----|---|----|----|-----|
| Abc | → | 65 | 98 | 99* |
| aBc | → | 97 | 98 | 99  |
|     |   | 32 |    |     |

SOP(b2); → -32.

(4) CompareToIgnoreCase() :-

int compareToIgnoreCase(String str)

→ Compare to strings lexicographically, ignoring  
case difference.

(5) == operator :-

This operator used for comparing object

references (ref) not reference Compare-

→ It compares hashcode.

Ex :-

class StringDemo3

{ public static void main (String args[])

{ String s1 = "abc";

String s2 = "abc";

```

if(s1 == s2)
    sop("equal");
else
    sop("not equal");

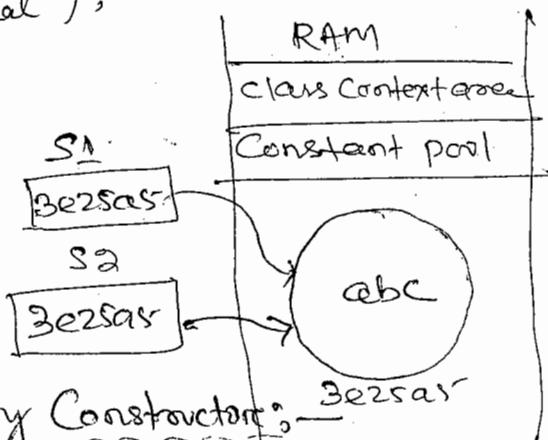
```

3 3

```

if(s1 == s2)
    ↴
    Be2sas
    ↴
    Be2sas

```



Creating a String using Constructor:

Class String Demo4

```

public static void main (String args[])

```

```

{
    String s1 = new String ("abc");
    String s2 = new String ("abc");
    if (s1 == s2)
        sop ("equal");
    else
        sop ("not equal");
}

```

3 3

if(s1 == s2)

19821f  
not equal.

But

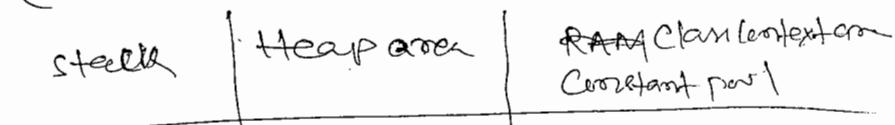
if(s1.equals(s2))

↳ (equal)  
↳ String.equals

Value of method:

- It is a static method of String class.
- This method overrides String.equals of any of value.
- This method is overloaded.

RAM



s1  
19821f

s2  
add1

19821f

add1

RAM Class Context Area  
Constant pool

abc

Be2sas

Converts int to  
String

`String.valueof(int);`  
`String.valueof(short)`  
`String.valueof(float)`  
`String.valueof(double)`

## Concatenation methods :-

- Concatination means combine to string.
- It uses 2 methods.

### 1. String concat (String str) :-

- Concatenates the specified string to the end of this string.

### 2. + :-

- It combines two strings. Concatenates the specified string to the end of this string.

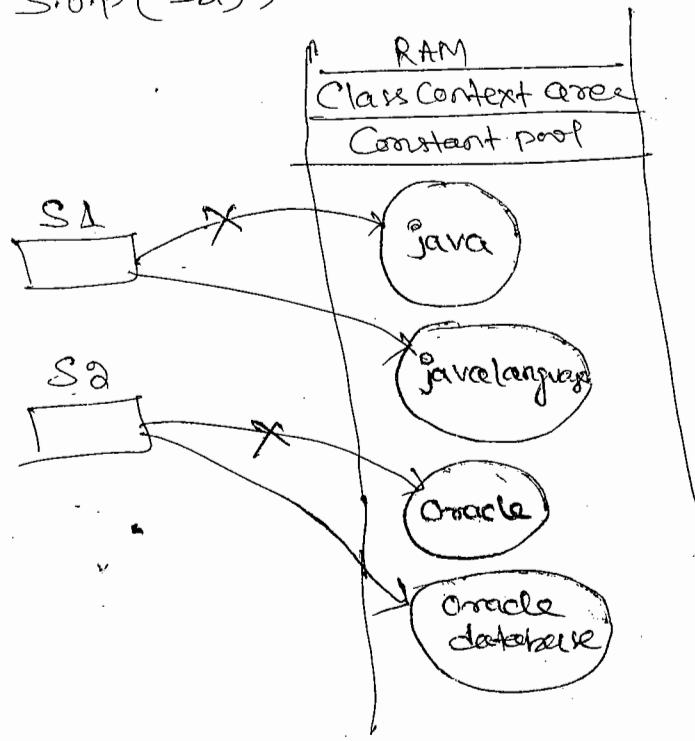
ex :-

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

```
{ String s1 = "java";
    s1 = s1.concat ("language");
    System.out.println (s1); }
```

```
String s2 = "oracle";
    s2 = s2 + "database";
    System.out.println (s2); }
```

?



String Java Platform

Extracting Character or String from a String:

Char CharAt (int index)

→ Refers the char value at specified index.

Q. WAP to read string and Count no. of vowels.

import java.util.\*;

Class StringDemo5

{ public static void main (String args[])

{ int count = 0;

Scanner scan = new Scanner (System.in);

SOP ("Input any String");

String str = scan.next();

for (int i=0; i<str.length(); i++)

{ switch (str.charAt(i))

{ Case 'a':

Case 'e':

Case 'i':

Case 'o':

Case 'u':

Count = count + 1;

} }

SOP ("Count of vowels:" + Count);

String substrng (int beginIndex)

→ Refers a new string that is a substring of this string.

String substrng (int beginIndex, int endIndex)

→ Refers a new string that is a substring of this string.

Ex:

Class StringDemo6

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

{ String s1 = "Rama Rao";

String s2 = s1.substring(5);

SOP(s2); → Rao

String s3 = "Object Oriented Programming";

String s4 = s3.substring(7, 15);

\* begining index included but ending index  
not included. It just extract 7 to 14.

\*/

SOP(s3);

SOP(s4); → Oriented

}

Conversion methods:

String toLowerCase():

→ Converts all of the characters in the String  
to lower case using the rules of the default  
locale.

String toUpperCase():

Converts all of the characters in the String  
to uppercase using the rules of the default  
locale.

Locale → It is object in Java, which defines language.  
The default locale is English.

Ex:

Class StringDemo7

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

{ String s1 = "abc";

SOP(" " + s1.toUpperCase());

SOP(" " + s1.toLowerCase());

}

}

## Replacing methods:

1. String replace (char old char, char new char)

→ Returns new string resulting from replacing all occurrences of oldchar in this String with new char.

2. String replaceFirst (String regex, String replacement)

Replace the first substring that matches the given regular expression with the given replacement.

Ex: Class StringDemo8

```
{ public static void main(String args) }
```

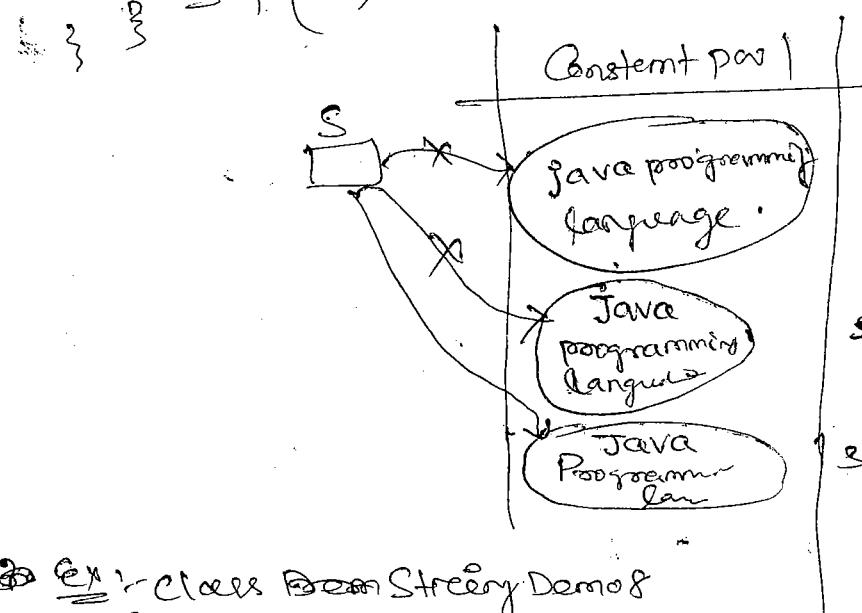
```
{ String s = "java Programming language";
```

```
s = s.replace('j', 'J');
```

```
s = s.replace('P', 'P');
```

```
s = s.replace('I', 'L');
```

```
SOP(s); → Java Programming language.
```



`s = s.replace('j', 'J');`

`s = s.replace('P', 'P');`

Ex: Class StringDemo8

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

```
{ String s = "java Programming Language"; }
```

```
s = s.replaceFirst("java", "net");
```

```
SOP(s); → net Programming language
```

## String[] split (String regex)

→ Splits this string around matches of the given regular expression.

Ex:- Class StringDemo10

```
{
    public static void main (String args[])
    {
        String s1 = "Java language";
        String s[] = s1.split("a");
        System.out.println(s.length);
        for (int i=0; i<s.length; i++)
            System.out.println(s[i]);
    }
}
```

## byte[] getBytes()

Encodes this String into a sequence of bytes using platform's default charset, storing the result into a new byte array.

Ex:- Class StringDemo11

```
{
    public static void main (String args[])
    {
        String s = "java";
        byte b[] = s.getBytes();
        System.out.println(s);
        for (int i=0; i<b.length; i++)
            System.out.println(b[i]);
    }
}
```



Converts immutable object to mutable object.

## String Buffer :

A thread-safe, mutable sequence of characters.

A String buffer is like a string, but can be modified. At any point of time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.

Q) What is the diff between String & String buffer?

### String

### String Buffer

- |                                                                  |                                         |
|------------------------------------------------------------------|-----------------------------------------|
| (i) These are immutable.                                         | (i) Mutable.                            |
| (ii) String methods are not synchronized.                        | (ii) Methods are synchronized.          |
| (iii) Not thread safe.                                           | (iii) Thread safe.                      |
| (iv) String is suitable for used in single threaded application. | (iv) Used in multithreaded application. |
| (v) It is created in constant pool of object class-context-area. | (v) Created in heap area.               |

- |                        |                                 |
|------------------------|---------------------------------|
| (vi) String is shared. | (vi) <del>not</del> not shared. |
|------------------------|---------------------------------|

→ String Buffers are safe to use multiple threads.

## Constructors:

### StringBuffer():

Constructs a string buffer with no characters in it and an initial capacity of 16 characters.

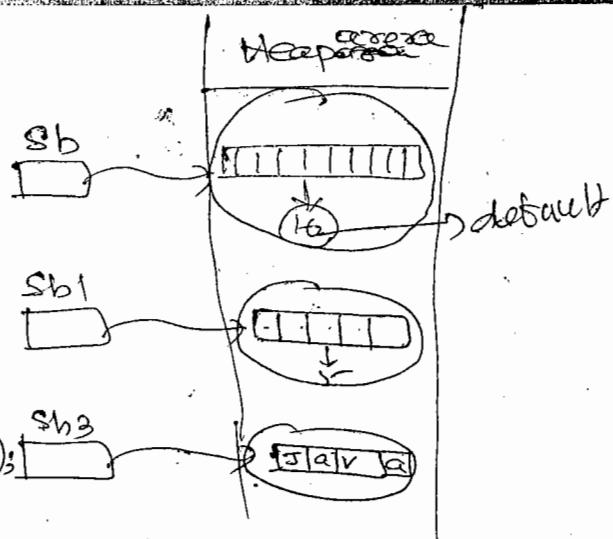
### StringBuffer(String str):

Constructs a string buffer initialized to the content of the specified string.

StringBuffer sb = new  
StringBuffer();

StringBuffer sb1 =  
new StringBuffer('s');

StringBuffer sb3 =  
new StringBuffer("java");



methods of String:—

append:—

- This method is overloaded in order to append various types of values.

append(int)  
append(float)  
append(double)  
append(char)  
append(String)

append(byte)  
append(long)  
append(char[])

insert:—

- This method provides to insert value at given position.

- This method is overloaded in order to insert various types of values.

insert(int offset, int)  
insert(int offset, float)  
insert(int offset, double)  
insert(int offset, char)  
insert(int offset, boolean)  
insert(int offset, short)  
insert(int offset, String)

Offset → index number where it has to be inserted.

## int capacity():

Retrieves the current capacity (how many characters it holds).

ex:-

### Class StringDemo12

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

{

```
StringBuffer sb = new StringBuffer();
```

```
SOP(sb.capacity());
```

```
SOP(sb);
```

```
sb.append("java");
```

```
SOP(sb);
```

```
SOP(sb.length());
```

```
SOP(sb.capacity());
```

```
sb.append("language");
```

```
SOP(sb);
```

```
sb.insert(4, "Programming");
```

```
SOP(sb);
```

```
SOP(sb.length());
```

```
SOP(sb.capacity());
```

}

}

→ These all op's are done on one object becoz  
String objects are mutable.

16/11/12

- S
- ① Irr
  - ② Ec
  - ③ m  
sy
  - ④ Ni
  - ⑤ Con  
pool  
conne
  - ⑥ Us  
Thro

Ex:-

Class StringBufferDemo3

{ public static void main (String args [ ] )

{ StringBuffer sb = new StringBuffer ("Java Language");

SOP (sb); → Java Language

sb.delete (5, 12);

SOP (sb); → java e

} }

### ③ String Builders :

~~~~~

→ A mutable sequence of characters : This class provides an API compatibility with StringBuffer, but with no guarantee of synchronization.

→ This class is designed for use as a drop-in replacement for StringBuffer in places where the StringBuffer was being used by a single thread (as is generally the case).

(4)
→
E

String

- | | | |
|---|---------------------------------------|---|
| ① Immutable | ① Mutable | ① Mutable |
| ② can be shared | ② Cannot be shared | ② Cannot be shared |
| ③ methods are not synchronized. | ③ Methods are synchronized | ③ Not synchronized |
| ④ Not thread safe | ④ Thread safe | ④ Not thread safe |
| ⑤ Created in constant pool of class context area. | ⑤ Created in heap area. | ⑤ Created in heap area. |
| ⑥ Used in single threaded app | ⑥ Used in multi-threaded application. | ⑥ Used in single-threaded applications. |

StringBuffer

StringBuilder

Ex:-

```
class StringBuildersDemo1
```

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

```
{
    StringBuilder sb = new StringBuilder("JavaLang");
    sop( sb );
    sb.delete( 5, 13 );
    sop( sb );
}
```

④ StringTokenizer:

→ This class is a member of "java.util package".

Ex:- ① "RamaRao" → one string

tokens
separator (here space is used as delimiter)

② "BB&R, Orissa"
delimiters (here ',' is delimiter).

→ String Tokenizer class allows an application to break a string into Tokens.

→ The set of delimiters (The characters that separate tokens) may be specified either at creation time or on a per-token basis.

Constructors:

1. StringTokenizer (String str)

Constructs a StringTokenizer for the specified string.

2. StringTokenizer (String str, String delim)

Constructs a StringTokenizer for the specified string.

Methods:

1. int countTokens()

Calculates the no. of tokens that this tokenizer's nextToken method can be called before it generates an exception.

2. String nextToken()

Retrieves the next token from the StringTokenizer.

3. boolean hasMoreTokens()

Tests if there are more tokens available from this tokenizer's string.

Ex:-

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

```
class StringTokenizerDemo
```

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

```
        String address = "1-2-A/3, SRT-39, SRNAGAR,  
        Hyd";
```

```
        StringTokenizer st = new StringTokenizer  
        (address, ",");
```

```

SOP( st.countTokens() );
String hno = st.nextToken();
String street = st.nextToken();
String city = st.nextToken();
System.out.printf("%s %s %s", hno, street, city);
}
}

```

Ex-2:

```

import java.util.*;
class StringTokenizerDemo2
{
    psvm(String args[])
    {
        String s = "Java Programming Language";
        StringTokenizer st = new StringTokenizer(s);
        String s1 = st.nextToken(" ");
        String s2 = st.nextToken("re");
        String s3 = st.nextToken();
        SOP(s1); → java
        SOP(s2); → P
    }
}

```

Ex-3:

```

import java.util.*;
class StringTokenizerDemo2
{
    psvm(String args[])
    {
        String s = " Java Programming Language ";
        StringTokenizer st = new StringTokenizer
            (s, " ");
        while(st.hasMoreTokens())
    }
}

```

```

    {
        String token = st.nextToken();
        SOP(token);
    }
}

```

OP
 java
 Programming
 Language

Exception Handling

→ While developing an application or a project we find 2 types of errors —

1. Compile time errors
2. Runtime errors

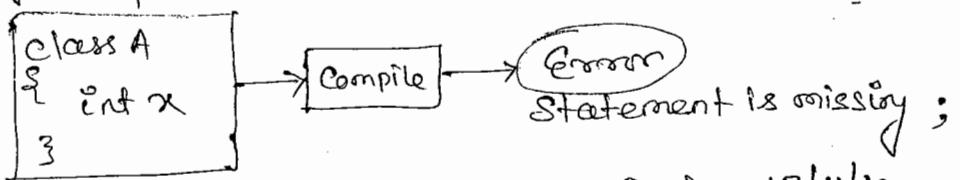
Compiletime errors:

→ Errors which are generated ~~at~~ during compilation of program is called compiletime errors.

→ All syntax errors are of this categories.

→ If any error during compilation, compiler doesn't generate bytecode. This error has to rectified by programmer.

Ex:-



Date - 17/11/12

Runtime errors:

→ An error occurs during execution of program is called runtime errors / logical errors.

→ Exception is a simple logical error which occurs during execution of program.

→ Exception is a runtime error.

→ Exception is an exception?

Q) What is an Exception?

→ Exceptional event — typically an error which occurs during run time.

→ Cance normal program flow to be disrupted.

→ An Exception is an event that occurs during execution of program that disrupts the normal flow of instructions.

Examples:-

- ① Divided by zero errors.
- ② Accessing the elements of an array beyond its range.
- ③ Invalid input.
- ④ Hard disk crash.
- ⑤ Opening a non-existent file.

Ex:-

Class DivideByZero

```

    {
        public static void main( String args[] )
        {
            SOP( 3/0 );
            SOP( "Pls. point me." );
        }
    }
  
```

→ The above program display this error message -

Exception in thread "main"

java.lang.ArithmaticException : / by zero

at DivideByZero.main(Divide By zero . java . 3)

Default Exception handlers :-

- provided by Java reenterne.
- prints out exception description.
- prints the stack trace (Hierarchy of methods or where the exception occurs).
- Causes the program to terminate.

Benefits of Java Exception handling framework :-

Separating Errorhandling code from "regular" business logic code.

Propagating errors up the call stack.

Grouping and differentiating error types.

Avoiding the abnormal termination of a program.

→ Creating an exception and giving to JVM called throwing exception.

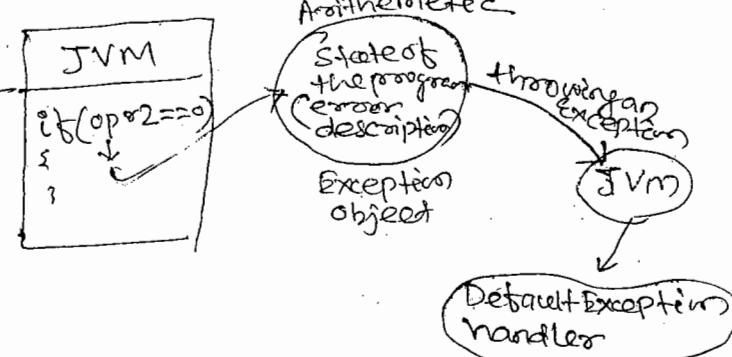
In that happens when ^{an} exception occurs;

→ When an exception occurs within a method, the method typically creates an exception object and hands it off to the runtime system.

- Creating an exception object and handing it to the runtime system is called "throwing an exception".

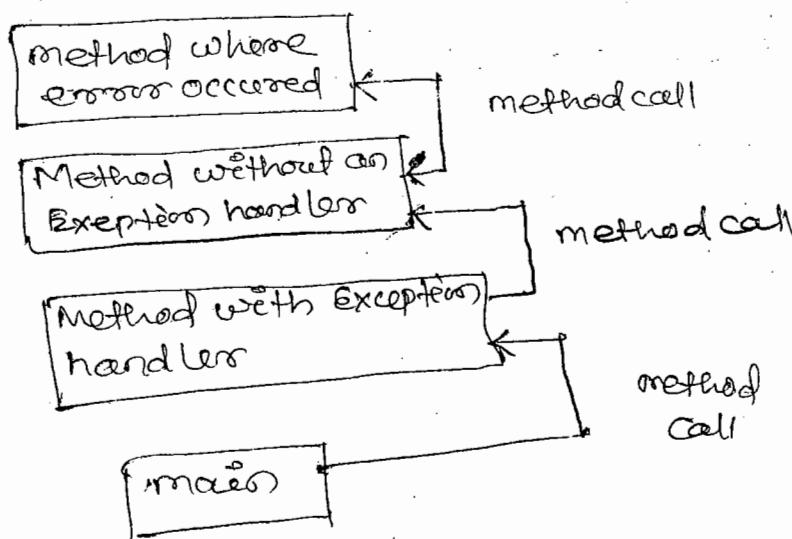
- Exception object contains information about the error, including its type and state of the program, when the error occurred.

```
PSVM(String args[])
{
    operands
    Sop("5/0");
    Sop("pointe");
}
```



⌚ Default Exception handler don't have the capability to stop abnormal termination. Default Exception handler is present in JVM.)

- The runtime system searches the call stack for a method that contains an exception handler.



red

Class Demo

```
{  
    static void m1() {  
        System.out.println("m1");  
    }  
    static void m2() {  
        m1();  
    }  
    static void m3() {  
        m2();  
    }  
    public static void main(String args[]) {  
        m3();  
    }  
}
```

~~but~~
→ When an appropriate handler is found, the runtime system passes the exception to the handler.

- An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.
- The exception handler chosen is said to catch the exception.

∴ the runtime system exhaustively searches all the methods on the call stack.

Q) What are the differences between Error & Exception ?

Date - 19/11/12

①

- Error type exceptions are raised or thrown due to the program problem occurred inside JVM, like -
- If there is no memory in SSA to create new stack frames for execute method, then JVM process is killed by throwing Error type Exception "java.lang.StackOverflowError".
 - If there is no memory in HA :- "java.lang.OutOfMemoryError".
- Exception type exceptions are raised or thrown, due to the problem occurred in java program logic execution, like
- If we divide an integer number with zero, then JVM terminates program execution by :- "java.lang.ArithmaticException".
 - We cannot catch Error type exceptions, becoz if error type exception is raised JVM process will be terminated.
 - We can catch Exception type exceptions, becoz JVM process will not be terminated.

Exception class Hierarchy :-

The Errors & Exception classes :-

1. Throwable class :-

- Root class of exception classes
- Immediate subclasses
 - Error
 - Exception

2. Exception class :-

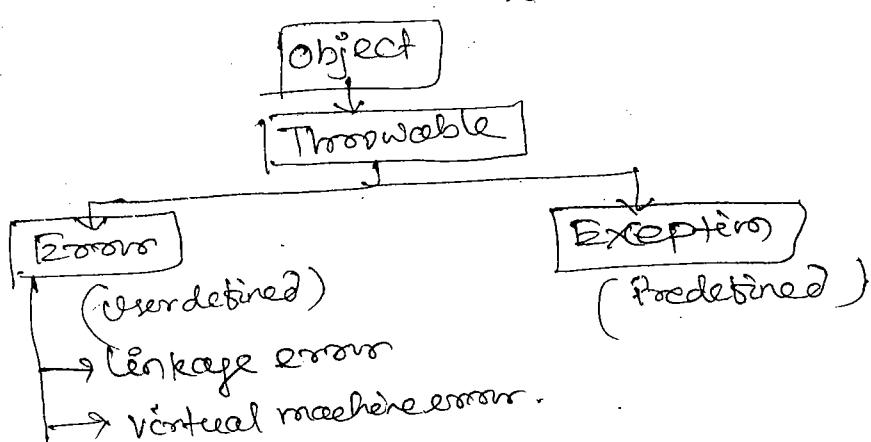
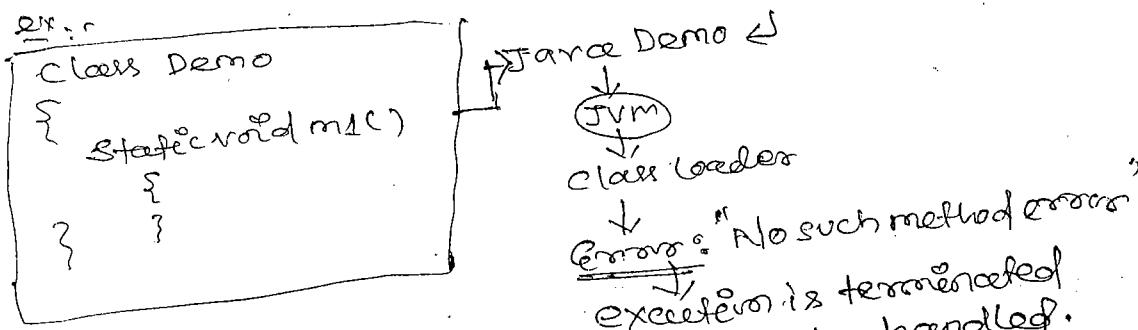
- Conditions that user programs can reasonably deal with.
- Usually the result of some flaws in the user program code.

3. Error class :-

- used by Java run-time System to handle errors occurring the run time environment.
- Generally beyond the control of user programs.

Example :-

- Out of memory errors.
- Hard disk Crash.

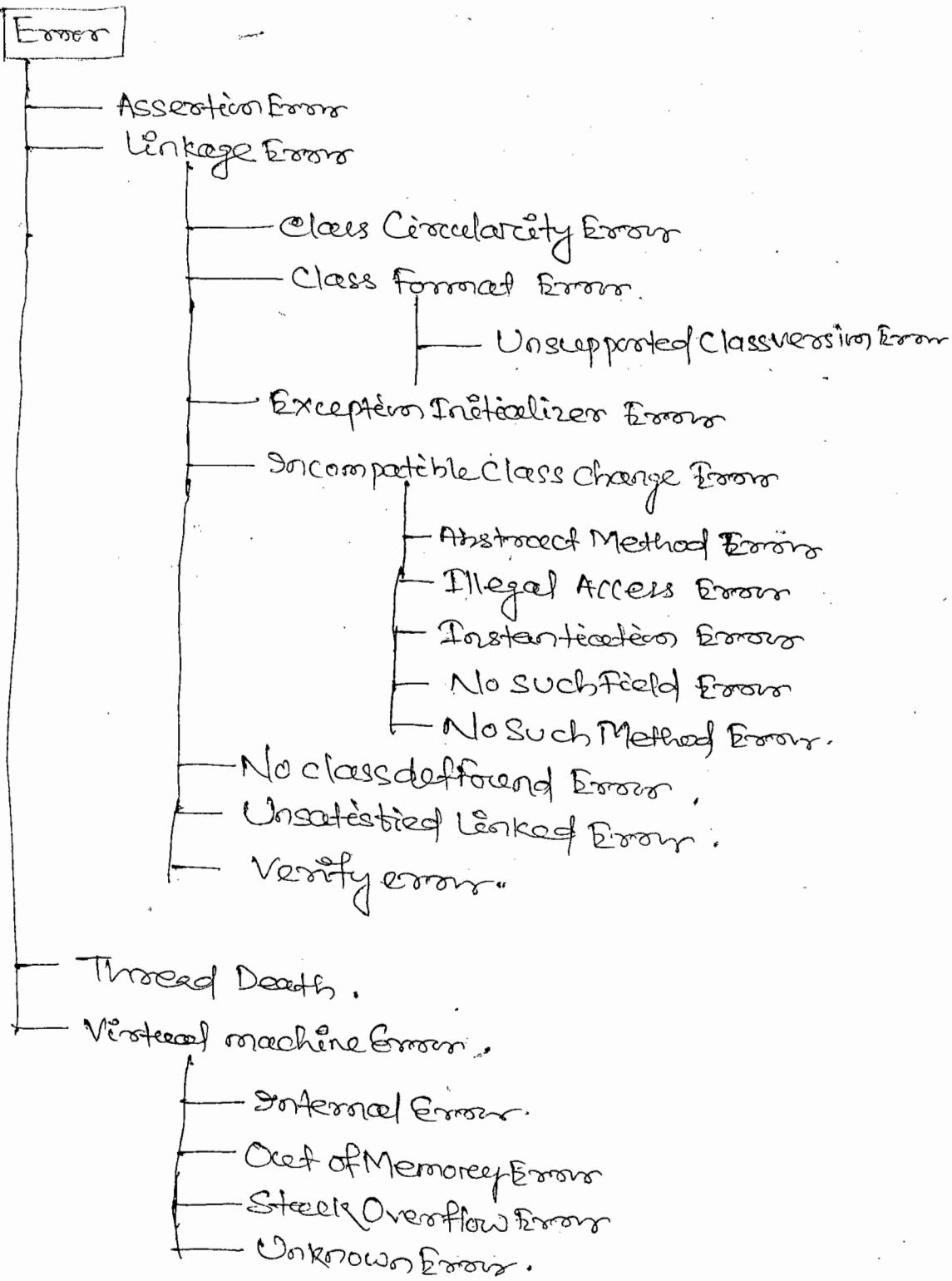


Exception

- ClassNotfound Exception
- ~~Clone~~ NotSupported Exception
- IllegalAccess Exception
- Instantiation Exception
- InterruptedException
- IO Exception → I·EOF Exception
a. file Not found Exception.
- NoSuchField Exception
- NoSuchMethod Exception
- RuntimeException →
 1. Arithmetic Exception
 2. ArrayStoreException
 3. ClassCastException
 4. IllegalArgument Exception

(a) IllegalThreadStateException
(b) NumberFormatException
as subclass)

- 5. IllegalMonitorStateException
- 6. ~~IllegalStateException~~ IndexOutOfBoundsException .
- 7. NegativeArraySizeException
- 8. NullPointerException
- 9. SecurityException
- 10. EnumConstantNotPresentException
- 11. TypeNotPresentException
- 12. IndexOutOfBoundsException
 - (a) ArrayIndexOutOfBoundsException
 - (b) StringIndexOutOfBoundsException



Types of Exception:-

- 2 types of Exceptions are there —
 1. checked Exception.
 2. Unchecked Exception.

Checked Exception:-

- It is a critical logical error.
- Java compiler checks if the program either catches or lists the occurring checked exception.
- If not compiler error will come.
- Subclasses of `Exception` class are checked exception, except `RuntimeException`.

Unchecked Exception:-

- Not subject to compile time checking for exception handling.
- Built-in exception classes
 - Error
 - RuntimeException
 - Their subclasses.
- If we don't handle Unchecked exceptions, it may not give any error.

Exception handlers:-

- ① Try & Catch (User-defined exception handle).

Try block:-

- The code which has to be monitored for exception handling defined within this block.
- Statements which are expected to throw an exception included inside this block.

Syntax:- `try`

```

    {
        Statements;
    }
```

try {

(if all the code present
inside the try block)

} → If it finds error at let statement, then it will not execute execute other statement.

→ This block contain business logic code.

2. Catch block :-

- It is an exception handler.
- This block contain error handling code.
- This block receive an exception thrown by try block.
- A try block followed by one or more than one catch block.

Syntax :-

catch (Exception-type reference)

{
 statements;
}

Syntax of try & catch :-

try
{
 statements which are has to be monitored for exception handling;
}

catch (Exception-type ref)

{
 error logic code;
}

catch (Exception-type ref)

{
 error logic code;
}

Null pointer Exception:

- These exceptions are Unchecked Exceptions.
- This extends from RunTimeException.
- ✓ Thrown when an application attempts to use NULL in case, when an object is required.
- These includes —
 - (i) Calling the instance method of NULL object.
 - (ii) Accessing or modifying the field of a NULL object.
 - (iii) Taking the length of null as if it were an array.
 - (iv) Throwing NULL as if it were a throwble value.
 - (v) Accessing or modifying the slots of null as if it were an array.
- Applications should throw instances of this class to indicate other illegal uses of the null object.

```
Ex:- class A
    {
        void m1()
    }
}
```

⇒ A obj1 = null;
obj1.m1(); ← Exception.

Ex:-

```
class A
{
    int x=10;
}
```

Exception

class ExceptionDemo

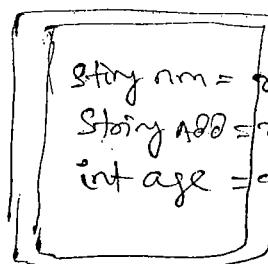
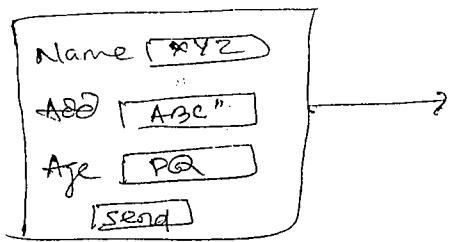
```
{
    psvm(String args[])
}
```

```
{
    A obj1=null;
    SOP(obj1.x); ← Exception
}
```

java.lang.NullPointerException

NumberFormatException:

Thrown to indicate that the application has attempted, to convert a String to one of the numeric types, but that the String does not have the appropriate format.



Sent to:

String s = "abc";
Integer.parseInt(s);
↓
Exception

String s = "123";
int x = Integer.parseInt(s);
↓
Success

Ex:-

Class ExceptionDemo2

```
{ public void main (String args[])
{ int x = Integer.parseInt(args[0]);
  int y = Integer.parseInt(args[1]);
  int z = x+y;
  System.out.println(z);
}
> java ExceptionDemo2 10 20
30
> java ExceptionDemo2 10 abc ← ← Exception
```

Command Line Argument

InstantiationException:

→ It is a checked exception.

→ Thrown when an application tries to create an instance of a class using the newInstance() method in class Class, but the specified class object cannot be instantiated. The InstantiationException can fail for a variety of reasons including but not limited to:

- The class object represents an abstract class, an interface, or an array class, or primitive type or void.
- The class has no nullary constructor.

- (1) new \leftarrow checked at the time of compilation.
- (2) Class.forName("class-name") • newInstance()
 ↗ dynamically creating object. checked
 at runtime not compile time.

4(s): ClassNotFound Exception:

- It is a checked exception.
- Thrown when an application tries to load an class through its strong name using:
 - The forName() method in class Class.
 - The findSystemClass() method in class ClassLoader.
 - The loadClass() method in class ClassLoader.
- but no definition for the class with the specified name could be found.

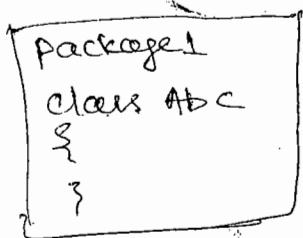
Ex:- class ExceptionDemo2

```
{ public static void main(String args[])
{ Class.forName("abc"); } }
```

↳ Exception.

IllegalAccess Exception:

- It is a Checked Exception.
- An IllegalAccess Exception is thrown when an application tries to reflectively create an instance (other than an array) set or get a field, or invoke a method, but the currently executing method does not have access to the definition of the specified class, field, method or constructor.



`Class.forName("package1.Abc")`

↓
exception

becoz Abc is declared cannot access
outside the package.

↓
Illegal Access Exception →

(Always check
access specifiers
when this
error comes.)

Date - 21/11/12

Class Cast Exception:

→ It is a unchecked exception.

→ Thrown to indicate that code has attempted
to cast an object to a subclass of which
it is not an instance.

for example:-
the following code generates a class cast

exception.

`Object x = new Integer(0);` contain integer object.

`SOP((String)x);` ↑ object type converts to
String type

ex:- `SOP((Integer)x);` → does not throw any exception.

`Employee e = new SalariedEmployee();`

`SalariedEmployee sd = (SalariedEmployee)e;`

↳ valid

`Department d = (Department)e` → Invalid (Exception)

Arithmetic Exception:

→ Unchecked exception.

→ Thrown when an exceptional arithmetic
condition has occurred.

ex:- An integer "divided by zero" throws an
instance of this class.

Array Index Out of Bounds Exception:

→ Unchecked Exception

→ Thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

Array Store Exception:

→ Unchecked Exception

→ Thrown to indicate that an attempt has been made to store the wrong type of object into an array of objects.

For ex:-

The following code generates an exception.

Object x[] = new String[3];

x[0] = new Integer(0); ← Array is String type
but we wants to
store Integer type.

Example:-

WAP to read the elements of an array.

import java.util.*;
class ExceptionDemo3

{ public static void main(String args[])

{ int a[] = {10, 20, 30, 40, 50};

Scanner scan = new Scanner(System.in);

sop("Enter index");

int index = scan.nextInt();

try{

sop(a[index]);

}

Catch (ArrayIndexOutOfBoundsException e)

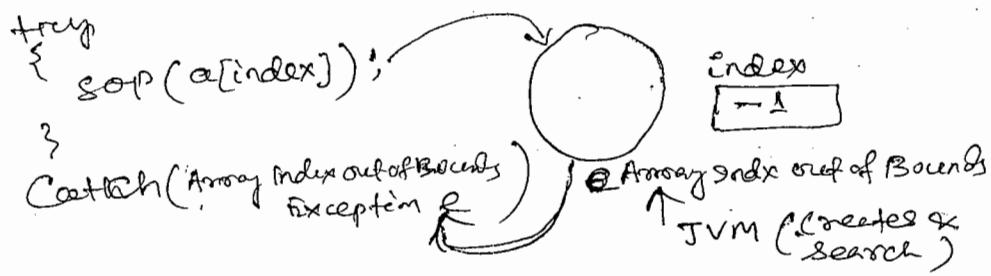
{ sop("invalid index number");

}

sop("Point me");

}

}



→ If there is one try block, no

try block followed by multiple catch block:

→ If try block expected to throw multiple types of exceptions, those exceptions handle using multiple catch blocks.

Syntax:

```

try {
    // code which has to be monitored for
    // exception handling.
}
catch (ExceptionType ref) {
    // error logic code
}
catch (ExceptionType ref) {
    // error logic code
}

```

.....
MAP to create an object of any class.

```

import java.util.*;
class ExceptionDemo {
    public static void main (String args[]) {
        Scanner scan = new Scanner (System.in);
        System.out.println ("Input class name");
        String name = scan.next();
    }
}

```

try {

Object o = Class.forName(name).newInstance();

SOP(o);

}

Catch (ClassNotFoundException e1)

{
SOP("invalid class name or class not found");

}

Catch (InstantiationException e2)

{
SOP("Error in creating object");
or the class object is an interface, array, abstract class

SOP("Container");

Catch (IllegalAccessError e3)

{
SOP("insufficient privilege");

}

OR Spec classname

A

A@61de33

} only A (not A.class)

Ex:-

import java.util.*;

Class ExceptionDemos

{ public static void main (String args[])

{ Scanner scan = new Scanner (System.in);

try {

SOP("Input any two numbers");

int x = scan.nextInt();

int y = scan.nextInt();

int z = x+y;

SOP("z = "+z);

}

Catch (ArithmeticeException e1)

{
 SOP("Error in Division");

}
Catch (NumberFormatException e2)

{
 SOP("Invalid type of values");

}

}{
}

Handling multiple types of Exceptions thrown by try block using one catch block :-

→ If try block expected to throw multiple types of exceptions, those exceptions are handle using one catch block.

→ In this type case catch block must have parameter of Supertype.

→ Supertype can hold Subclass object.
ex:-

1. Catch (Throwable t) → handle Errors & Exceptions.

{

}

2. Catch (Exception e) → Exception Subclasses checked/unchecked

{

}

3. Catch (RuntimeException e) → handle only runtime exception subclasses unchecked Exceptions.

{

}

example :-

// try block followed by one catch block

Class ExceptionDemo6

{
 public static void main (String args [])

{

```
try{
```

```
    int x = Integer.parseInt(args[0]);  
    int y = Integer.parseInt(args[1]);  
    int z = x/y;  
    sop("z = "+z);
```

```
}
```

```
Catch (Exception e)
```

```
{ if (e instanceof NumberFormatException)
```

```
    sop("invalid type argument");
```

```
else if (e instanceof ArithmeticException)
```

```
    sop("Error in division");
```

```
else if (e instanceof ArrayIndexOutOfBoundsException)
```

```
    sop("invalid no. of arguments");
```

```
}
```

Date - 22/11/12

Try block having multiple catch block:-

Class Exception Demo

```
{ class ExceptionDemo
```

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

```
{ try{
```

```
    int x = Integer.parseInt(args[0]);
```

```
    int y = Integer.parseInt(args[1]);
```

```
    int z = x/y;
```

```
    sop("z = "+z);
```

```
}
```

```
Catch (ArithmetcException e)
```

```
{ sop("Error in Division");
```

```
}
```

```
Catch (Exception e1)
```

```
{
```

```
    sop(e1);
```

```
}
```

```
}
```

- The hierarchy of writing the catch block is sub-type followed by super-type.
- If we write let super-type it will provide error, that the "xxxException has been already caught".

Rule-1 :-

```
Class ExceptionDemo8
{
    public static void main( String args[])
    {
        try{
            int x = Integer.parseInt(args[0]);
            SOP(x);
        }
        Catch( ArrayIndexOutOfBoundsException e1)
        {
            SOP("Invalid no. of arguments");
        }
        Catch( ArrayIndexOutOfBoundsException e2)
        {
        }
    }
}
```

two
two

- We cannot have same catch block, if provide errors.

- The above program display compile time errors that the "xxxException has already caught".
- There should not be more than one catch block having same type of exception.

Rule-2 :-

```
Class ExceptionDemo8
{
    p.s.v.m( String args[])
    {
        try{
            int x = Integer.parseInt(args[0]);
            SOP(x);
        }
    }
}
```

Catch (Exception e1)

{
 sop ("Inside Exception");

}

Catch (ArrayIndexOutOfBoundsException e2)

{
 sop ("Invalid Array Index");

}

}

The above program displaying complete error.

- The above program displaying complete error.
- Catch block with Sub-type exception followed by Catch block with super-type.

Finally :-

Answer

- It is a keyword.
- The block of code is always executed despite of different scenarios —
 - forced exit occurs using a return, continue or a break statement.
 - Normal completion.
 - Caught Exception thrown
 - Exception was thrown and Caught in the method.
 - UnCaught Exception thrown
 - Exception thrown was not specified in any Catch block in the method.

→ Finally is a block of code, which is executed before termination of the program.

→ Try block must have one finally block, but more than one ~~one~~ catch block.

Syntax :-

1. try - catch - finally :-

```
try {  
    statements which are expected to throw  
    exception;  
}  
catch (Exception-type ref)  
{  
    error logic code;  
}  
finally  
{  
    code which must be executed;  
}
```

2. try...catch...catch...finally :-

```
try {  
    statements to have monitor for exception  
    handling;  
}  
catch (Exception-type ref)  
{  
    error logic code;  
}  
catch (Exception-type ref)  
{  
    error logic code;  
}  
finally  
{  
    code must be executed;  
}
```

3. try...finally:-

```
try {  
    statement expected to throw exception  
}  
finally {  
    code must be executed;  
}
```

Ex:-

P.S.V.m (String args[])
try {

open connection to pointer
with pointing.

Normal Flow

caught Excep.

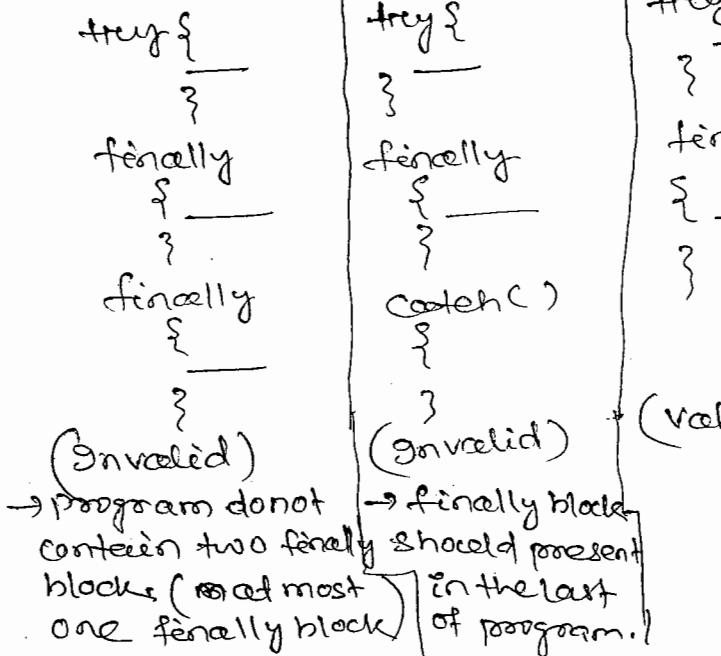
Catch()

finally

close the Connection

SOP ("Thanks for visiting me");

when uncaught Exception
arises program
terminated
abnormally
but finally
block executed.



Ex:-

Class Exception Demo 10

{

P.S.V.m (String args[])

{ SOP ("Inside main method");

try {

SOP ("Inside try block");

SOP (Integer.parseInt(args[0]));

}

Catch (NumberFormatException e)

{ SOP ("Inside catch block");

SOP ("Invalid type of arguments");

}

```
finally
{
    SOP("Inside finally block");
}
SOP("Continue ....");
```

O/P:- java ExceptionDemo@

Inside main method

Inside try block

Inside final block

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
. because exception".

Ex:-

Class ExceptionDemo11

```
{ static int m1()
```

```
{ int num=5;
```

try{

```
SOP("Inside try block");
```

```
if(num/0=0)
```

```
return num;
```

}

finally

```
{ SOP("Inside finally block");
```

}

```
return num;
```

```
}
```

public static void main (String args[])

```
{
```

```
int x = m1();
```

```
SOP(x);
```

}

}

23/11/12

Throws :-

~~~~~

Rules on Exception :-

→ A method is required to either catch or list all exceptions it might throw

- Exception from Error or RuntimeException, or their subclasses.

→ If a method may cause an exception to occur but does not catch it, then it must say so using the throws keyword

- Applies to checked exception only.

Syntax :-

`<type> <method-name> (<parameters-list>) throws  
                          <exception-list>`

{

    method body ;

}

(OR)

`return-type <method-name> (<parameters>) throws  
                          <exception-types>, <exception-type>, ...`

{

    statements ;

}

Ex:-

`class ExceptionDemo2`

`throws ClassNotFoundException`

{  
    public static void main (String args[]) throws A,  
                          InstantiationException, IllegalAccessException

{  
    Object o = Class.forName ("A").newInstance ();

    System.out.println (o);

}  
}

→ Here main method does not handle the exception ; but

it just forwarding the Exceptions .

Ex-1

## Class Exception Demo13

static Object createObject (String name) throws  
ClassNotFoundException, InstantiationException,  
IllegalAccessException

```
{ Object o = null;  
    o = Class.forName(name).newInstance();  
    referen o;
```

3 public static void main (String args [ ] )

*Σ* *λαζ*

try {  
 Object obj1 = CreateObject("A"); // static call  
 another static

SOP(Obj1);

۳

Catch (Exception)

۱۷۰

$\text{sop}(0);$

Rules from method overriding:

Class A

{ void m1() throws ClassNotFoundException

۳۲

3

- Class B extends A

8

void m1() throws Exception

३

3

→ The above program displays CE that bcz overriding method cannot throw super-type exception of exception thrown by overridden method.

Class A

```
{ void m1() throws Exception  
{  
}  
}
```

Class B extends A

```
{ void m1() throws ClassNotFoundException  
{  
}  
}
```

→ The above program does not display any error

overriding method can throw sub-type exception thrown by overridden method.

~~Note~~ → Throws clause should not be added if superclass method doesn't contain it. If superclass method contains throws clause, in subclass overriding method should contain throws clause with same exception class or its subclasses or it can remove throws clause all together.

Throw : —

~~~~~

→ It is a keyword.

→ Java allows us to throw exceptions (generate exception)

throw <exception object>

→ An exception we throw is an object

→ We have to create an exception object in the same way we create any other object.

Q) What is the diff. b/w throw & throws?

Throw

- It is used for generating → It is used for forwarding exception explicitly.
- A throw is used to throw an exception manually.

Where as throws is used in the case of checked exceptions, to tell the compiler that we haven't handled the exception, so that the exception will be handled by the calling function.

Throwable Class :-

- The throwable class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement.
- Similarly only this class or one of its subclasses can be the argument type in a catch clause.

Constructors :-

Throwable():-

Constructs a new throwable with null as its detail message.

Throwable(String message):-

Constructs a new throwable with the specified detail message.

methods:-

String getMessage():-

Refers the detailed message String of this throwable.

void printStackTrace():-

Prints this throwable and its backtrace to the standard error stream.

void printStackTrace(PrintStream s):-

Prints this throwable and its backtrace to the specified print stream.

Date - 24/11/12

WAP for generating exception or throwing exception:-

import java.util.*;

Class ExceptionDemo15

{ public static void main (String args[])

{ Scanner scan = new Scanner (System.in);

sop ("Input any two numbers");

int n1 = scan.nextInt();

int n2 = scan.nextInt();

try {

if (n2 == 0)

throw new ArithmeticException();

else

{ int n3 = n1 * n2;

sop(n3);

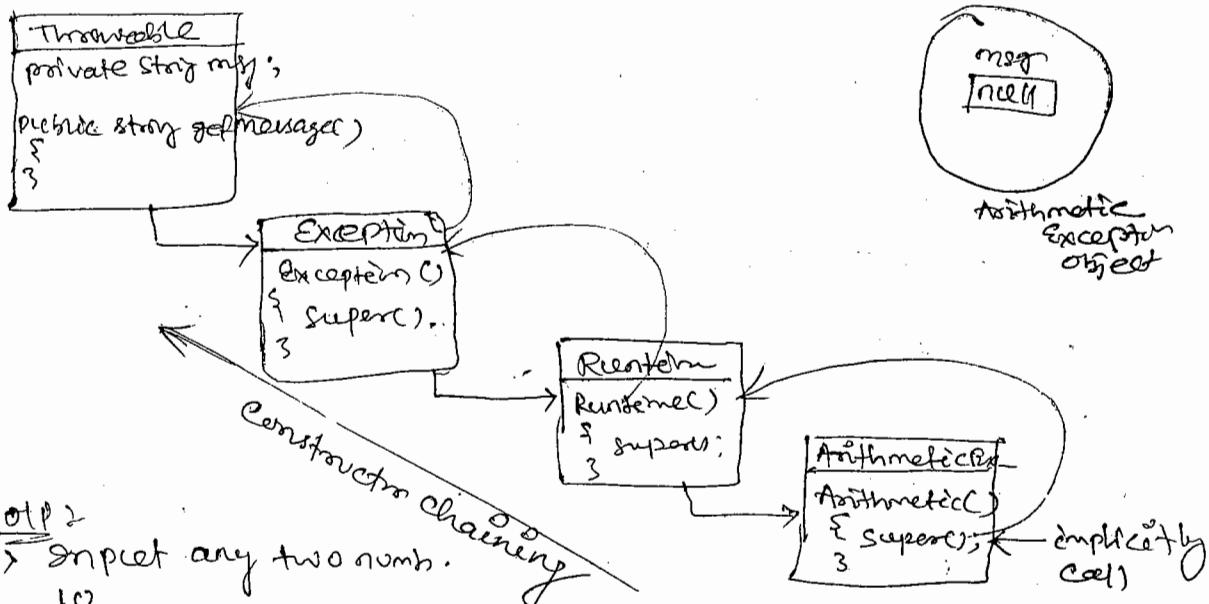
} // try

catch (ArithmeticException e)

{ sop(e.getMessage());

}

} ?



Q1 P2
 > Input any two numbers.

10

20

200

> -
 > Input any two numbers

10

0

null

T

It produces error as null bcz.
 object contain the variable msg,
 which has value null. As we use
 default const. we get null.

→ The above example shows the default Constructor
 call. Constructor chaining occurs implicitly.

// Generate error with parameterized Const.

WAP for Login application.

import java.util.*;

class Login

{ public static void main (String args[])

{ Scanner scan = new Scanner (System.in);

System.out.println("Enter User name");

String Uname = scan.next();

System.out.println("Enter password");

String password = scan.next();

String

{ if (Uname == "PRITAM")

if (username.equals("pratam") && password.equals("patnaik"))

sop("Welcome to my application");

else

throw new RuntimeException("Invalid Username
or password");

} // try

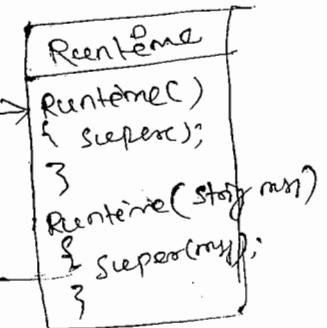
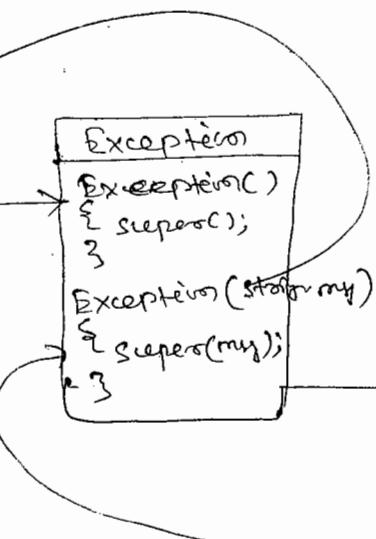
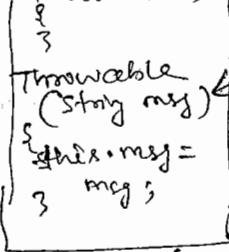
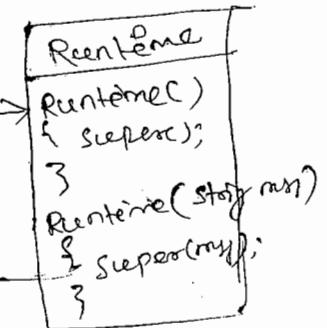
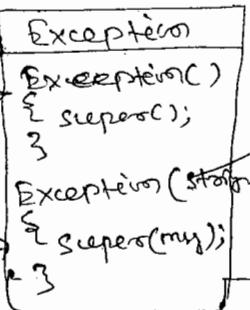
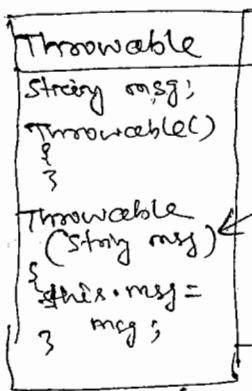
catch (RuntimeException e)

{ sop(e.getMessage());

}

}

new RuntimeException (e)



tic
caption
object

placement
x1)

Creating own Exception Class:

Steps to follow —

- Create a class that extends the RuntimeException or the exception class.
- Customize the class.
 - Members and constructors may be added to this class.

import java.util.*;

↳ unchecked
Exception.

```
class InsufficientBalanceException extends
    RuntimeException
```

```
{   InsufficientBalanceException()
```

```
{}
```

```
}
```

```
InsufficientBalanceException(String msg)
```

```
{
```

```
super(msg);
```

```
}
```

```
}
```

class Account

```
{ private int accno;
```

```
private float balance;
```

```
Scanner scan = new Scanner(System.in);
```

```
void read()
```

```
{   System.out.print("Input accno");
```

```
accno = scan.nextInt();
```

```
System.out.print("Input balance");
```

```
balance = scan.nextFloat();
```

```
balance = scan.nextFloat();
```

```
}
```

void withdraw (float amt)

```
{   float
```

```
{
```

if (amt > balance)

```
throw new InsufficientBalanceException
("Balance not available");
```

```

    else
        balance = balance - tamt;
    }
    } catch (InsufficientBalanceException e) {
        SOP("e.getMessage()");
        {
            {
                {
                    // try
                    String getAccount()
                    {
                        return accno + "balance"
                    }
                }
            }
        }
    }
}

class ExceptionDemo16
{
    public static void main(String args[])
    {
        Account acc1 = new Account();
        acc1.read();
        Scanner scan = new Scanner(System.in);
        SOP("Input transaction amount");
        float amnt = scan.nextFloat();
        acc1.withdraw(amnt);
        SOP(acc1.getAccount());
    }
}

```

Ex-2:-

```

import java.util.*;
class LoginException extends Exception
{
    LoginException()
    {
        super("Invalid Username or password");
    }
}

```

class Login

```

{
    static void login(String uname, String pwd)
    {
        try
        {
            if(uname.equals("Mike") & pwd.equals("miki"))
                SOP("Welcome to my Application");
        }
    }
}

```

```

        else
            throw new LoganException();
    }

    Catch ( LoganException e )
    {
        sop ( e.getMessage() );
    }
}

```

Class ExceptionDemo17

```

public static void main ( String args[] )
{
    Scanner scan = new Scanner ( System.in );
    sop ( " Input user name " );
    String cname = scan.next();
    sop ( " Input password " );
    String pwd = scan.next();
    Logan logan ( cname, pwd );
}
}

```

Nested try block :

→ A try block inside/written another try block
is called nested try block.

Syntax :

```

try
{
    statements expected to throw exception;
}

try
{
    statement expected to throw
    exception;

    catch (Exception-type ref)
    {
        error logic code;
    }

    catch (Exception-type ref)
    {
        error logic code;
    }
}

```

Ex:-

Class Exception Demo 18

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

```
        int x = Integer.parseInt(args[0]);
        int y = Integer.parseInt(args[1]);
```

```
        int z = x / y;
        System.out.println("z = " + z);
    } catch (ArithmaticException ae)
    {
        System.out.println("Error in division");
    }
}
```

I
N
N
E
R
T
R
Y

```
    } catch (NumberFormatException ne)
    {
        System.out.println("Invalid type of arguments");
    }
}
```

```
    } catch (ArrayIndexOutOfBoundsException aie)
    {
        System.out.println("Invalid no. of arguments");
    }
}
}
```

Ex-2:-

Class Exception Demo 19

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

```
        int a =
```

```
        System.out.println("stmt1: " + Integer.parseInt(args[0]));
```

```
        try {
```

```
            int b =
```

```
            System.out.println("stmt2: " + Integer.parseInt(args[1]));
```

```
}
```

```
    } catch (Exception e)
    {
        System.out.println("Inner try catch block");
    }
}
```

```

    sop("stmt 3");
}
// Outer try
Catch(RecognitionException e1)
{
    sop("Outer catch block");
}
}
}

```

Ex-3:-

```

Class ExceptionDemo20
{
    public static void main (String args[])
    {
        try
        {
            sop("Outer try block" + Integer.parseInt(args[0]));
        }
        try
        {
            sop("Inner try block" + Integer.parseInt(args[1]));
        }
        Catch(NumberFormatException e1)
        {
            sop("Inner Catch block");
        }
    }
}

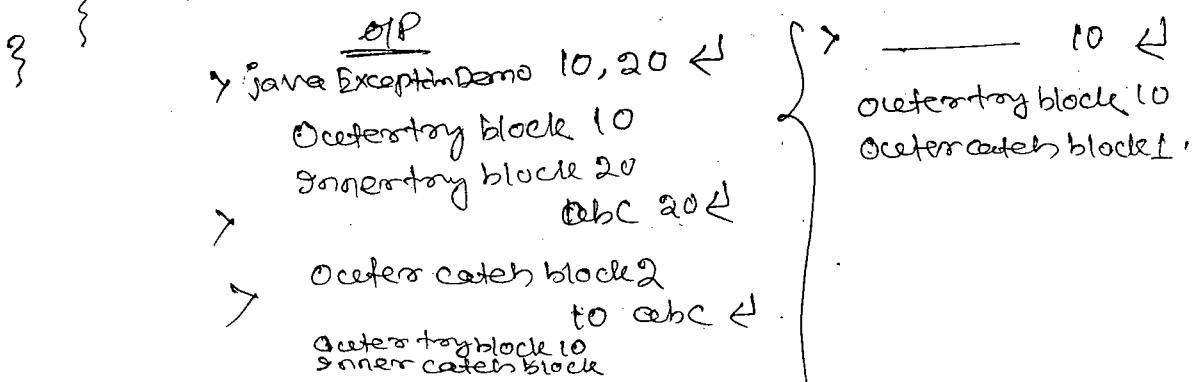
```

Catch (ArrayIndexOutOfBoundsException e2)

```

{
    sop("Outer catch block1");
}
Catch(NumberFormatException e3)
{
    sop("Outer catch block2");
}
}
}

```



Re-throwing Exception:

- The exception thrown by inner try block forward it to the outer catch block. It is called re-throwing exception.
- Forwarding exception from one catch block to another ^(inner catch) ^(outer catch) catch block is called re-throwing exception.

Ex:-

Class ExceptionDemo21

{ public static void main (String args[])

{

 try

 { System.out.println(args[0]);

}

 catch (ArrayIndexOutOfBoundsException e)

{

 throw e;

}

}

 OP

> java ExceptionDemo21 10

10

>

↙

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException

Exception.

Ex:-

Class ExceptionDemo22

{ public static void main (String args[])

{

 try

 {

 System.out.println(args[0]);

}

 catch (ArrayIndexOutOfBoundsException e)

{

 throw e;

}

↙

10

<1.

Catch (Array Index Out of Bounds Exception) e2)

SOP ("order batch") ;

O/P
> java ExceptionDemo22
refercatch :

Asseretéons

What are Assertions?

- Allows the programmer to find out if the program behaves as expected.
 - Informs the person reading the code that a particular condition should always be satisfied.
 - Running the program informs us, if assertions made are true or not.
 - If an assertion is not true, An Assertion Error is thrown
 - User has the option to turn it off or on when running the application.

Syntax:

Assent Conditions;

- Assertions are introduced in "javac -d".

Ex :-

Class AssetDeriv1

Class AssertDeriv1
{} public static void main (String args []) { }

```
{ int x = Integer.parseInt(args[0]); }
```

assert x >= 10 ;

Sop(2);

Op

→ java assertDemo⑤ ↵

5

Not checked assertion
here

> java -ea AssertDemo1 5 ↵

→ for enabling Assertion.

Exception in thread main "java.lang.AssertionError".

> java -ea AssertDemo1 25 ↵

25.

Note :- If we write "-ea", at runtime the assertion will be enabled. otherwise disable.

// printStackTrace() :- (It will print back tracing)

Class ExceptionDemo22

{ static void div()

{ SOP(5/0); }

public static void main(String args[])

{

try

{ div(); }

} catch(ArithmeticException e)

{ e.printStackTrace(); // back tracing.

}

}

O/P :-

java.lang.ArithmeticeException : / by zero.

at ExceptionDemo22.div (ExceptionDemo22.java:7)

at ExceptionDemo22.main (ExceptionDemo22.java:14)

→ back tracing. (the flow of propagating errors).

→ By using this method identifying the exception is very easy.

Wrapper Classes

26/11/12

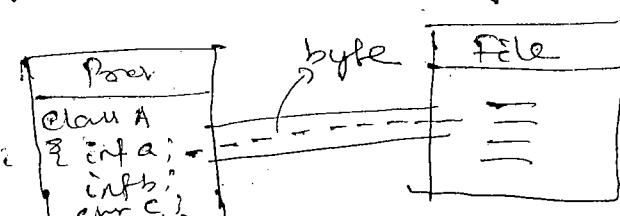
→ A

What is Wrapper Classes?

- They are wrappers to primitive types.
They allows us to access primitives as objects.
- Wrappers classes allows to encapsulate primitive data and representing it as an object type.
- The classes which are used to represent primitive value as object are called Wrapper class.
- In `java.lang` package we have 8 wrapper classes one per each primitive type separately.

| Primitive datatype | Wrapper class |
|--------------------|---------------|
| int | Integer |
| long | Long |
| short | Short |
| byte | Byte |
| float | Float |
| double | Double |
| char | Character |
| boolean | Boolean |

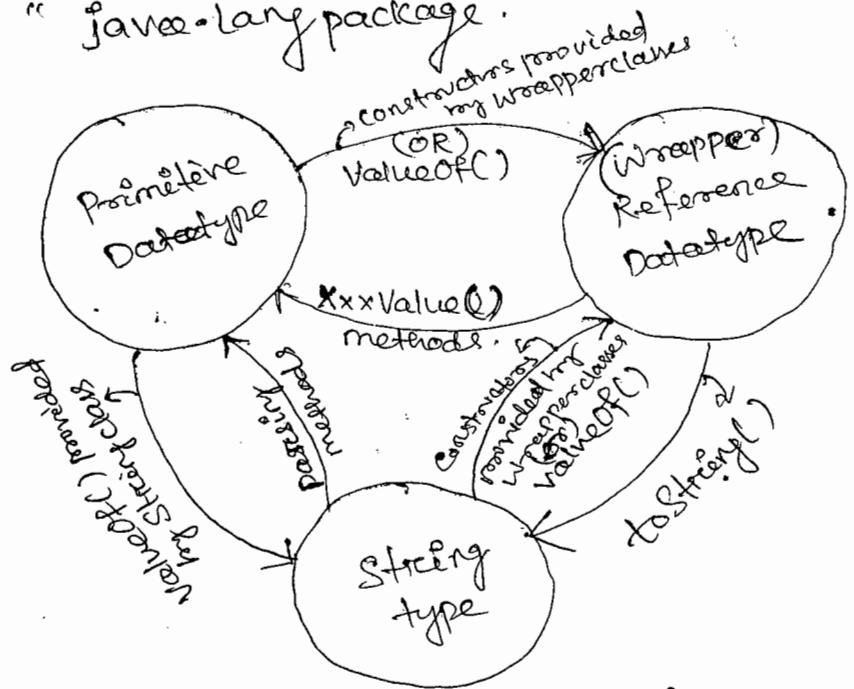
- Wrapper classes are used to converting one datatype to another datatype.



(int, char, float data converts to byte by
(Wrapper class)).

- Wrapper classes allows or provide methods for converting one type data to another type.
- Wrapper classes are used in project to perform conversion operations.

→ All these classes are available in —
 "java.lang package"



Constructors of Wrapper Classes:

(B) Every wrapper class provides Constructors.

- One constructor which wrap primitive type and represented as Object.
- Other constructor which wrap string type and represented as Object.

Ex:- Integer class

1. Integer (int)
2. Integer (String)

Ex:-

Class WrapperDemo1

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

```
{ Integer x = new Integer(10); }
```

```
Integer y = new Integer("10");
```

```
float z = new Float(1.5f);
```

```
float f1 = new Float("1.5");
```

```
Double d1 = new Double(2.5);
```

```
Double d2 = new Double("2.5");
```

sop(x); // x.tostring()
 sop(y);
 sop(z);
 sop(f1);
 sop(d1);
 sop(d2);

Output

(3) → E

(A) →

C

P

→

(2) → Every wrapper class overrides "tostring() method of object class, which does not get hashCode but get contents of the object.

Ex:

Class A

{ int x = 10;

} Class B

{ psvm (String args[])

{ A obj1 = new AC();

sop(obj1); // obj1.tostring() → A@25a5

getting the hashCode of
the object, but not give
the content.

Ex

(

)

:

Ex:

Class A

{ int x = 10;

public String tostring()

{ return String.valueOf(x);

}

Class Demo

{ psvm (String args[])

{ A obj1 = new AC();

sop(obj1);

overriding method.

it provides
Content of
hence it
override the
tostring()
method of
object class.

E

→ T

+

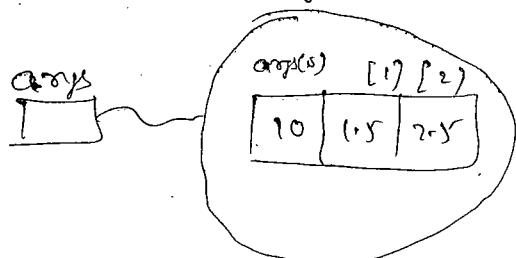
- Every wrapper class having single field of type primitive.
- Wrapper classes provides parsing methods, which converts string representation of primitive type to primitive types.
- All these parsing methods are static.

| method name | class | referent type |
|-------------|---------|---------------|
| parseInt | Integer | int |
| parseFloat | Float | float |
| parseDouble | Double | double |
| parseLong | Long | long |
| parseByte | Byte | byte |

ex:-

Class Wrapper Demo2

```
{
    public static void main (String args[])
    {
        int x = Integer.parseInt (args[0]);
        float y = Integer.parseFloat (args[1]);
        double z = Integer.parseDouble (args[2]);
        System.out.println (x);
        System.out.println (y);
        System.out.println (z);
    }
}
```



- Here we don't required to create an object bcoz there come static method.

⑤ Every wrapper class provides value() methods, which converts reference type(wrapper type) to primitive type.

→

→ These value() methods are not static methods.

→ The xxxValue() methods are —

int Value()

float Value()

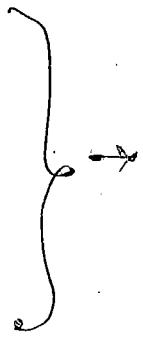
double Value()

long Value()

short Value()

byte Value()

boolean Value()



Provided by
all wrapper
classes

→ As these methods are not static methods, therefore we required to create object.

Ex:-

Class WrapperDemo3

{ public static void main (String args[])

 Integer x = new Integer(10);

 Integer y = new Integer(20);

 int n1 = ~~x~~ x.intValue();

 int n2 = y.intValue();

 int n3 = n1 * n2;

 System.out.println(n3);

}

⑥ Every wrapper classes provides valueOf() methods, which converts primitive type to reference type and String type to reference type.

→ This method is overloaded and it is a static method.

→ Wrapper class provides 2 valueOf() methods —

- 1. valueOf(String)
- 2. valueOf(primitive-type)

Ex:-

Integer:

valueOf(String)

valueOf(int)

Double:

valueOf(String)

valueOf(double)

④ Wrappers class "toString()" method returns String representation of wrapper type.

→ toString() method is method of Object class &

→ toString() method is method of Object class.

It's overridden in wrapper class.

→ This method is a non-static method.

Ex:-

Class WrapperDemo4

{

 p.s.v.m(s1)

 p.s.v.m(String arr[7])

 { Integer x = new Integer(10);

 Integer y = new Integer(20);

 String s1 = x.toString();

 String s2 = y.toString();

 sop(s1+s2); //Op → 1020

}

}

- (8) → String class provides valueOf() methods, which convert primitive type to String type.
 → This method is overloaded in String class.
 → It is a static method.

Ex:-

Class WrapperDemo5
 { psvm (String args[])

{ int x = 10;

int y = 20;

String s1 = String.valueOf(x);

String s2 = String.valueOf(y);

SOP(s1+s2); // 1020

}

Date - 27/11/12

Autoboxing & Unboxing:

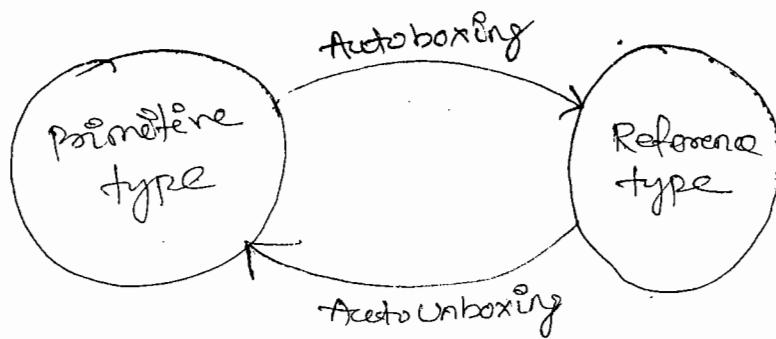
~~~~~  
 → It is a new feature of "java 5.0"

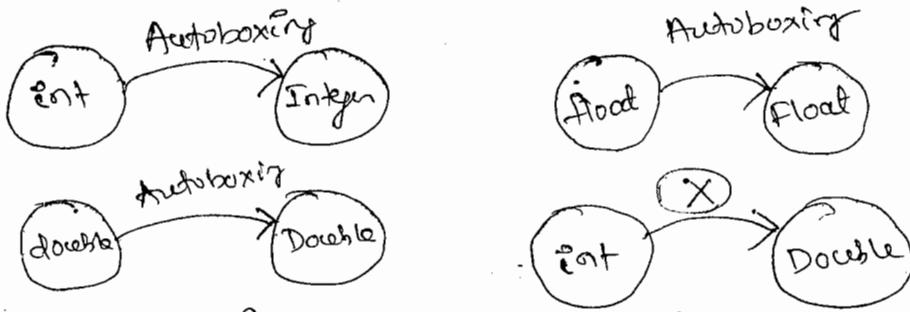
Autoboxing:

→ Autoboxing is a process of converting primitive type to reference type.

→ This conversion is automatic which done by compiler.

Integer x = 10; // error (before 1.5)





→ This conversion is between similar types.

Eg: double x = 10;

Double y = x; ✓

long l = x; → // error (not similar types)

Q2:-

Class AutoBoxDemo1

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

{ int x = 10;

Integer y = x; // AutoBoxing

float f1 = 1.5f;

Float f2 = f1; // AutoBoxing

SOP(x);

SOP(y);

SOP(f1);

SOP(f2);

}

Autounboxing:

→ The process of converting reference type to primitive type is called "autounboxing".

→ This conversion is done by compiler.

Q3:-

Integer x = new Integer(10);

int y = x; ← Autounboxing

Double d = new Double(9);

double d1 = d;

double d2 = x; ← error (not similar types)

# I/O Streams

What is Input?

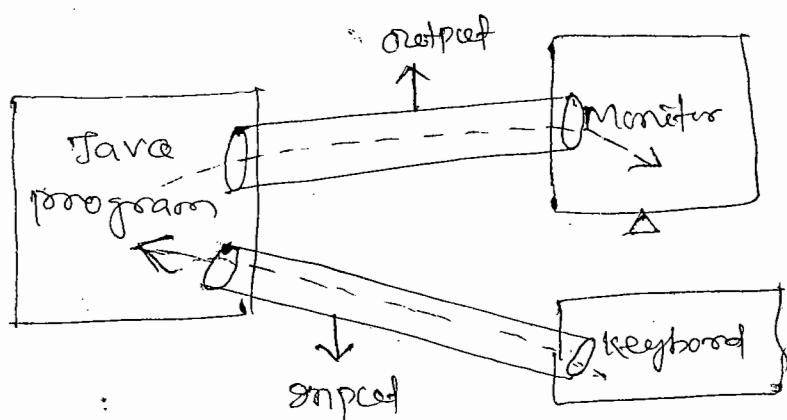
- Input The Data or information given to program is called Input.
- Data or Information which flow inside program is called Input.

What is Output?

- Data or information given by the program is called Output.
- Data or information which flow outside program is called Output.

Stream:

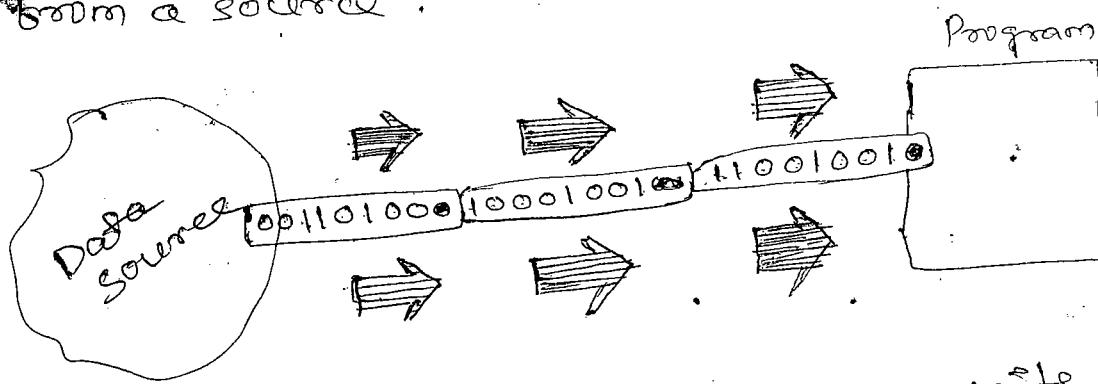
- Stream is class which provides a set of methods for input & output operations.
- A Stream is nothing but a flow of data.
- All I/O Stream classes available in "java.io package".



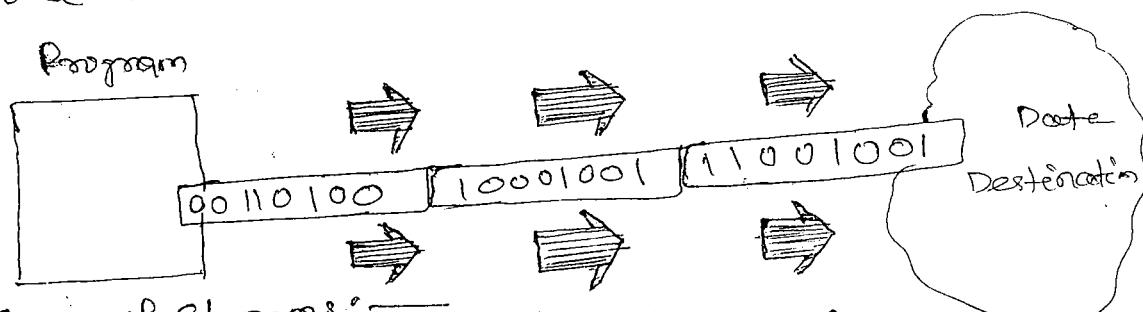
- Stream can be defined as "continuous flow of data between java program and persistence media".

## I/O Stream Basics:

- An I/O Stream represents an input source or an output destination.
- A Stream can represent many different kinds of sources and destinations, including disk files, devices, other programs etc.
- Streams support many different kinds of data, including simple bytes, primitive data types, localized characters and objects.
- Some Streams simply pass on data, others manipulate and transform the data in useful ways.
- A stream is a sequence of data.
- A program uses an Input Stream to read data from a source.



- A program uses an output Stream to write data to a destination.

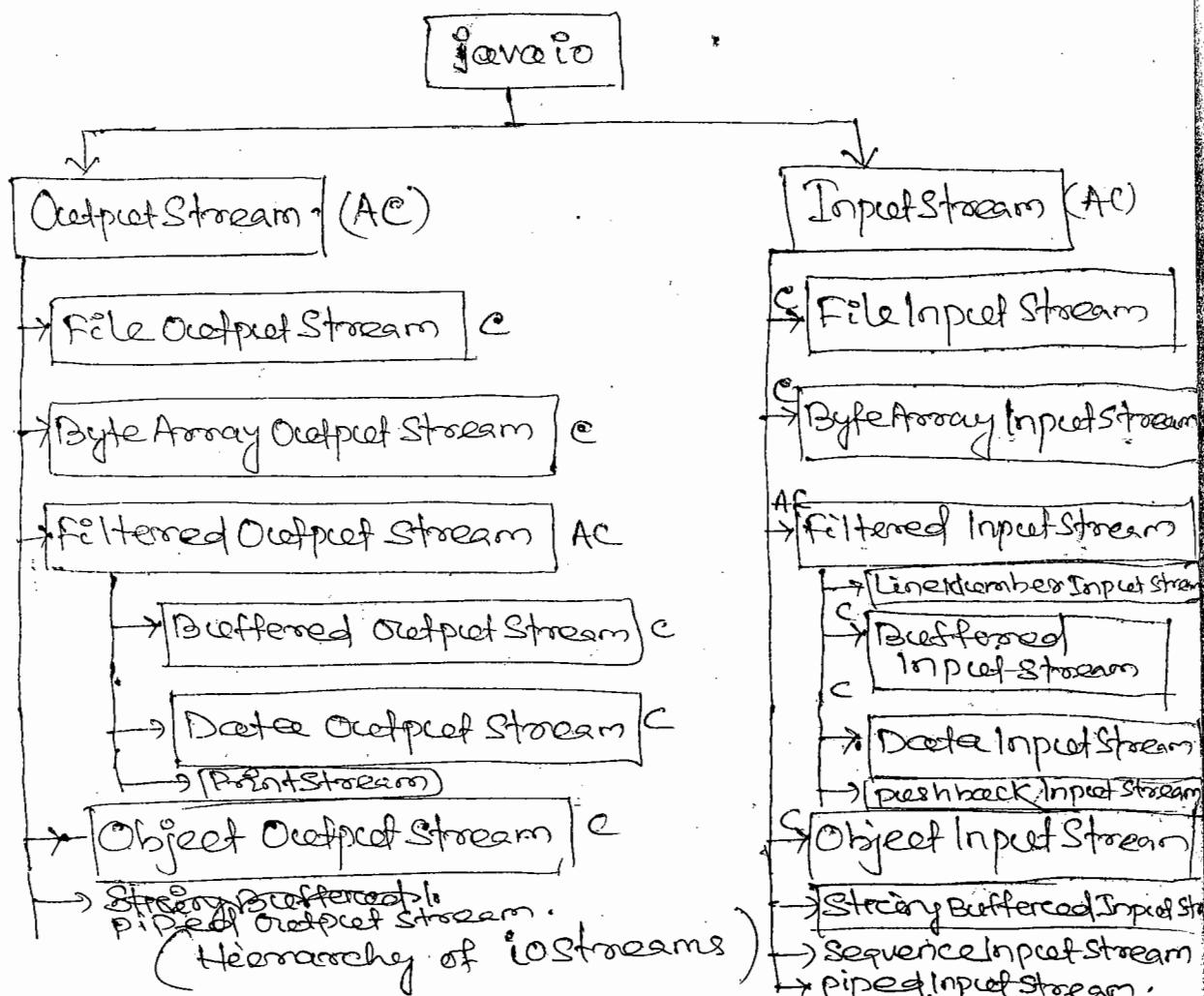


## Types of Streams:

- Stream is divided into 2 types - (based on type of data passed through streams)
  - 1) byte Streams.
  - 2) Character Streams.
- Stream is divided into 2 types based on data flow direction
  - 1) Input Stream.
  - 2) Output Stream.

Byte Streamis :-

- Programs use byte streams to perform input and output of 8-bit bytes. (i.e. 1 byte format)
  - All byte stream classes are ~~descended~~ descended from Input Stream & Output Stream.



Note :- C → Class & AC → Abstract Class

## Methods of iostreams:-

- `int read():`— Reads a byte of data from this input stream.
  - `int read(byte[] b):`— Reads upto b.length bytes of data from this input stream into an array of bytes.

- void write(int b) :- Writes the specified byte to this file output stream.
- void write(byte[] b) :- Writes b.length bytes from the specified byte array to this file output stream.
- void close() :- Closes the file Input and Output streams and releases any system resources associated with the Stream.

28/11/12

### File Output Stream :

- A(1)
- zam
- fstream
- means
- inputstream
- zam
- Stream
- outStream
- fstream
- (Input Stream)
- zam
- et
- z
- A file output stream is an output stream from writing data to a file.
- file output stream is meant for writing streams of raw bytes such as image data. For writing streams of characters, consider using FileWriter.
- We can store the data in —
- ① files → by using FileOutputStream API.
  - ② Database → by using JDBC API.

### Constructors:

1. FileOutputStream (String name) :-  
Creates an output file stream to write to the file with the specified name.
2. FileOutputStream (File file) :-  
Creates a file output stream to write to the file represented by the specified file object.
3. FileOutputStream (String name, boolean append) :-  
Creates an output file stream to write to the file with the specified name.  
The value of append may be true or false.  
If append is true → it will override in existing file.  
false → Create a new file.

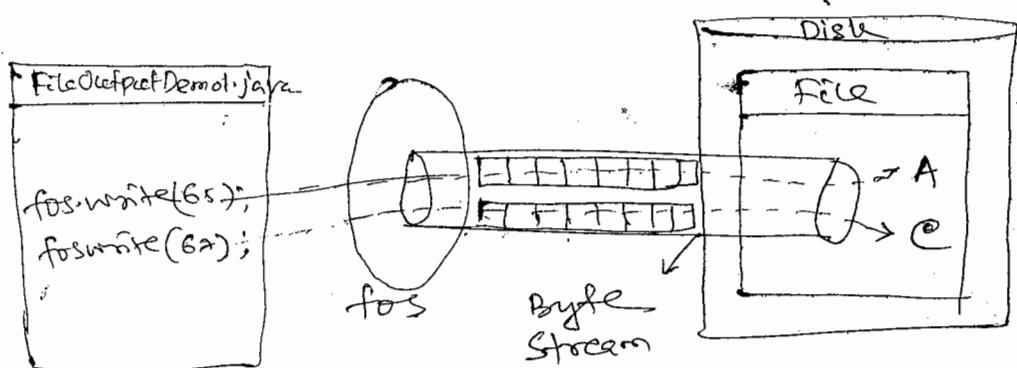
Ex:-

```
import java.io.*;
class FileOutputStreamDemo1
```

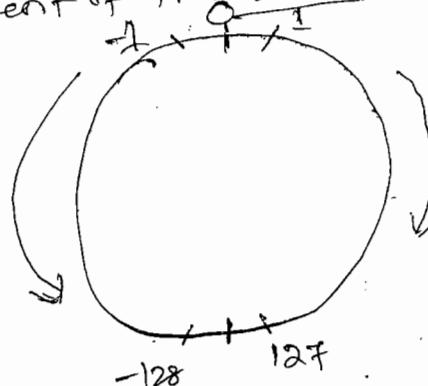
```
{
    public static void main(String args[])
        throws Exception
}
```

```
{    FileOutputStream fos =
        new FileOutputStream("file1");
    fos.write(65);
    fos.write(66);
    fos.write(67);
    fos.close();
}
```

{ As no location  
given so file  
is stored in  
current directory  
i.e.  
"file1".



→ type file1  $\leftarrow$  <sup>Syntax</sup> → type file\_name  $\leftarrow$   
 $\leftarrow$  It is not the java command but it does is  
the DOS prompt command used to show the  
Content of the file.



→ for locating particular location we use  $\leftarrow$   
 $\leftarrow$  d:\file1  $\leftarrow$  store the file in d-drive

Ex:-

```

import java.io.*;
class FileOutputStreamDemo2
{
    public static void main(String args[])
    {
        byte b[] = { 65, 66, 67, 68, 69 };
        FileOutputStream fos =
            new FileOutputStream("file2");
        fos.write(b);
        fos.close();
    }
}

```

### FileInputStream:

- A FileInputStream obtains input bytes from a file in a file system, what files are available depends upon the host environment.
- FileInputStream is meant for reading elements Streams of raw bytes such as image data.
- For reading streams of characters, consider using FileReader.

### Constructors:-

#### 1. FileInputStream (String name)

Creates a FileInputStream by opening a connection to an actual file, the file named by the pathname in the file system.

#### 2. FileInputStream (File file)

Creates a FileInputStream by opening a connection to an actual file, the file named by file object file in the file system.

Ex:  
`import java.io.*;`  
**Class FileInputStream Demo**

{ Psvm(String args[]) throws Exception

{ FileInputStream fes =  
 new FileInputStream("file2");

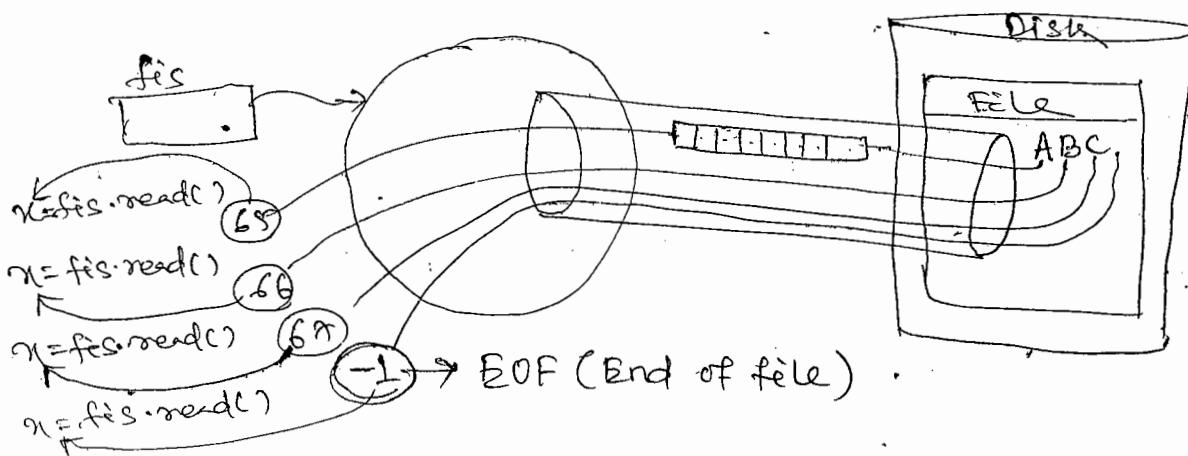
int x;  
 while((x = fes.read()) != -1)

Sop(x);

fes.close();

}

}



- Actually streams are allows us to write or read different date by byte format. So if we want to read or write a float, double etc... data, then ~~it can~~ date may be loss.
- Therefore to avoid the loss of date we come to the filter concept.

1  
2  
3  
4  
5  
6  
7  
8  
9

2

3

4

5

6

7

8

9

Fileentes:-

### ① Data Output Stream:-

A data output stream lets an application write primitive Java data types to an output stream in a portable way. An application can then use a data input stream to read the data back in.

Constructor:-

DataOutputStream (OutputStream out)

Creates a new data output stream to write data to the specified underlying output stream.

Methods:-

1. void writeDouble (double v) :-

Converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the underlying output stream as an 8-byte quantity, high byte first.

2. void writeFloat (float v) :-

Converts the float argument to an int using the floatToIntBits method in class float, and then writes that int value to the underlying output stream as a 4-byte quantity, high byte first.

3. void writeInt (int v) :-

Writes an int to the underlying output stream as four bytes, high byte first.

4. void writeLong (long v) :-

writes a long to the underlying output stream as eight bytes, high byte first.

5. writeChar (char)

6. writeChars (char [ ])

7. writeBoolean (boolean)

8. writeShort (short)

9. writeByte (byte)

Ex:

```
import java.io.*;  
class DataOutputDemo1
```

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

```
    FileOutputStream fos =
```

```
        new FileOutputStream ("data1");
```

```
    DataOutputStream dos =
```

```
        new DataOutputStream (fos);
```

```
    dos.writeInt (101);
```

```
    dos.writeDouble (2.5);
```

```
    dos.writeFloat (2.5f);
```

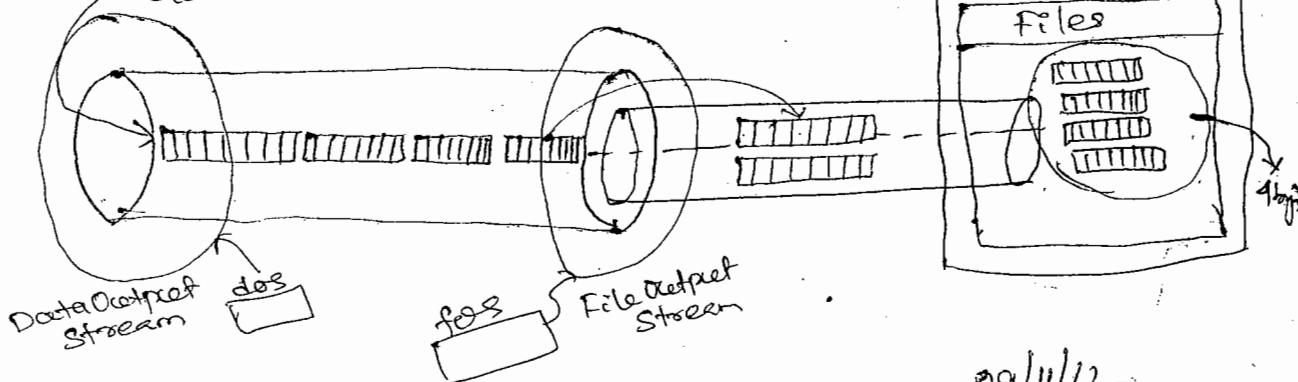
```
    dos.writeBoolean (false);
```

```
    dos.close();
```

```
    fos.close();
```

```
}
```

```
}  
dos.writeFloat(2.5f)
```



29/11/12

Data Input Stream:

- A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application stream or a data output stream to write data that can later be read by a data input stream.
- Data input stream is not necessarily safe for multithreaded access. Thread safety is optional and is the responsibility of users of methods in this class.
- It is a dependent stream, which requires an input stream.

### Constructors :-

`DataInputStream (InputStream in)`

Creates a DataInputStream that uses the specified underlying input stream.

### Methods :-

`boolean readBoolean();` :-

Reads one i/p byte and returns true if that byte is non zero, false if that byte is zero.

`double readDouble();` :-

Reads 8 input values and returns a double value.

`float readFloat();` :-

Reads 4 input bytes and returns a float value.

`int readInt();` :-

Reads 4 bytes and returns a int value.

`String readLine();` :-

Reads the next line of the text from the input stream.

`long readLong();` :-

Reads 8 input bytes and returns a long value.

`short readShort();` :-

Reads 2 input bytes and returns a short value.

Ex:-

```
import java.io.*;
```

```
Class DataInputStreamDemo1
```

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

```
    FileInputStream fis = new FileInputStream("data");
```

```
    DataInputStream dis = new DataInputStream(fis);
```

```
    int x = dis.readInt();
```

```
    double y = dis.readDouble();
```

```
    float z = dis.readFloat();
```

```
    boolean b = dis.readBoolean();
```

```

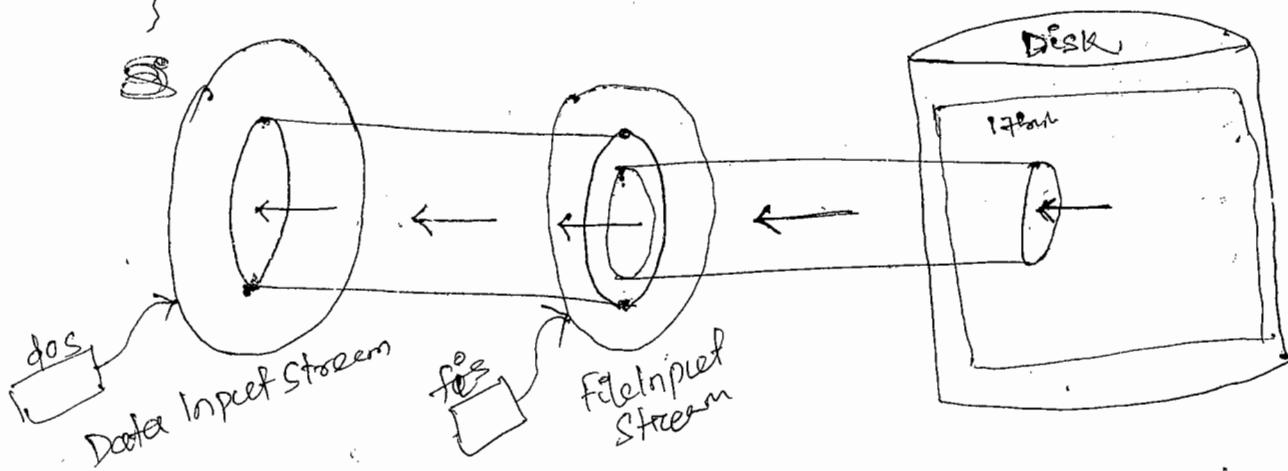
SOP(x); // 101
SOP(y); // 2.5
SOP(z); // 2.55
SOP(b); // feels.

```

```

fes.close();
dis.close(); /* deallocating the
resources */

```



### Serialization:

What is Serialization?

Ability to read or write an object to a stream.

- process of "flattening" an object.

- process of "flattening" an object.

Used to save object to some permanent storage.

- its state should be written in a serialized form

- to a file such that the object can be reconstructed at a later time to that file.

Used to pass on to another object via the output stream class.

- can be sent over the network.

Streams Used for Serialization:

- ObjectOutputStream

- for serializing (flattening an object)

- ObjectInputStream

- for deserializing (reconstructing an object)

## Requirement for Serialization :-

- To allow an object to be Serializable:
  - Its class should implement the Serializable interface.
  - Serializable interface is marker interface.
- Its class should also provide default constructor
  - (<sup>for</sup> constructor with no arguments)
- Serializability is inherited
  - Don't have to implement Serializable on every class.
  - Can just implement Serializable once along the class hierarchy.

## Object Output Stream :-

- An ObjectOutputStream writes primitive datatypes and graphs of Java objects to an OutputStream.
- The objects can be reconstituted or read using an ObjectInputStream.
- Only objects that support java.io.Serializable interface can be written to streams.
- It is dependent stream.

### Constructor :-

ObjectOutputStream (OutputStream out)

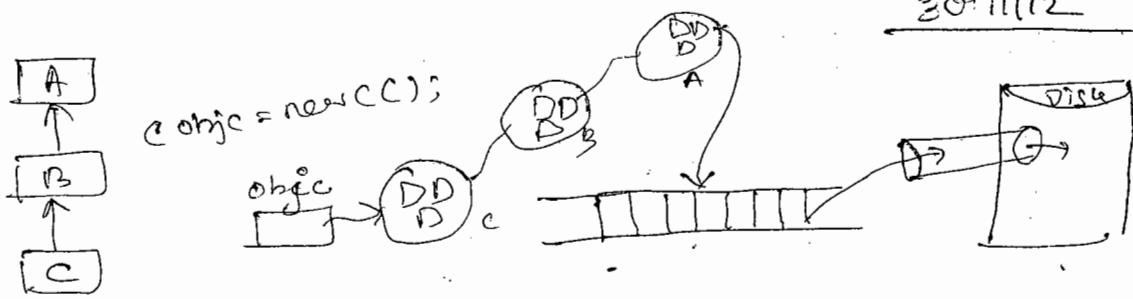
Creates an ObjectOutputStream that writes to the specified output stream.

### Method :-

void writeObject (Object obj)

Write the specified object to the ObjectOutputStream.

20/11/12



### Non Serializable Objects:

- Most Java classes are Serializable.
- Objects of some system level classes are not Serializable.
- Because the data they represent constantly changes.
- Reconstructed object will contain different value anyway.
  - For example, thread running in my Java JVM would be using my system's memory. Persisting it and trying to run it in your JVM would make no sense at all.
- A NotSerializableException is thrown if you try to serialize non-Serializable objects.

Ex:-

```
import java.io.*;  
class A implements Serializable
```

```
{ private int x, y;
```

```
    A(int x, int y)
```

```
{     this.x = x;
```

```
    this.y = y;
```

```
    String getXy()
```

```
{     return x + " " + y;
```

```
}
```

## Class Serialize Demo1

{ psvm("String args[]) throws Exception

{ A obj1 = new A(10, 20);

FileOutputStream fos = new FileOutputStream  
("file3");

ObjectOutputStream oos = new ObjectOutputStream  
(fos);

oos.writeObject(obj1);

oos.close();

fos.close();

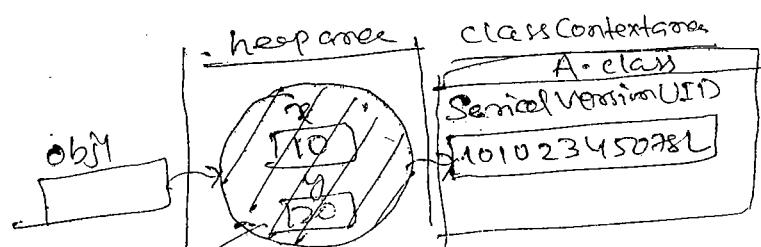
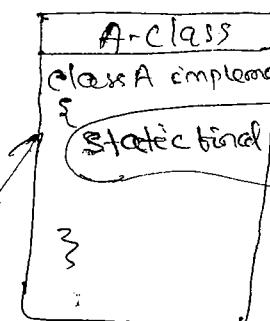
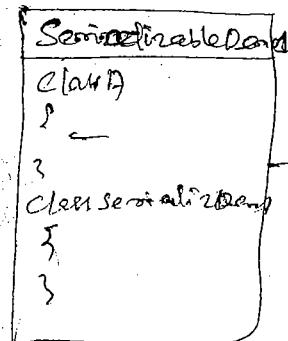
Compiler generates

A-class

class A implements Serializable

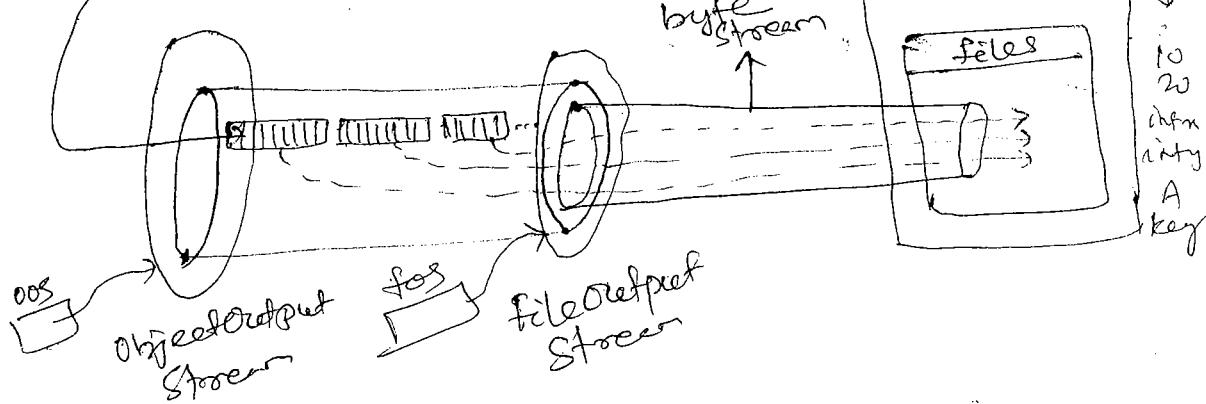
{ static final long serialVersionUID =  
10102345028L;

The key changes  
on every compilation



obj1

class contexts  
A-class  
Serial Version UID  
10102345028L



> type file3 <

What is preserved when an object is Serialized?

- enough information that is needed to reconstruct the object instance at a later time.
- only the object's data are preserved.
- Methods and constructors are not part of the Serialized Stream.
- Class Information is included.

DeSerialization : Reading an object Stream

- Use its readObject method of the ObjectInput Class

```
public class  
public final Object readObject() throws  
IOException, ClassNotFoundException
```

where,

- Obj is the object to be read from the Stream.

- The object type returned should be type casted to the appropriate class name before methods on that class can be executed.

Q) What is Serialization?

→ Serialization is the process of saving the state of an object.

Q) What is deserialization?

→ De-Serialization is the process of restoring the state of an object.

## // De-serialization

```
import java.io.*;
```

```
Class SerializeDemo2
```

```
{ psvm(String args[]) throws Exception
```

```
{
```

```
    FileInputStream fes = new FileInputStream("file3");
```

```
    ObjectInputStream ois =
```

```
        new ObjectInputStream(fes);
```

```
    A obj = ois.readObject();
```

// readObject provides byte stream

// So we typecast it.

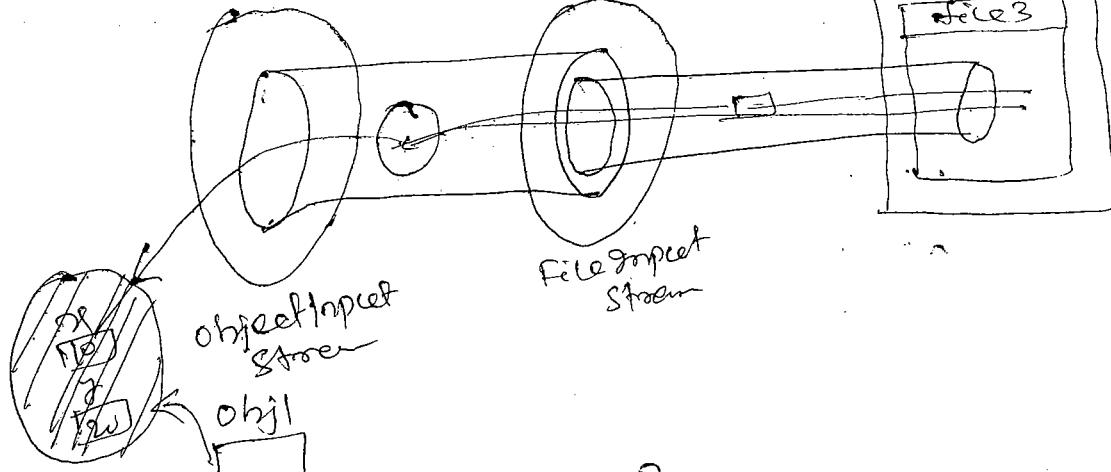
```
    Sop(obj.getXY());
```

```
    ois.close();
```

```
    fes.close();
```

```
}
```

```
}
```



Q) What is transient keyword?

The keyword indicates that the value of member variable does not have to be serialized with the object. When the class will be deserialized, this variable will be initialized with the default value of its data type. (i.e. zeros for integer).

→ We can use transient only for variables not for methods becoz methods are not part of Serialized Stream.

## Syntax :-

transient data-type variable\_name;

Ex:-

import java.io.\*;

Class A implements Serializable

{ int x = 65;

transient int y = 66;

}

Class SerializeDemo3

{ public void main(String args[]) throws Exception

{ A obj1 = new A();

FileOutputStream fos =

new FileOutputStream ("file1");

ObjectOutputStream oos =

new ObjectOutputStream (fos);

oos.writeObject(obj1);

oos.close();

fos.close();

}

// deserialization

import java.io.\*;

Class SerializeDemo4

{ public void main(String args[])

{ FileInputStream fis = new FileInputStream ("file1");

ObjectInputStream ois =

new ObjectInputStream (fis);

A obj1 = (A) ois.readObject();

System.out.println("obj1.x"); // 65

System.out.println("obj1.y"); // 0

## Version Control problem Scenario :-

→ Imagine we create a class, instantiate it, and write it out to an object stream.

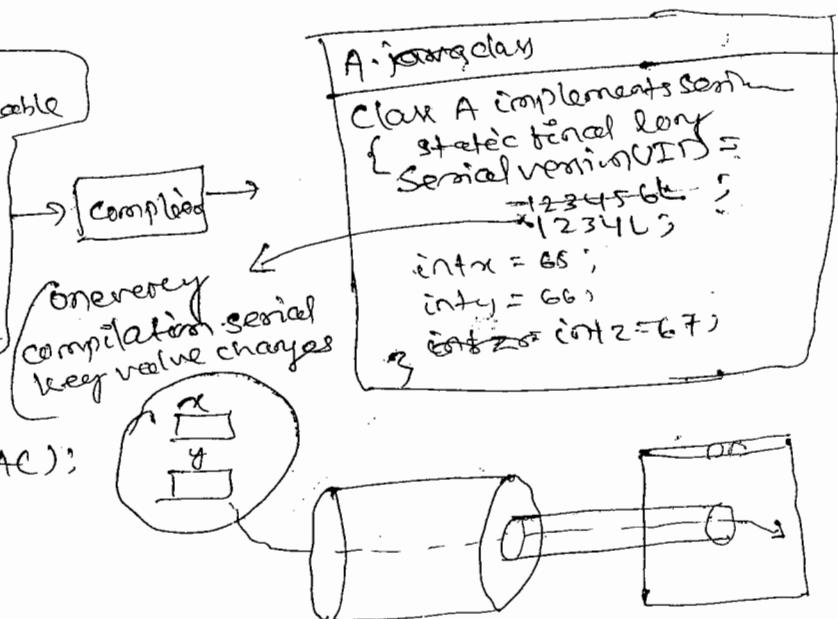
→ That flattened object sits in the file system for some time.

→ Meanwhile, we update the classfile, perhaps adding a new field.

- Meanwhile, adding a new field.
- What happens when we read the flattened object?
  - An Exception will be thrown - specifically, the `java.io.InvalidClassException`.

ex-

A: Jana  
class A implements Serializable  
{  
    int x = 65;  
    int y = 66;  
    int z = 67; } ↗



Unique Identifier

Unique Identifiers.  
↳ Why this Exception occurs & thrown?

→ Because all persistent - capable classes are automatically given a unique identifier.

→ if the identifier of the class doesn't equal the identifier of the flattened object,

Ex:-

```

import java.io.*;
class A implements Serializable
{
    int x = 65;
    int y = 66;
    int z = 67; // ← After storing A into the file,
    // then add a new variable
    // and compile.
}

```

→ javac A.java ←  
 → serialver A ←  
 → We can see the serial version key by entering  
 this command.  
 "SerialVer class-name" ←

Ex:-

```

class Application
{
    public static void main (String args[])
        throws Exception
    {
        A obj1 = new A();
        FileOutputStream fos =
            new FileOutputStream ("file5");
        ObjectOutputStream oos =
            new ObjectOutputStream (fos);
        oos.writeObject (obj1);
        oos.close();
        fos.close();
    }
}

```

// After compiling and adding another variable we  
 want to read the file from the persistence  
 media.

```

import java.io.*;
class Application2
{
    psvm (String args[])
        throws Exception
    {
        FileInputStream fis =
            new FileInputStream ("files");
        ObjectInputStream ois =
            new ObjectInputStream (fis);
        ois.readObject();
        A obj1 = (A) ois.readObject();
        sop (obj1.x);
        sop (obj1.y);
    }
}

```

- 3.3
- This above program display ~~exception~~ exception.
  - In this above program (reversion control), we provide ~~SerialVersionUID~~ SerialVersionUID.
  - To avoid this error, we provide ~~SerialVersionUID~~ explicitly.

Q:-

```

import java.io.*;
class B implements Serializable
{
    static final long serialVersionUID = 123L;
    int x = 65;
    int y = 66;
    int z = 67; ← Adding variable.
}

```

## Buffered Stream

→ There are 2 Buffered Stream.

- 1) Buffered Output Stream
- 2) Buffered Input Stream.

→ The examples shown so far use unbuffered I/O.

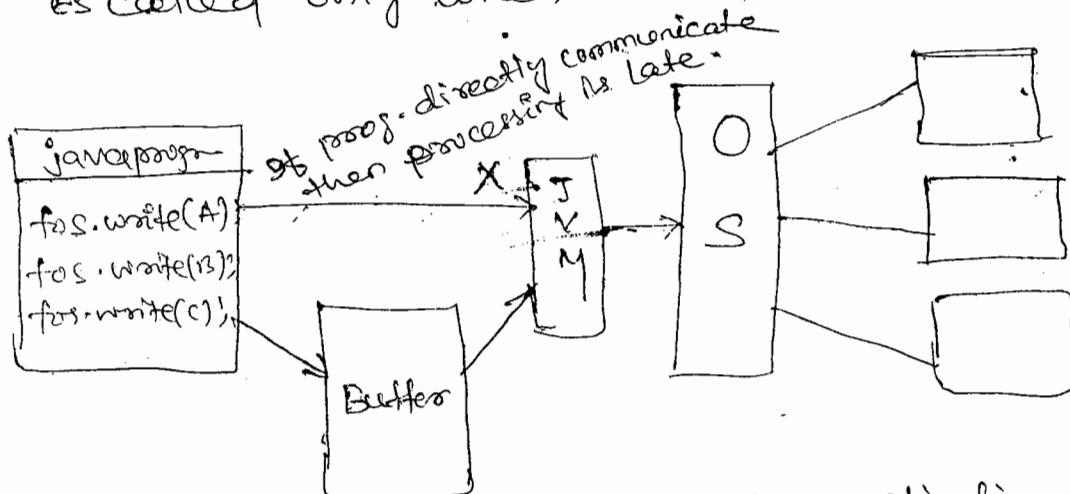
→ This means each read or write request is handled directly by the underlying OS.

→ This can make a program much "effecient", since each such request often triggers disk access, network activity, or some other operations that are relatively expensive.

→ To reduce this kinds of overhead, the Java platform implements buffered I/O streams.

→ Buffered Input streams read data from memory one known as buffer, the native API is called only when the buffer is empty.

→ Similarly the buffer output streams write data to a buffer, and the native output API is called only when the buffer is full.



→ OS is performing following application operation -

- 1) memory management
- 2) I/O oper<sup>n</sup>.
- 3) Process management
- 4) Device management

## ① Buffered Output Stream :-

→ The class implements a buffered Output Stream. By setting up such an output stream, an application can write bytes to the underlying output stream.

### Constructors :-

BufferedOutputStream (OutputStream out)

Creates a new buffered output stream to write data to the specified underlying output stream.

BufferedOutputStream (OutputStream out, int size)

Creates a new buffered output stream to write data to the specified underlying output stream with the specified buffer size.

### Method :-

void flush() :-

flushes this buffered output stream.

Ex :-  
import java.io.\*;

Class BufferDemo

{ psvm (String args [ ]) throws Exception

{ FileOutputStream fos =  
new FileOutputStream ("file6");

BufferedOutputStream bos =  
new BufferedOutputStream (fos);

bos.write (65);

bos.write (66);

bos.write (67);

bos.close();

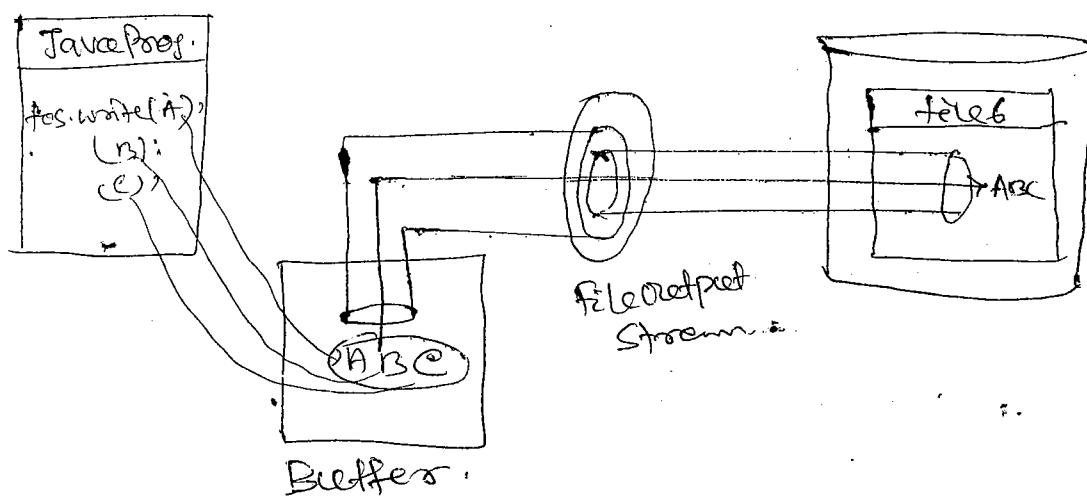
fos.close();

}

}

→ Buffer writes the content to the file by underlying output stream in 3 cases -

- 1) when buffer is full. (Automatic)
- 2) invoking flush() explicitly.
- 3) before closing buffer.



// with size of buffer

import java.io.\*;

class BufferDemo2

{

  public void main (String args [ ]) throws Exception

{

    fileOutput Stream fos =  
    new fileOutput Stream ("file7");

    BufferedOutput Stream bos =

    new BufferedOutput Stream (fos, 2);

    bos.write(65);

    bos.write(66);

    bos.write(67);

    bos.write(68);

    bos.write(69);

    bos.flush();

    bos.close();

    fos.close();

  }

  }

2/12/12

## Buffered Input Stream :-

A buffered input stream adds functionality to another input stream - namely, the ability to buffer the input and to support the mark and reset methods. When the buffered input stream is created, an internal buffer array is created.

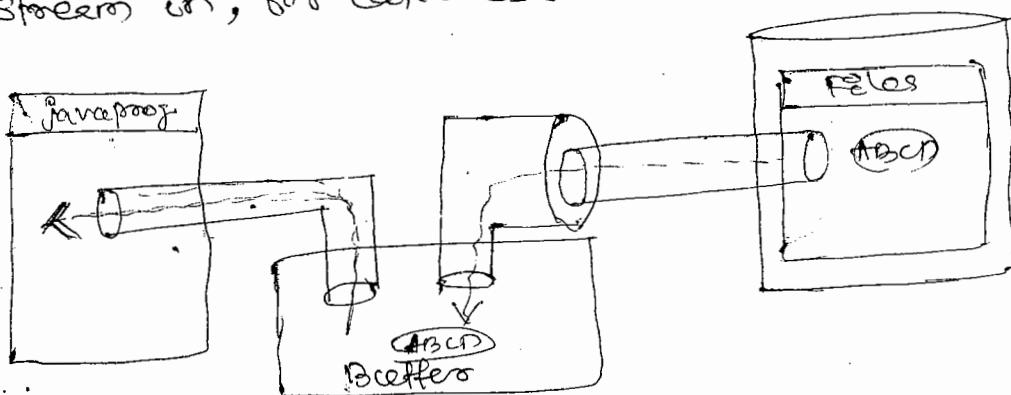
### Constructors :-

BufferedInputStream (InputStream in) :-

Creates a BufferedInputStream and saves its argument the InputStream in, for later use.

BufferedInputStream (InputStream in, int size) :-

Creates a BufferedInputStream with the specified buffer size, and saves its argument, the InputStream in, for later use.



Ex :-

```
import java.io.*;
```

Class BufferDemo3

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

```
{ FileInputStream fes =  
new FileInputStream ("file6");
```

BufferedInputStream bis =

```
new BufferedInputStream ( fes, 2 );
```

```
int x;
```

```
x = bis.read();
```

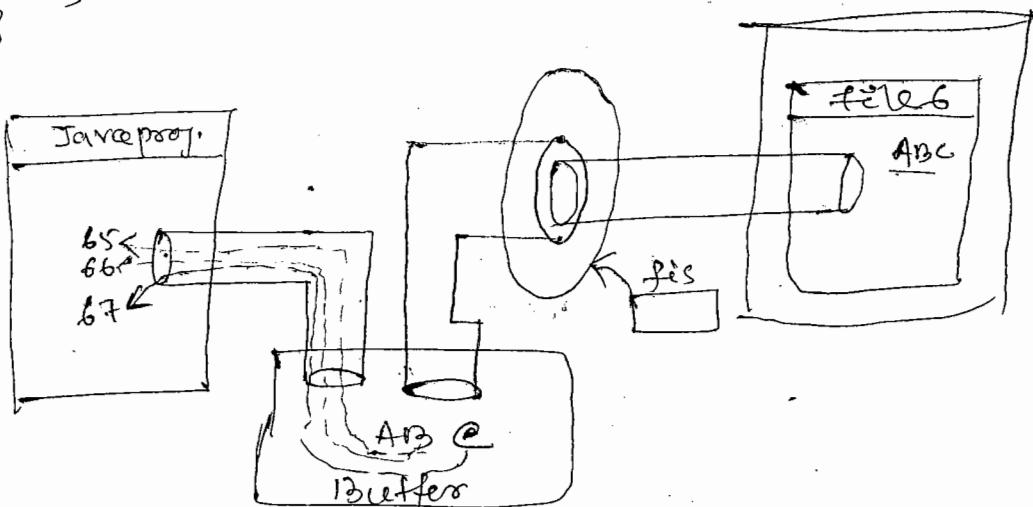
```
SOP(x); → 65
```

```
or  
bis.close();  
fes.close();
```

```

x = bis.read();
System.out.println(x); // → 66
x = bis.read();
bis.close();
System.out.println(x); // → 67
bis.close();
FileOutputStream fos;
}
}

```



### Byte Array Output Stream :

→ This class implements an output stream in which, the data is written into a byte array. The buffer automatically grows as data is written to it. The data can be reterived using "ByteArray" and "toString()".

→ Closing a Byte Array Output Stream has no effect. The methods in this class can be called after the stream has been closed without generating an IIOException.

### Constructor :

Byte Array Output Stream() :

Creates a new byte array output stream.

Byte Array Output Stream(int size) :

Creates a new byte array output stream, with a buffer capacity of the specified size, in bytes.

```

Ex:
import java.io.*;
class ByteArrayOutputDemo {
    { psvm(String args) throws Exception
    {

```

ByteArrayOutputStream = bos =  
new ByteArrayOutputStream(4);

bos.write(65);

bos.write(66);

bos.write(67);

bos.write(68);

bos.write(69);

String s = bos.toString();

SOP(s);

byte b[] = bos.toByteArray();

for (int i=0; i<b.length; i++)

SOP(b[i]);

} 3. ByteArray OutputStream

ByteArray

byte b[] = new

byte(4);

b[0] = 65;

b[1] = 66

b[2] = 67

b[3] = 68

say()

ByteArrayOutputStream = new

new ByteArrayOutputStream(4);

bos.write(65);

bos.write(66);

(67);

(68);

bos.write(69);

garbage collected

→ It is dynamically growable

→ It uses methods for writing  
data in byte array.

→ This is not fixed in size.

→ It uses indexes for writing data.

## ByteArrayInputStream:

- A ByteArrayInputStream contains an internal buffer that contains bytes that may be read from the stream.
- An internal counter keeps track of the next byte to be supplied by the read method.
- Closing a ByteArrayInputStream has no effect. The methods in this class can be called after the stream has been closed without generating an IOException.

## Constructors:

### ByteArrayInputStream (byte[] buf):

Creates a ByteArrayInputStream, so that it uses buf as its buffer array.

### ByteArrayInputStream (byte[] buf, int offset, int length):

Creates a ByteArrayInputStream, so that it uses buf as its buffer array.

By this constructor we can read specific bytes from an array by providing starting index & length.

Ex :-

```
import java.io.*;
```

```
class ByteArrayInputDemo1
```

```
{ public void main (String args [] ) throws Exception
```

```
{ byte b [] = { 65, 66, 67, 68, 69 } ;
```

ByteInputStream bis =

```
new ByteArrayInputStream ( b ) ;
```

```
int x ;
```

```
x = bis.read () ;
```

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

→ 65

```
bis.skip ( 2 ) ; // → skip 2 bytes
```

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

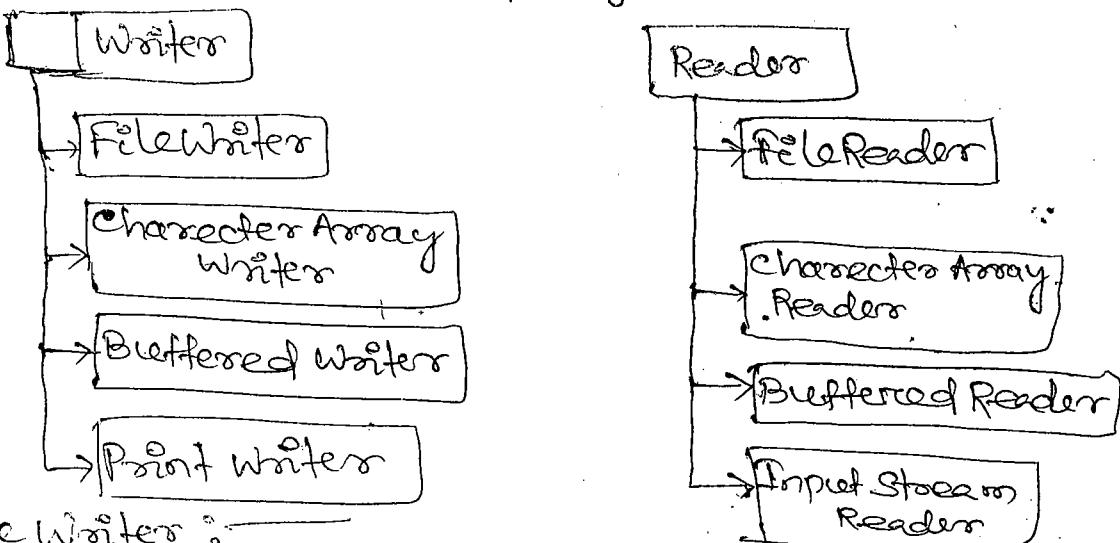
`SOP(x);` → 68  
`bis.reset();` // ← After cursor comes to 1st position i.e. b[0].

→ It allows us to read not changes in array. It only moves cursor.

Ex: `byte b[] = { 65, 66, 67 };`  
`m1(b);`      `void m1(byte[] b)`  
b                  b  
[32char]          [32char]  
In order to avoid this modification we provide ~~ByteStream~~ ByteArray Input Stream.

### Character Streams:

- The Java platform stores character values using encode conventions.
- Character stream I/O automatically translates the internal format to and from the local character set.
- For most applications, I/O with character streams is no more complicated than I/O with byte streams.
- A program that uses character streams in place of byte streams automatically adapts to the local character set.
- All character stream classes are descended from Reader and Writer.
- Character streams are often "wrapped" for byte streams.
- The char. stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes.
- File Reader, for example, uses file input stream, while File Writer uses file output stream.

java.io.packageFile Writers :-

Convenience class for writing character files.  
The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values yourself, construct an `Output Stream Writer` on a `File Output Stream`.

→ It is used to create text file.

Constructors :-

`FileWriter (String filename)` :-

Constructs a `FileWriter` object given a filename.

`FileWriter (File file)` :-

Constructs a `FileWriter` object given a file object.

Methods :-

`void write (char[] charr)`

Writes an array of characters.

`void write (String str)`

Writes a String.

Ex :-

`import java.io.*;`

• Class `FileWriter Demo1`

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

{ `FileWriter fw = new FileWriter ("text1");`

`fw.write ('A');`

`fw.write ("Java");`

`fw.write ("language");`

`fw.close();`

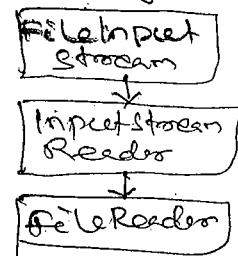
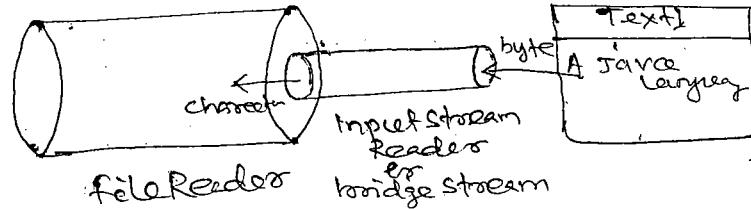
O/P

A Java language

The above programme O/P is textile becoz it is collection of characters.

### File Reader:-

FileReader is meant for reading stream of characters.  
For reading streams of raw bytes, consider using  
FileInputStream.



### Constructor:-

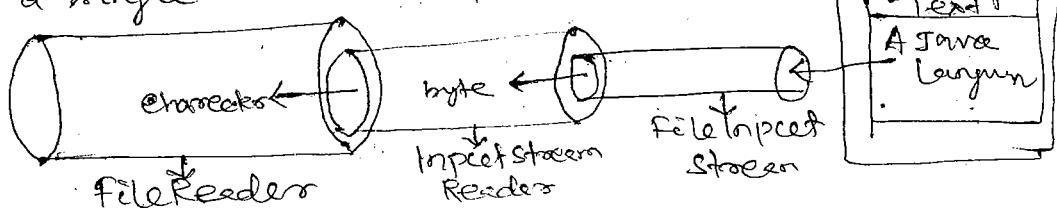
FileReader(String filename)

Creates a new FileReader, given the name of the file to read from.

### method:-

int read()

Reads a single character.



ex:-

```
import java.io.*;  
Class FileReaderDemo1
```

```
{ psvm (String args[]) throws Exception
```

```
{ FileReader fr = new FileReader ("text1");
```

```
int x;
```

```
while ((x = fr.read()) != -1)
```

```
System.out.print ("%c", x);
```

```
fr.close();
```

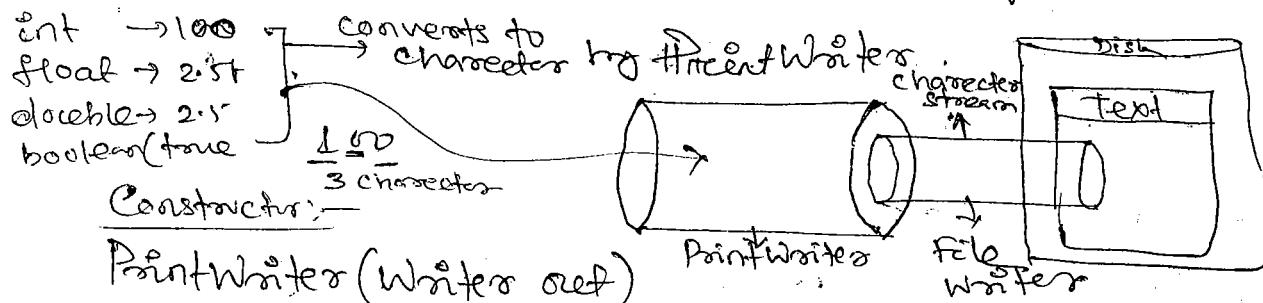
```
} }
```

1)

ye

## ~~PrintWriter~~

- Points formatted representation of object to a text-output stream.
- This class implements all of the print methods found in PrintStream. It doesn't contain any methods for writing raw bytes, for which a program should use unencoded byte streams.



Creates a new PrintWriter, without automatic line flushing.

→ This class provides following methods for writing. These methods are overloaded for writing various types of data.

- 1) print → print(int), print(float), print(double)....
- 2) println → println(int), println(float), ... but a new line.

→ We can convert or write other types of data except characters to Textfile by using PrintWriter.

Ex:-

import java.io.\*;

Class PrintWriterDemo

{

    PSVM (String args[]) throws Exception

    {

        FileWriter fw = new FileWriter ("text2");

        PrintWriter pw = new PrintWriter (fw);

        pw.println (100);

        pw.println (1.5f);

        pw.println (2.5);

        pw.println ("java");

        pw.println (true);

        pw.close();

        fw.close();

}

## Buffered Reader :-

Reads text from character input stream, buffering characters so as to provide for the efficient reading of characters, arrays and lines.

### Constructors :-

BufferedReader (Reader in) :-

Creates a buffering character - input stream that uses a default size input buffer.

BufferedReader (Reader in, int size) :-

Creates a buffering character - input stream that uses an input buffer of the specified size;

### method :-

int read () :-  
Reads a single character.

String readLine () :-  
Reads a line of text.

Ex :-  
import java.io.\*;

class BufferedReaderDemo {

{ osvm (String args) throws Exception

```
    { FileReader fr = new FileReader ("text2");
      BufferedReader br = new BufferedReader (fr);
      int x = Integer.parseInt (br.readLine ());
      float y = Float.parseFloat (br.readLine ());
      double z = Double.parseDouble (br.readLine ());
      String s = br.readLine ();
      boolean b = Boolean.parseBoolean (br.readLine ());
      SOP (x);
      SOP (y);
      SOP (z);
      SOP (s);
      SOP (b);
      br.close ();
      fr.close (); }
```



Input Stream Reader:-

- An Input Stream Reader is a bridge from byte stream to character stream.
- It reads byte and decode them into characters using a specified charset.

Constructor:-

Input Stream Reader (InputStream in) :-

Creates an Input Stream Reader that uses the default charset.

// Reading data from keyboard without using scanner class.

```
import java.io.*;
```

```
Class InputStreamDemo1
```

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

```
    InputStreamReader isr =
```

```
        new InputStreamReader(System.in));
```

```
    BufferedReader br =
```

```
        new BufferedReader(isr));
```

```
    System.out.println("Input name");
```

```
    String name = br.readLine(); // read one line of  
                                   // information.
```

```
    System.out.println("Input Age");
```

```
    int age = Integer.parseInt(br.readLine());
```

```
    System.out.println("Name " + name);
```

```
    System.out.println("Age " + age);
```

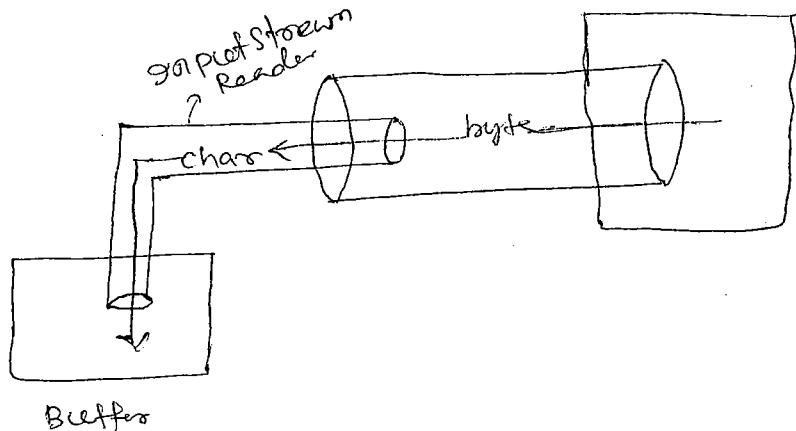
```
    br.close();
```

```
    isr.close();
```

```
}
```

(Omissa)

Preetam Pathak Dr

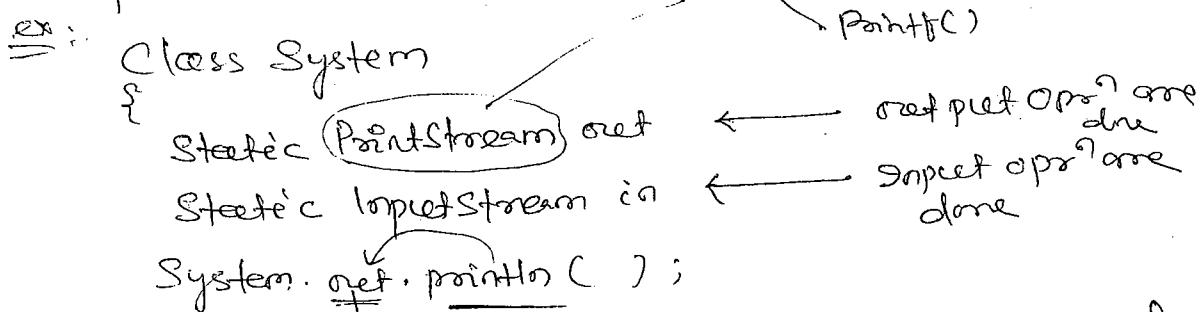


### PointStream :-

A point PointStream adds functionality to another Output Stream, namely the ability to print representations of various data values conveniently.

- Two other features are provided as well.  
Unlike other output streams, a PointStream never throws an IOException.

- PointStream is an OutputStream/Filtered Output Stream.



→ PointStream <sup>reference variable</sup> Constructor :- <sup>is used for designed Userdefined System class.</sup>

PointStream(OutputStream out) :-

Creates a new pointstream.

PointStream(String filename) :-

Creates a new PointStream, without automatic line fleshing, with the specified filename

Methods :-

- It provides the following methods to perform output operations.

These methods are overloaded.

- ① `println()` → `println(int)`, `println(float)`, ...
- ② `print()` → `print(int)`, `print(float)`, ...
- ③ `printf()` → Similar to C `printf`.

//Building user defined System class

```
import java.io.*;
```

```
Class Monitor
```

```
{ static PointStream out;
```

```
static
```

```
{ try {
```

```
out = new PointStream("con."); }
```

```
} catch (Exception e) { }
```

console

```
}
```

```
Class Demo
```

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

```
{
```

```
Monitor.out.println ("Java");
```

```
Monitor.out.println ("F.O");
```

```
}
```

```
}
```

//Writing into file

```
import java.io.*;
```

```
Class PointStream Demo1
```

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

```
{
```

```
FileOutputStream fos =
```

```
new FileOutputStream ("file1");
```

```
PointStream ps = new PointStream (fos);
```

```
ps.println (10);
```

```
ps.println ("xyz");
```

```
ps.println (1.5);
```

```
ps.close();
```

```
fos.close();
```

```
}
```

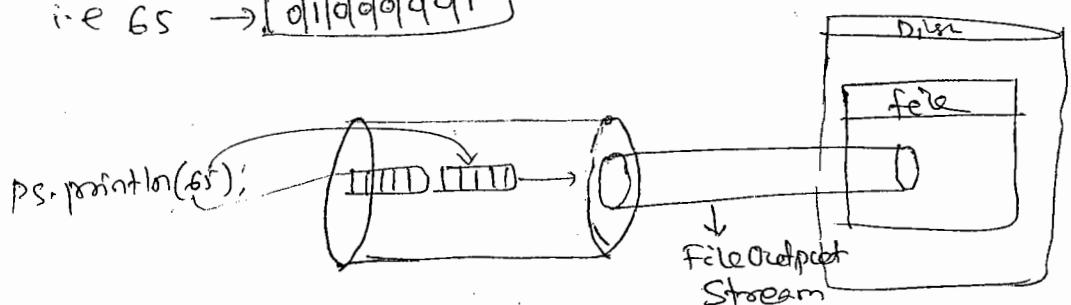
```
}
```

→ PrintStream convert 1 byte each . i.e individual byte

$$(01 \rightarrow \underline{1} \quad \underline{0} \quad \underline{1})$$

→ But if we use FileOutputStream then it convert total into byte .  
i.e 65 → 01000001

$$i.e 65 \rightarrow \boxed{01000001}$$



File : \_\_\_\_\_

num

→ The File class makes it easier to write platform independent code that examines and manipulates files .

→ The name of the class is misleading . File instances represent filename , not files .

→ The file corresponding to the file name might not even exist .

Q Why create a File object for a file that doesn't exist ?

→ A program can use the object to parse a file name . Also the file can be easily created by passing the File object to the constructor of some class , such as FileWriter .

→ If the file doesn't exist , a program can examine its attributes and perform various operations on the file , such as renaming it , deleting it , or changing its permissions .

Constructors : —

File (String pathname) : —

Creates a new file instance by converting the given Pathname String into an abstract pathname .

## File(URI class) :-

Creates a new file instance by converting the given  
file : URI into an abstract pathname.

### Methods :-

1. boolean exists() :-

Tests whether the file or directory denoted by this  
abstract pathname exists.

// Test for file exist or not

```
import java.io.File.*;
```

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

```
class FileDemo1
```

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

```
{ Scanner scan = new Scanner(System.in);
```

```
System.out.println("Input filename");
```

```
String fname = scan.next();
```

```
File f = new File(fname);
```

```
if (f.exists())
```

```
System.out.println("file found");
```

```
else
```

```
System.out.println("file not found");
```

```
}
```

→ Every file has 3 attributes

1) read

2) write

3) execute

2. boolean canExecute() :-

Tests whether the application can execute the  
file denoted by this abstract pathname.

3. boolean canRead() :-

Tests whether the appl' can read the file denoted  
by this abstract pathname.

4. boolean canWrite():—

Tests whether the application can modify the file denoted by this abstract pathname.

Date - 6/12/12

5. boolean isDirectory():—

Tests whether the file denoted by this abstract pathname is a directory.

6. boolean isFile():—

Tests whether the file denoted by this abstract pathname is a normal file.

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

Class fileDemo2

```
{ public static void main(String args[])
{
    Scanner scan = new Scanner(System.in);
    SOP("Input filename");
    String fname = scan.nextLine();
    File f = new File(fname); } // we can do the operation
    // when we create a file object.

    if(f.exists())
    {
        if(f.isFile())
            SOP("It's ordinary file");
        else
            if(f.isDirectory())
                SOP("It is a directory file");
        else
            SOP("filename not found");
    }
}
```

7. String[] list():—

Returns a array of strings naming the files and directories in the directory denoted by this abstract pathname.

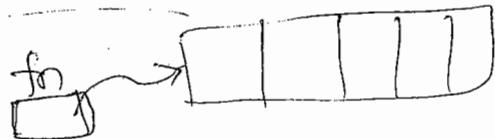
## 8. File [] listFiles() :-

Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.

~~Application to view the content of directory or folder.~~

```
import java.util.*;
import java.io.*;
class fileDemo4
{
    public static void main(String args[])
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Input filename");
        String fname = scan.nextLine();
        File f = new File(fname);
        if(f.exists())
            if(f.isDirectory())
            {
                String fn[] = f.list();
                for(String s:fn)
                    System.out.println(s);
            }
            else
                System.out.println("It is not a directory file");
        else
            System.out.println("file not exists");
    }
}
```

for (String s : fn)  
System.out.println(s);  
Inputfilename  
d:\<u>



group  
// Application to find count no. of files & directories in a given path.

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

Class file Demo 5

```
{ public static void main(String args[])
{
    Scanner scan = new Scanner(System.in);
    System.out.println("Enter path");
    String pName = scan.nextLine();
    File f = new File(pName);
    int fcotent = 0, dcotent = 0;
    if(f.exists())
        if(f.isDirectory())
    {
        File fname[] = f.listFiles();
        for(File fn : fname)
        {
            if(fn.isFile())
                fcotent++;
            else
                dcotent++;
        }
    }
    else
        System.out.println("It is not a directory file");
    else
        System.out.println("File not exist");
    System.out.println("File count: " + fcotent);
    System.out.println("Directory count: " + dcotent);
}}
```

9. String[] list(FilenameFilter filter)

Refers an array of strings naming the files and directories in the directory denoted by this abstract pathname and selected by the

~~File Filter :-~~

filename Filter :-

public interface filenamefilter

→ ~~File~~ filename filter is an interface.

→ Instances of classes that implements this interface are used to filter filenames. These instances are used to filter directory listings in the list method of class File, and by the abstract Window Toolkit's file dialog component.

method :-

boolean accept(File dir, String name) :-

Tests if a specified file should be included in a file list.

Ex:-

import java.util.\*;

import java.io.\*;

Class Myfilenamefilter implements filenamefilter

{ public boolean accept(File d, String name)

{ int i;

boolean b1, b2 = false;

i = d.compareTo("d:\\"); } file f = new File("d:\\");  
i = d.compareTo(f);

b1 = name.equals("package 2");

If (i == 0 & b1 == true)

b2 = true;

return b2;

}

Class FileDemo6

{ public void main(String args[])

{ Scanner scan = new Scanner(System.in);

System.out.println("Input file name");

String fname = scan.next();

File f = new File(fname);

Myfilenamefilter filter = new Myfilenamefilter();

```

if(f.exists())
    if(f.isDirectory())
    {
        String fn[] = f.listFiles();
        for(String s : fn)
            SOP(s);
    }
    else
        SOP("file is not a Directory file");
else
    SOP("file not exist");
}
}

```

#### 10. boolean delete():-

Deletes the file or directory denoted by this abstract pathname.

// Application for deleting a file

```

import java.util.*;
import java.io.*;
class fileDemo
{
    public static void main(String args[])
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Input filename");
        String fname = scan.nextLine();
        File f = new File(fname);
        if(f.isFile())
        if(f.exists())
            if(f.isFile())
            {
                boolean b = f.delete();
                if(b == true)
                    SOP("File deleted");
                else
                    SOP("Error in deletion");
            }
            else
                SOP("It is a directory");
    }
}

```

```
        else
            SOP("filename not exist");
    }
}
```

11. boolean mkdirs():—

Creates the directory named by this abstract pathname.

12. boolean mkdir():—

Creates a directory named by this abstract pathname, including any necessary but nonexistent parent directories.

// App1 for creating a directory.

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

```
import java.io.*;
```

```
Class FileDemo 8
```

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

```
    Scanner scan = new Scanner(System.in);
    SOP("Enter directory name");
```

```
    String dname = scan.nextLine();
```

```
    File f = new File(dname);
```

```
    boolean b = f.mkdir();
```

```
    if(b==true)
```

```
        SOP("Directory is created");
    else
```

```
        SOP("Error in creating directory");
}
```

```
}
```

# Multithreading

Date - 7/2/12

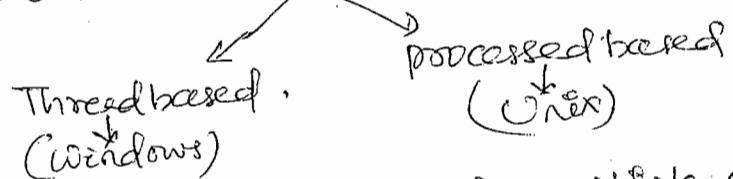
→ Notes

- 1) Single Tasking applications
- 2) Multitasking applications.

## Single Tasking appli :-

An application which performs only one operation at a time is called single tasking application.

## Multitasking application :-



→ Simultaneous execution of multiple operations is called multitasking.

→ Multitasking allows several activities to occur concurrently on the computer.

→ A distinction is usually made between

### 1. Process based multitasking.

2. Thread based multitasking. (multiple oprgns in a single program)

## Advantage of multitasking :-

1. Utilizing CPU Ideal time.

2. Increase the efficiency of program / applications

3. Shortening resources.

Q) What is a process?

- A process is an instance of program.
- Whenever we execute a program, a process is created.
- Process-based multitasking, which allows processes (i.e. programs) to run concurrently on the computer. A familiar example is running the spreadsheet program while also working with the word-processors.
- Thread-based multitasking, which allows parts of the same program to run ~~concurrently~~ concurrently on the computer. A familiar example is a word-processor that is printing and formatting text at the same time.  
This is only feasible, if the two tasks are performed by two independent paths of execution at one time. The two tasks ~~are~~ would correspond to executing parts of the program concurrently.
- The sequence of code executed for each task defines a separate path of execution, and is called a thread (of execution).

Ex:- (Single path exec. / singletasking)

Class Demo

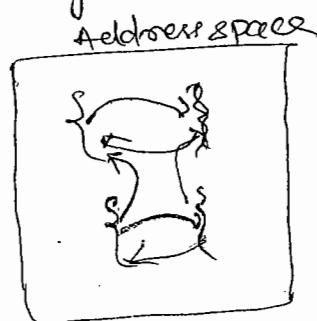
```
{  
    static void m1()  
    {  
        System.out.println("Inside m1()");  
    }  
    static void m2()  
    {  
        System.out.println("Inside m2()");  
    }  
    static void m3()  
    {  
        System.out.println("Inside m3()");  
    }  
    public static void main(String args[])  
    {  
        m1();  
        m2();  
        m3();  
    }  
}
```

\* At a time  
one method  
execute.

- After executing  $m_1()$ ,  $m_2()$  executed.
- In order to do it simultaneously, we provide separate paths or i.e go for Thread based multitasking.

Advantage of thread-based multitasking compared to process-based multitasking :-

1. Thread share the same address space.
2. Context switching between threads is usually less expensive than between processes.
3. Cost of communication between threads is relatively low.



thread is light weight component.

- A thread is an independent sequential path of execution within the program.

→ Many threads can run concurrently within a program. At runtime, threads in a program exists in a common memory space and can, therefore share both data and code, that is, they are lightweight compared to processes. They also share the processes running the program.

Scheduling:-

- Thread Scheduling (done by OS) is the mechanism used to determine how runnable threads are allocated CPU time.
- A thread scheduling mechanism is either

## 1) preemptive

### Preemptive 2) non-preemptive

- In preemptive scheduling, the thread scheduler pauses the running thread to allow different threads to execute.
- The scheduler runs the current thread until it has used up a certain tiny fraction of a second, and then 'preempts' it and resumes another thread for next tiny fraction of a second.

→ ~~Ex~~

### Non-preemptive : —

- In non-preemptive scheduler relies on the running thread to yield control of the CPU, so that other thread may execute. Different operating systems and thread packages implement a variety of scheduling policies.

Q) what is the diff. between timeslicing and preemptive scheduling .

### Preemptive

- In this case, highest priority task continues execution till it enters to a not running state or a higher priority task comes into existence.

### Time slicing (non-preemptive)

- In this case, task continues its execution for a predefined period of time and reenters the pool of ready tasks.

Creating a Thread :-

→ 2 ways are there for creating and starting a thread.

1. Extending the thread class.

2. Implementing the Runnable Interface.

Q) How can we create a thread?

Extending a Thread class :-

→ The subclass extends Thread class.

- The subclass overrides the run() method of Thread class.

→ An object instance of the subclass can then be created.

→ Calling the start() method of the object instance starts the execution of the thread.

- Java runtime starts the execution of the thread by calling run() method of object instance.

Syntax :-

```
class Thread-type-name extends Thread
{
    public void run()
    {
        operation performed by thread;
    }
}
```

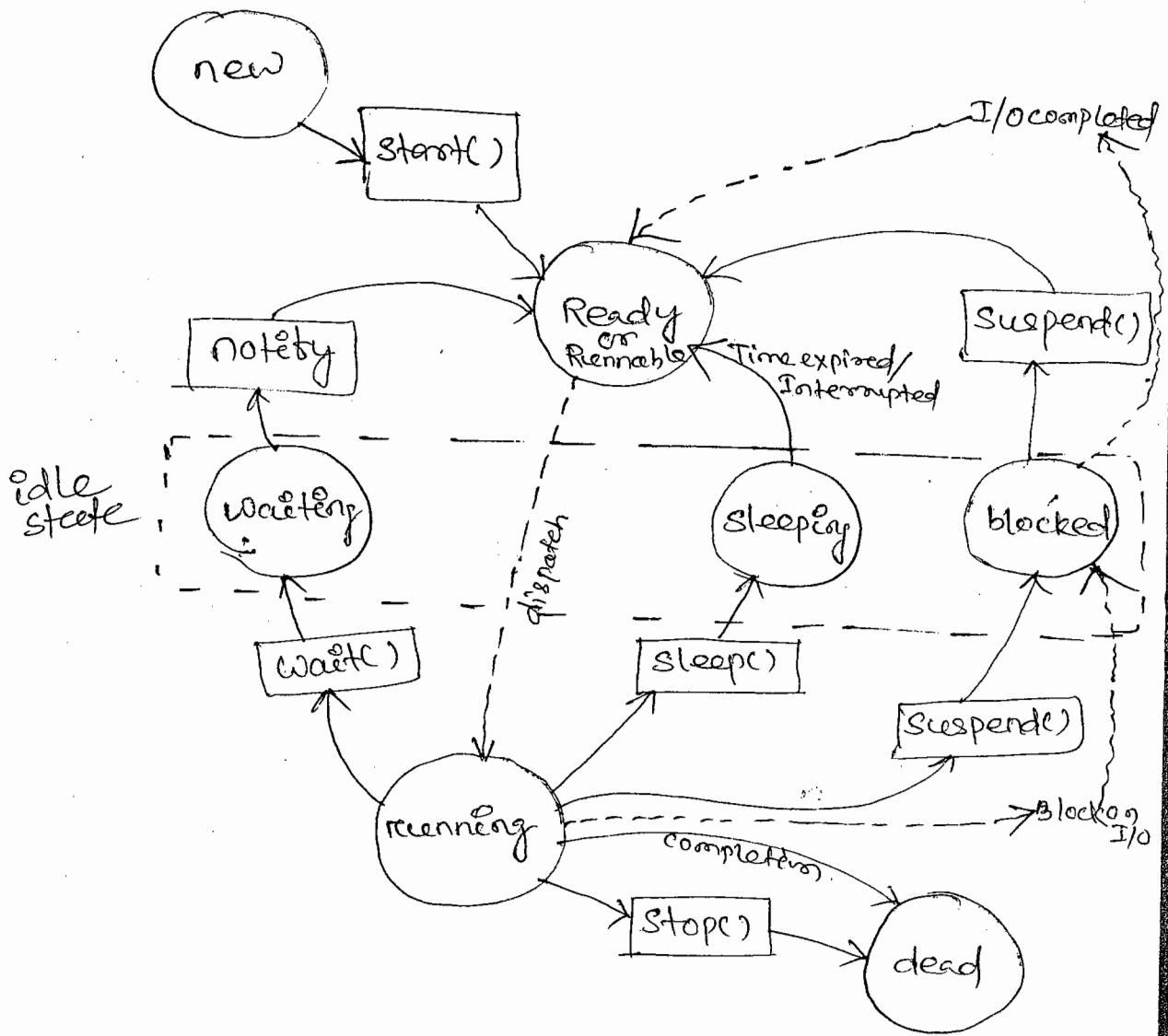
→ run() method is like main() of a Thread.

→ It is like main() of a Thread.

→ Thread execution is done by invoking run() method.

→ Runnable Thread interface just scheduling the thread.

## Life cycle of Thread :-



Q) What is the initial state of a thread when it is created and started?

→ Ready or Renovable state.

Q) What are the different states of the thread?

1. Newborn thread state
2. ready state / renovable state
3. running state
4. idle state
5. dead state.

New born state :-

On creation of thread object it is moved into new born state.

Ready state :-

On a new born thread, when we move invoke start method it is moved into ready state or runnable state, the thread which is in runnable state is scheduled by operating system.

Running state :-

When JVM receives the control of runnable state thread it moves into running state.

Dead state :-

Once the execution of thread is over it is moved into dead state.

Idle state :-

When running thread is interrupted in between it is moved into idle state.

Q) How can we create a thread?

Q) How can we create by extending Thread class can be created by implementing Runnable interface, then we need to override the method public void run().

Q) How does multithreading occur on a computer with a single CPU?

The task scheduler of OS allocates an execution time for multiple tasks. By switching between different executing tasks, it creates the impression that tasks execute sequentially. But actually there is only one task is executed at a time.

Q: What is the difference between creating a thread by extending Thread class and by implementing Runnable interface? Which one should prefer?

→ When creating a thread by extending the Thread class, it is not mandatory to override the run method (if we are not overriding the run method, it is useless), becoz Thread class have already given a default implementation for run method. But if we are implementing Runnable, it is mandatory to override the run method.

→ The preferred way to create a thread is by implementing Runnable interface, becoz it gives loose coupling and provide multiple inheritance.

//WAP to generate messages hello and bye concurrently.

### Single threaded :

```
class Application
{
    static void pointHello()
    {
        for(int i=1; i<=10; i++)
            SOP("Hello");
    }

    static void pointBye()
    {
        for(int i=1; i<=10; i++)
            SOP("Bye");
    }
}
```

```
} // class Application
```

### Multithreaded :

```
class HelloThread extends Thread
{
    public void run()
    {
        for(int i=1; i<=10; i++)
            SOP("Hello");
    }
}
```

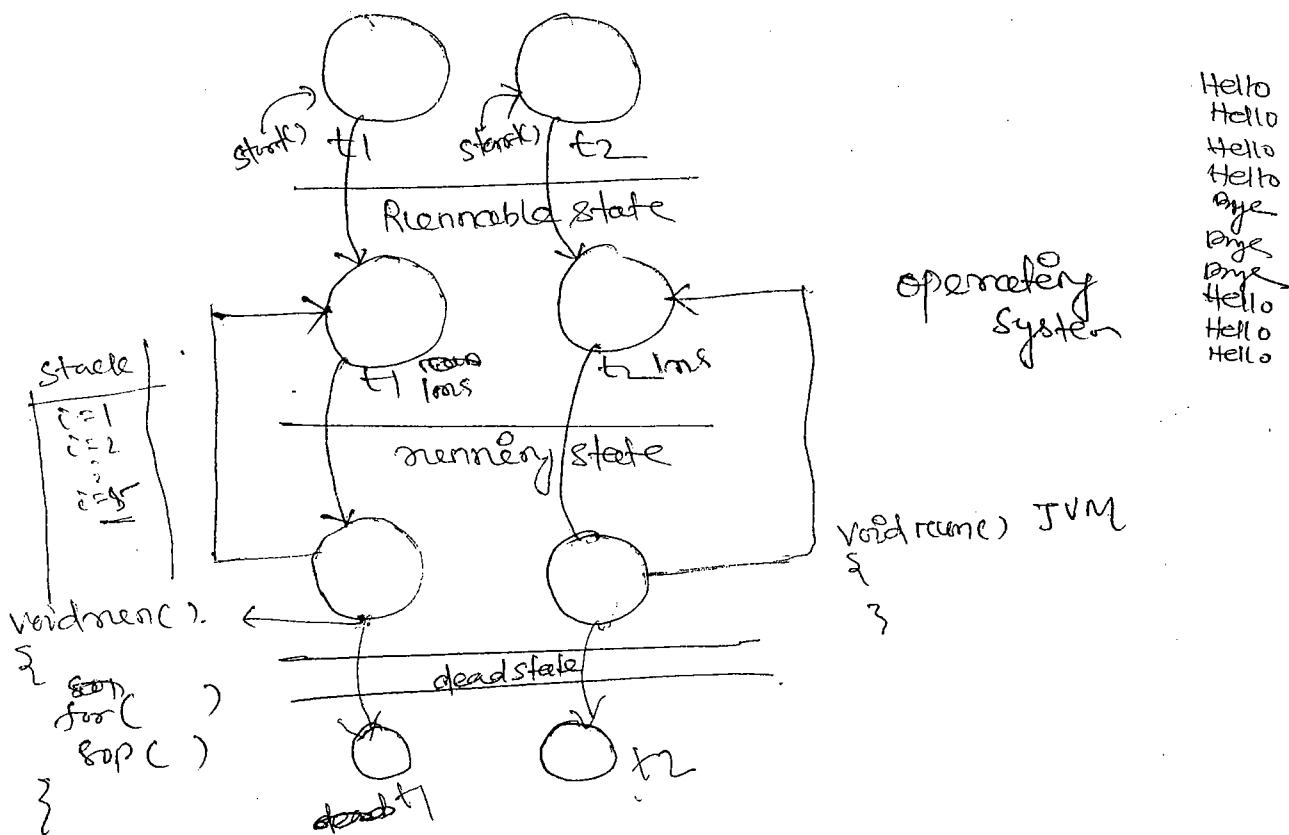
Class ByeThread extends Thread

```
{ public void run() {  
    for( int i = 1; i <= 10; i++ )  
        System.out.println("Bye");  
}
```

Class ThreadDemo1

```
{ public static void main( String args[] ) {  
    HelloThread t1 = new HelloThread();  
    ByeThread t2 = new ByeThread();  
    t1.start();  
    t2.start();  
}
```

Newborn state



## Methods of the Thread class :-

→ The thread class has 8 constructors.

1. Thread() :-

Creates a new thread object.

2. Thread(String name) :-

Creates a thread object with specified name.

3. Thread(Runnable Target) :-

Creates a new thread object based on Runnable object. target refers to the object whose run method is called.

4. Thread(Runnable target, String name) :-

Creates a new thread object with the specified name and based on Runnable object.

// Application which generate Even and odd numbers concurrently.

Class EvenThread extends Thread

{

    EvenThread(String name)

{

    super(name);

}

public void run()

{

    for(int i=1; i<=20; i++)

    { if(i%2==0)

        System.out.println("Even no : " + i);

}

}

}

Class OddThread extends Thread

```
{  
    OddThread (String name)  
    {  
        super(name);  
    }  
    public void run()  
    {  
        for(int i; i<=20; i++)  
        {  
            if(i%2!=0)  
                System.out.println("Odd No: " + i);  
        }  
    }  
}
```

Class ThreadDemo2

```
{  
    public static void main (String args[])  
    {  
        EvenThread t1 = new EvenThread ("Even");  
        OddThread t2 = new OddThread ("Odd");  
        t1.start();  
        t2.start();  
    }  
}
```

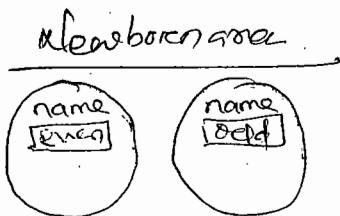
- In the above application 3 threads are created. i.e 1. even thread, 2. odd thread, 3. main thread (default thread).
- main thread is by default provided by JVM.
- Thread class have many properties
  - 1. name (by default given by JVM)
  - 2. priority
  - 3. Id (process id) → it provides distinction b/w more than one thread.
- When we calling called start() method it off OS schedule for 3 threads, then JVM calls the run() method for concurrent operations.

→ If we declared  
`t1 = new EvenThread("even");  
t2 = new OddThread("odd");`

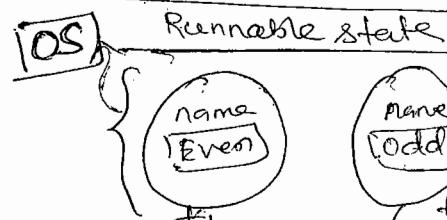
then the application executed sequentially but not simultaneously.

Even Thread t1 =  
`new EvenThread("even"); →`

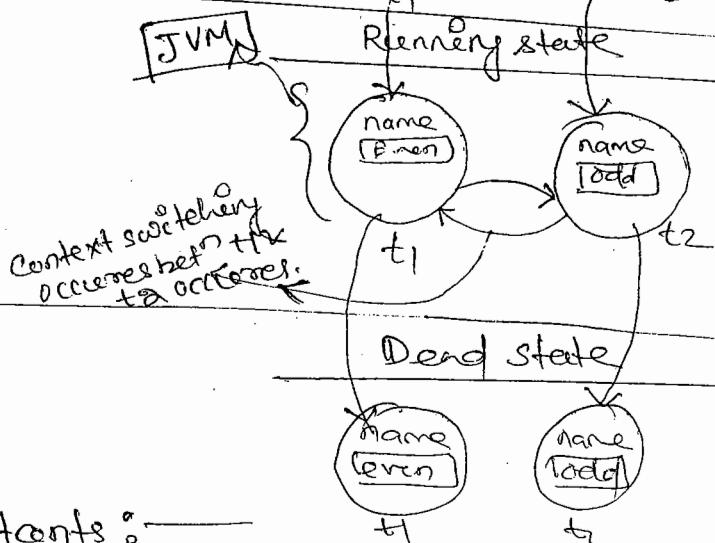
Odd Thread t2 =  
`new OddThread("odd"); →`



`t1.start(); → OS`  
`t2.start(); → OS`



Even Thread → public void run()  
{ }  
Odd Thread → public void run()  
{ }  
{ }



### Thread Class Constants:

→ Contains fields for priority values.

→ It contains 3 constants.

1. MAX\_PRIORITY
2. MIN\_PRIORITY
3. NORM\_PRIORITY

1. public final static int MAX\_PRIORITY; —

→ The maximum priority value 10.

2. public final static int MIN\_PRIORITY; —

The minimum priority value 1.

3. public final static int NORM\_PRIORITY; —

The default priority value, 5.

## Thread class Methods :-

1. public static Thread currentThread() :-

Refers to reference to a thread that is currently running.

2. public final String getName() :-

Refers to the name of the thread.

3. public final void setName(String name) :-

Renamed the thread to the specified argument name, May throw Security Exception.

4. public final int getPriority() :-

Refers to the priority assigned to the thread.

5. public final boolean isAlive() :-

Indicates whether the thread is running or not.

// to find name of main Thread

Class Thread Demo 3

{ public static void main (String args[])

{ Thread t = Thread.currentThread();

SOP(t.getName());

SOP(t.getPriority());

O/P :- main

t.setName("PRITAM");

SOP(t.getName());

}

→ When a JVM starts up, there is a single non-daemon thread (which typically calls the method named main of some designated class)

## Thread Priorities :-

Date - 14/12/12

- Thread priority is an integer value that identifies the relative order in which it should be executed with respect to others. The thread priority value ranging from 0 to 10 and the default value is 5.
- But if a thread have higher priority doesn't means that will execute first. The thread scheduling dependent on OS.

Q) Why Priorities?

- Determine which thread receives CPU Control and gets to be executed first.

Definition :-

- Integer value ranging from 1 to 10.

→ Higher the thread priority - larger chance of

executed first

→ Example :-

- Two threads are ready to run.
- First thread priority of 5, already running
- Second thread = priority of 10, comes in while first thread is running.

Ex:-

Class AlphaThread extends Thread

```
{ public void run()
```

```
{ for(int n=65; n<=90; n++)  
    System.out.println(n); }
```

}

Class NumThread extends Thread

```
{ public void run()
```

```
{ for(int n=65; n<=90; n++)  
    System.out.println(n); }
```

}

## Class Thread Demo 4

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

```
{ AlphaThread t1 = new AlphaThread();
```

```
    NumThread t2 = new NumThread();
```

```
    Sop(t1.getPriority());
```

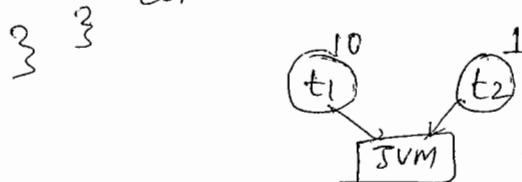
```
    Sop(t2.getPriority());
```

```
    t1.setPriority(Thread.MAX_PRIORITY);
```

```
    t2.setPriority(Thread.MIN_PRIORITY);
```

```
    t1.start();
```

```
    t2.start();
```



### Context Switch:

→ Occurs when a thread snatches the control of CPU from another

- Q When does it occur?
- Running Thread voluntarily relinquishes CPU Control.
  - Running thread is preempted by higher priority Thread.

→ More than one highest priority thread that is ready to run

- Deciding which receives CPU control depends on the OS.
- Windows 95/98/NT uses time-sliced round-robin.
- Solaris: Executing thread should voluntarily relinquish.

## Interrupted Methods:-

→ These methods generate Interrupted Exception, which is a checked exception.

→ The methods are

1. sleep.
2. join
3. yield

### Sleep :-

1. Static void sleep(long millis):-

Causes the currently executed thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of a system timer and scheduler.

2. Static void sleep(long millis, int nanos):-

Causes the currently executed thread to sleep (cease execution) for the specified no. of milliseconds plus specified no. of nanoseconds, subject to the precision and accuracy of a system timer and scheduler.

### Ex:-

Class AlphaThread extends Thread

{ public void run()

{ for(int n=65; n<=90; n++)

{ sleep  
System.out.printf("n%ns = %c", getname(), n);

if(n==80)

{

try

{ sleep(1000);

}

Catch (Exception e)

{

{

3 3 3

## Class AlphaServer

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

{

AlphaThread t1 = new AlphaThread ("client 1");

AlphaThread t2 = new AlphaThread ("client 2");

AlphaThread t3 = new AlphaThread ("client 3");

t1.setName ("client 1");

t2.setName ("client 2");

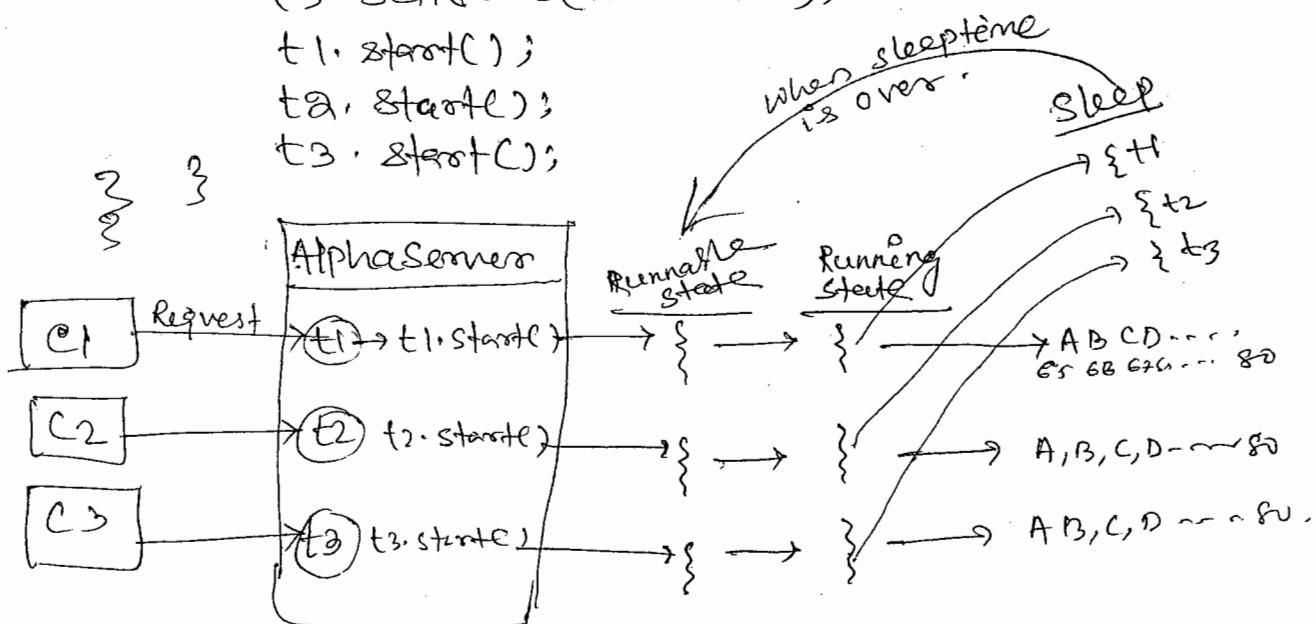
t3.setName ("client 3");

t1.start();

t2.start();

t3.start();

}  
}



2. join :-

(i) public final void join (long millis) throws InterruptedException :-

Wait at most millis milliseconds for this thread to die. A timeout of 0 means to wait forever.

(ii) public final void join (long millis, int nanos)

throws InterruptedException :-

Waits at most millis milliseconds plus nanos nanoseconds for this thread to die.

(iii) public final void join () throws InterruptedException :-

waits for this thread to die.

→ If thread 1 invokes join() on thread 2, thread 1 is join at the end of thread 2 (i.e. until unless thread 2 oprg is executed Thread 1 not started)

→ In join() method, JVM select the thread which thread invoke join()

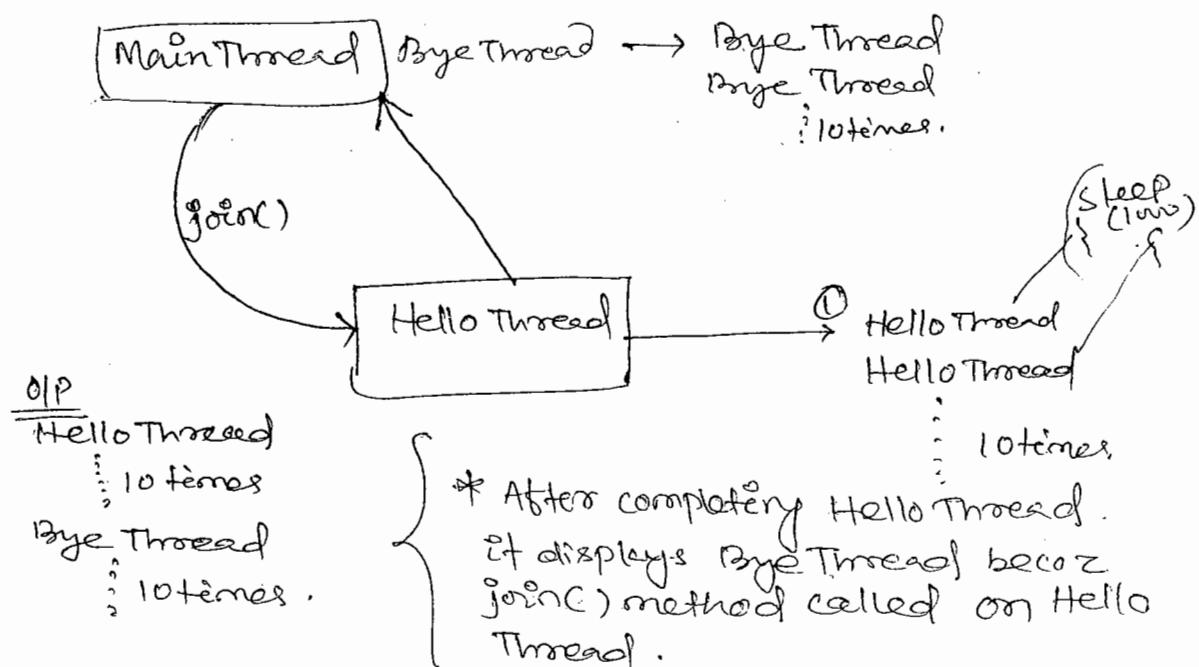
Ex:

Class HelloThread extends Thread

```
{ public void run()
{ for(int i=1; i<=10; i++)
{ System.out.println("Hello Thread");
try{ sleep(1000); }
catch(Exception e) {} }}
```

Class ByeThread

```
{ public void run()
{ HelloThread h = new HelloThread();
h.start();
h.join();
h.start();
for(i=1; i<=10; i++)
System.out.println("Bye Thread");
}}
```



Ex:-

psvm( String args[] )

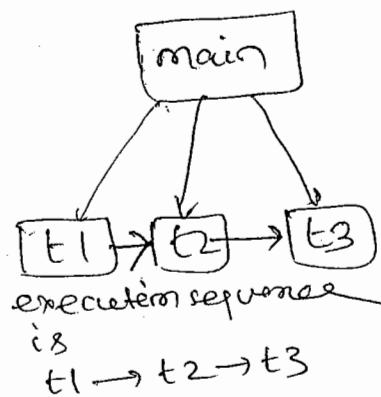
```

    {
        Thread t1 = new Thread();
        Thread t2 = new Thread();
        Thread t3 = new Thread();

        t1.start();
        t2.start();
        t3.start();

        t1.join();
        t2.join();
        t3.join();
    }
}

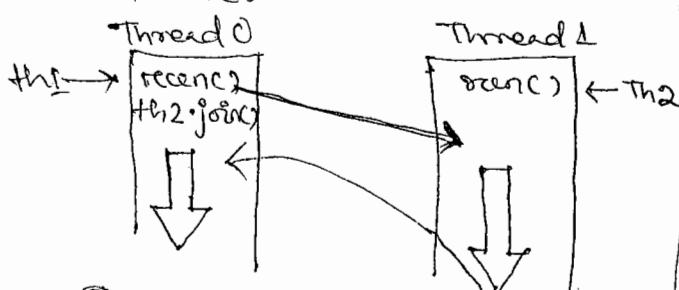
```



Q. What's the diff. bet' no-arg join and parameterized join methods?

join()

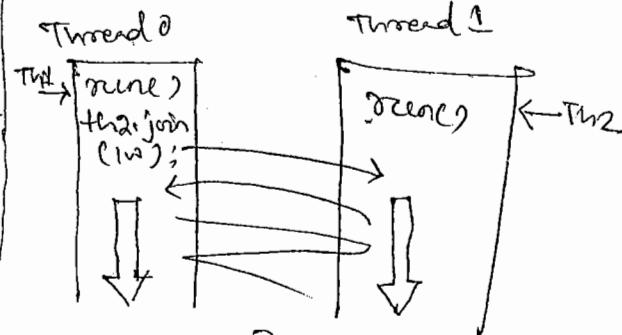
- This method pauses the thread execution until completion of other thread execution.
- If other thread execution is blocked forever, then this thread is also blocked forever.



Q. Diff bet' join(long) & sleep(long)?

join(long)

- This method will not pause the thread execution until completion of other thread execution.
- Its execution is resumed after completion of given time.



sleep(long)

- This method will pause thread execution dependent of other thread execution for the given period of time.
- It will not allow thread to run until the given time slice is completed.

→ It will pause the thread only for a given period of time, if that other thread execution is completed before the given period of time, current thread execution is resumed immediately.

→ Non-static method

→ static method

(3) Yield():

Static void yield():

Causes the currently executing thread object to temporarily pause and allow other threads to execute.

Ex:-

Class NumThread extends Thread

{ public void run()

{ for(int i=1; i<=10; i++)

{ System.out.println("i = " + i);

if (i == 5)

{

try

{ yield();

}

catch (Exception e)

{}

}

}

Class ThreadDemo

{

public static void main(String args[])

{

NumThread t1 = new NumThread();

NumThread t2 = new NumThread();

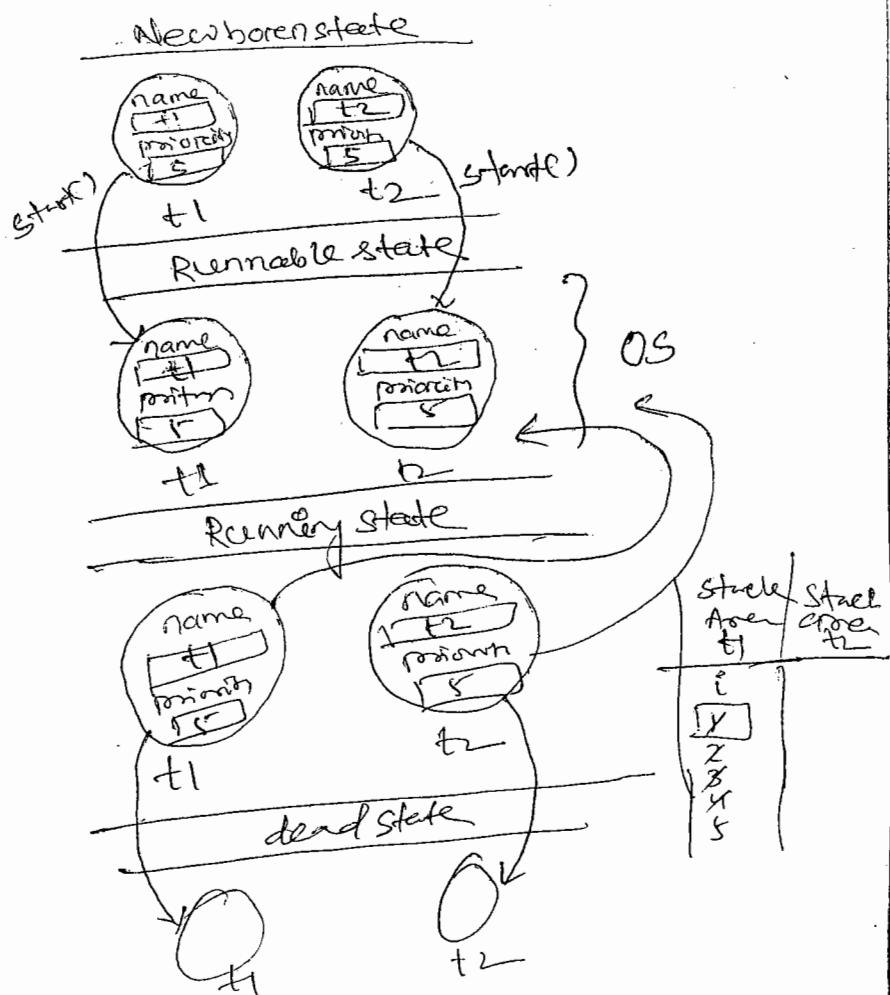
t1.setName("t1");

t2.setName("t2");

t1.start();

{ }

t2.start();



Sharing Resources: →  
 A resource can be an object or class which  
 can be used by more than one thread.

Ex:-

Class Pointer

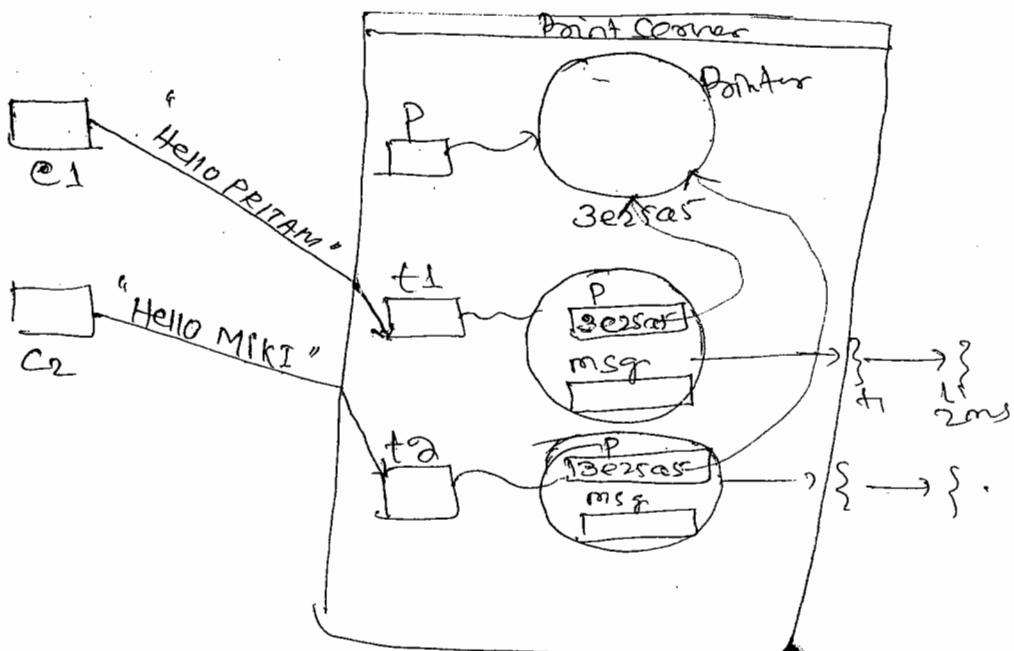
```

  void print(String msg)
  {
    for(int i=1; i<=10; i++)
      sop(msg);
  }
  }
```

3

Class PointThread extends Thread

```
{  
    Pointer p;  
    PointThread(Pointer p, String msg)  
    {  
        this.p = p;  
        this.msg = msg;  
    }  
    public void run()  
    {  
        p.print(msg);  
    }  
}  
  
class PointServer  
{  
    public void main(String args[]){  
        Pointer p = new Pointer();  
        PointThread t1 = new PointThread(p,  
                                         "Hello PRITAM");  
        PointThread t2 = new PointThread(p,  
                                         "Hello MIKI");  
        t1.start();  
        t2.start();  
    }  
}
```



→ Both msg are randomly pointed on object, before of OS

- If more than one thread operate on shared resource, when one thread operate on shared resource the other thread should not allow to perform operation.
- If it allows to perform operation on shared resource it leads to logical errors.
- We can avoid this errors by using Locking mechanism.

### Locks:

- It is a process of preventing a shared resource thread performing operation on shared resource when another thread performs operation.
- There are 2 types of Locking
  1. Object Locking.
  2. Class Locking.
- A thread can acquire the lock on object or class by using —
  1. Synchronized methods.
  2. Synchronized blocks / Synchronized statement.

- Q: What is Synchronization & why it is important?
- With respect to multithreading, Synchronization is the capability to control the access of multiple threads to shared resources.
  - Without Synchronization, it is possible for one thread to modify a shared object while another thread is <sup>in the</sup> process of using or updating that object's value. This often leads to significant errors.

Q) Diff' bet<sup>n</sup> yield() & sleep() ?

yield()

→ When object invokes yield(), it reenters to ready state.

sleep()

→ When object invokes sleep() method, it enters to idle state.

Synchronized methods :—

- Synchronized is a modifier or keyword used to define a method.
- A method declared with synchronized modifier is called synchronized method.
- If thread invoke synchronized method it acq lock on object.
- It allocates to develop thread safe applications.

Syntax :-  
Synchronized return-type method-name (parameters)

{  
    Statements ;

}

ex :-

Class Printer

{

    Synchronized void print (String msg)

    {  
        for (int i=1; i<=10; i++)  
            SOP (msg);

}

Class PointThread extends Thread

{

    Printer p;

    String msg;

    PointThread (Printer p, String msg)

    {  
        this.p = p;

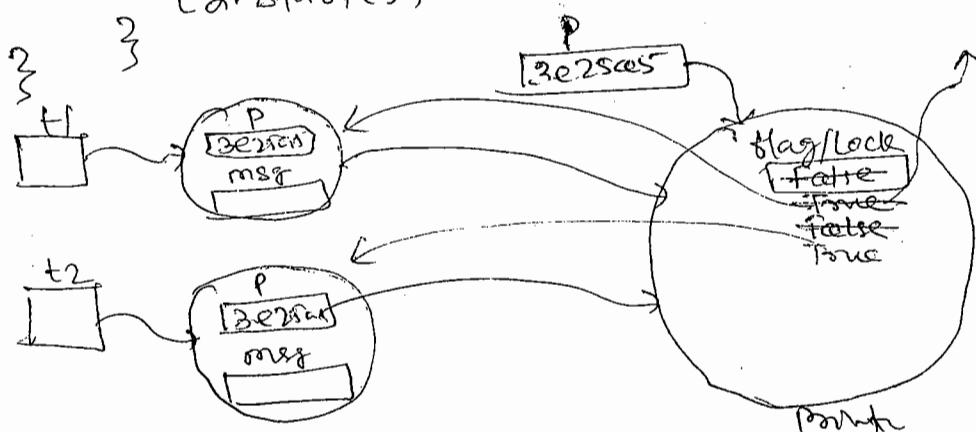
    }  
        this.msg = msg;

```

public void mren() {
    p.print(msg);
}

class PointServer {
    public void main(String args[]) {
        Pointer p = new Pointer();
        PointThread t1 = new PointThread(p, "Hello Client");
        PointThread t2 = new PointThread(p, "Hello Client 2");
        t1.start();
        t2.start();
    }
}

```



→ Default value of lock is false, when we call `t1.start()` it invokes `mren()` method.

Date - 13/12/12

Q: What are the differences b/w synchronized block & synchronized method?

→ Synchronized blocks place locks for the specified block whereas synchronized methods place locks for the entire block method.

Q: What are synchronized methods & synchronized statements?

→ Synchronized() method are methods that are declared with the keyword synchronized.

→ A thread executes the synchronized method only after it has acquired the lock for the method's object or class.

→ Synchronized statements are similar to synchronized() method, it is a block of code declared with synchronized keyword.

Synchronization

→ A synchronized statement can be executed only after a thread has acquired the lock for the object or class referenced in the synchronized statement.

Syntax:-

{ synchronized (reference)  
    statements ; }

Ex:-

Class Bus

{ private int seats ;

    Bus() {

        seats = 10 ;

    }

    Synchronized void reverse(int n) {

        if (seats < n) {

            System.out.println("seats not available");

        else {

            for (int i = 1; i <= n; i++) {

                seats = seats - 1;

            } System.out.println("seats are available:" + seats);

    }

    Bus b;

    int n;

    ReservationThread (Bus b, int n)

    { this.b = b;

        this.n = n;

    }

    public void run()

    { b.reverse(n);

    }

Class ReservationServer

{

    public static void main (String args[]) {

        Bus b1 = new Bus();

        ReservationThread t1 = new ReservationThread (b1, 5);

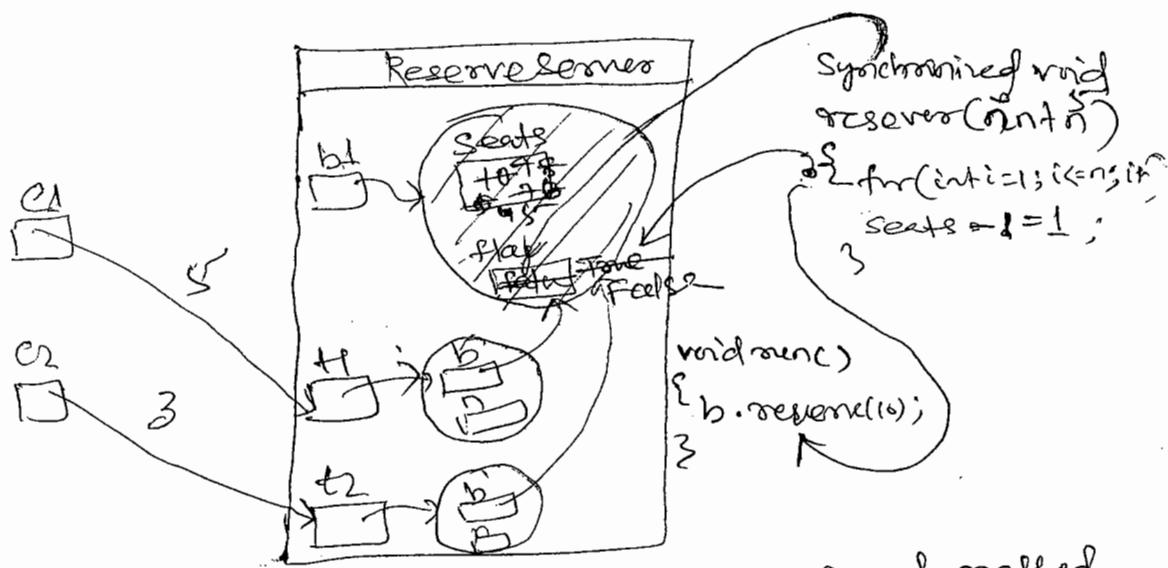
        ReservationThread t2 = new ReservationThread (b1, 3);

        t1.start();

        t2.start();

}

→ Note! → If the method is declared as synchronized then that method's complete logic is executed in sequence by multiple threads.



→ When `main()` invokes the synchronized method the lock/flag becomes true. When synchronized method returns to `main()`, then again lock/flag becomes false.

// Application for synchronized block.

class Account

```

    {
        private int accno;
        private float balance;
        Account (int accno, float balance)
    }
    
```

```

    {
        this.accno = accno;
        this.balance = balance;
    }
    
```

}

void withdraw (float amt)

```

    {
        if(amt > balance)
            System.out.println("Balance not available");
        else
    }
    
```

```

} Class TransactionThread extends Thread
{
    Account acc;
    float tamt;
    TransactionThread (Account acc, float tamt)
    {
        this.acc = acc;
        this.tamt = tamt;
    }
    public void run()
    {
        acc.withdraw(tamt);
    }
}

```

```

> Class TransactionServer
{
    public static void main(String[] args) {
        Account acc1 = new Account(101, 5000f);
        TransactionThread t1 =
            new TransactionThread(acc1, 3000f);
        TransactionThread t2 =
            new TransactionThread(acc1, 2000f);
        t1.start();
        t2.start();
    }
}

```

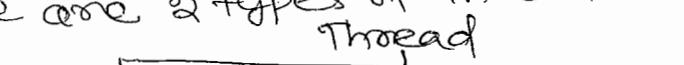
Note.  
→ If we declare a block as synchronized, only the statements written inside the block are executed sequentially not complete method logic.

### Note :-

- It's synchronized
- If synchronized method is static it acquire lock on class.
- If synchronized method is non-static it acquire lock on object.

### Types of Thread :-

- There are 2 types of thread.



- Garbage collector is daemon → Every thread is created as thread. Daemon threads are non-daemon thread, bcoz created for ~~for~~ provide main thread is non-daemon.

→

### Daemon Threads :-

#### Q) What is Daemon Threads?

- A daemon threads are sometimes called "Service or background" threads.
- These are threads are generally run at low priority and provide a basic service to a program when activity on a machine is reduced.
- An example of daemon thread that continuously running is the garbage collector thread.
- The JVM exists, whenever all non-daemon threads are completed, which means that all daemon threads are automatically stopped.
- To make a thread as a daemon thread in Java we write —  
myThread.setDaemon(true);

- The JVM always has a main thread as default, and this thread is always non-daemon.

- The user threads are created from the main thread, and by default they are non-daemon.
- If we want to make a user created thread to be daemon (i.e. stops when the main thread stops) use —

setDaemon(true);

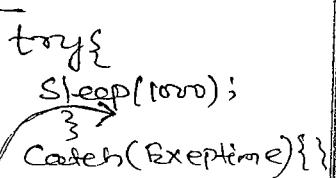
Ex:-

Class NumThread extends Thread

{ public void run()

{ for(int i=1; i<=10; i++)

{ System.out.println("Inside num thread: " + i); }



Class MainThread

{ public static void main(String args[])

{ NumThread t1 = new NumThread();

t1.setDaemon(true);

t1.start();

for(int i=1; i<=10; i++)

System.out.println("Inside main thread: " + i);

Date - 1A (12/12)

### Inter-Thread Communication:

- Communicating one thread with another thread is called "interthread communication".
- The process of executing threads in sequence with communication is called "inter thread communication"
- We develop this concept when two different dependent task want to be executed contiguously in sequence one after another by two different threads.
- To develop this thread communication application we must use below 3 methods —
  - wait()
  - notify()
  - notifyAll()

→ notifyAll() method is similar to notify() but notifyAll() is called when there are multiple waiting threads are available.

## Inter-Thread Communication & Methods Box

### Object class :-

1. public final void wait() :-

Causes this thread to wait until some other thread calls the notify or notifyAll method on this object. May throw InterruptedException.

2. public final void notify() :-

wakes up a thread that called the wait method on the same object.

3. public final void notifyAll() :-

wakes up all threads that called the wait() on the same object.

Q) what does a wait() method do?

It causes current thread to wait until either another thread invokes notify or notifyAll method of the current object; or a specified amount of time has elapsed.

Q) What is the diff. b/w sleep() and wait() method?

### Sleep()

→ The code sleep(millis) puts the thread for a particular time (milliseconds).

→ The method sleep() defines in the class Thread.

→ If a thread is in sleep() method after sleep time over the thread wakes up.

### wait()

→ The code wait(millis) causes wait upto particular time given as parameter argument.

→ The method wait() defines in the class Object.

→ If a thread is in sleep() → The wait()  
method after sleep time over the thread wakes up after getting the call to notify() and notifyAll().

## Wait() method of Object class:-

- Wait() method causes a thread to release the lock it is holding on an object, allowing another thread to run.
- wait() method is defined in the Object class.
- wait() can only be invoked from within synchronized code.
- It should always be wrapped in a try block as it throws IOExceptions.
- wait() can only be invoked by the thread that owns lock on the object.
- When wait() is called, the thread becomes disabled for scheduling and lies dormant until one of 4 things
  - another thread invokes the notify() for this object and the scheduler arbitrarily chooses to run the thread.
  - another thread invokes the notifyAll() method for this object.
  - another thread interrupts this thread.
  - the specified wait time elapses.
- When one of the above occurs, the thread becomes re-available to the thread scheduler and completes for a lock on the object.
- Once it regains the lock on the object, everything resumes as if no suspension had occurred.

Ex:-

Class ScmThread extends Thread

```
{  
    int total;  
    public void run()  
    {  
        synchronized (this)  
        {  
            for (int i=1; i<=100; i++)  
            {  
                total = total + i;  
                notify();  
            }  
        }  
    }  
}
```

// notify(); → notify() must call from  
} synchronized block.

} // class

Class Main Thread

{ psvm(String args[]) throws Exception

{ SumThread t2 = new SumThread();

t2.start();

Synchronized (t2)

{ t2.wait();

SOP(t2.total);

}

}

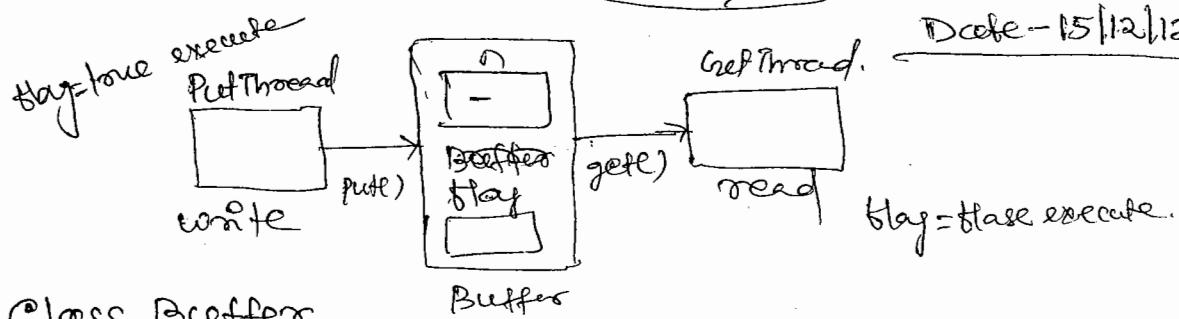
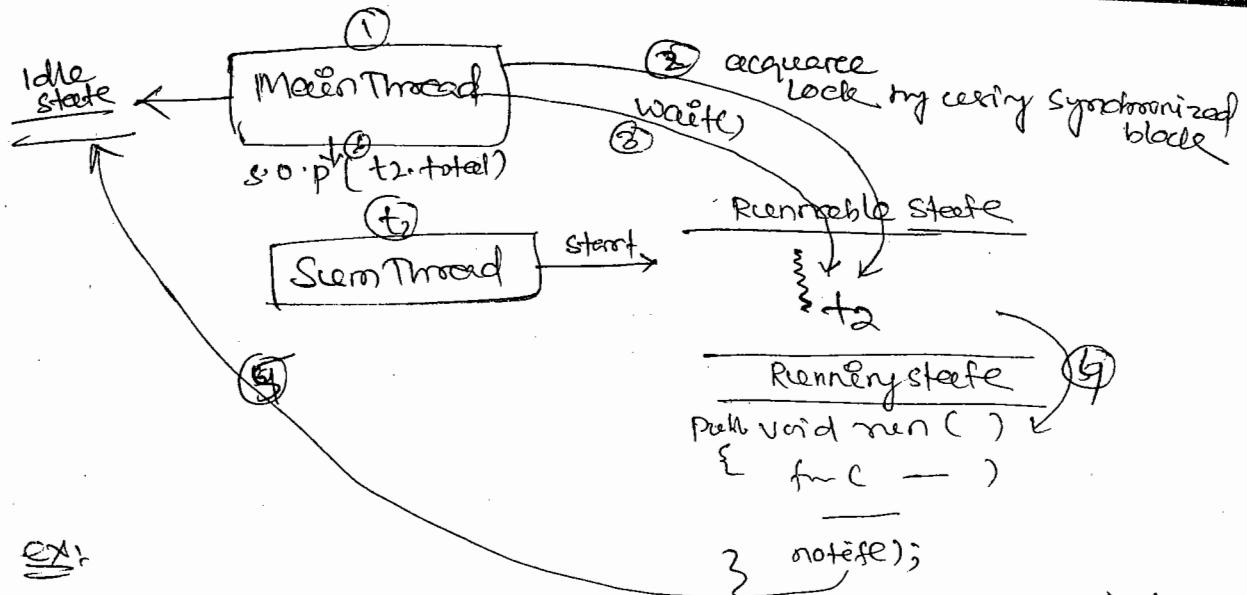
3 }  
3 }  
notify() method :-

→ Wakes up a single thread that is waiting  
on this object's monitor

- If any threads are waiting on this object, one of them is chosen to be awoken
- The choice is arbitrary and occurs at the discretion of the implementation.

→ Can only be used within the synchronized code.

→ The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object.



Class Buffer

```

    {
        int n;
        boolean flag;
        Synchronized void put (int i) throws Exception
    }

```

```

    {
        if (flag == false)
            wait();
        n = i;
        flag = true;
        wake();
    }

```

```

    else
        noteby();
    }

```

```

    Synchronized int get() throws Exception
    {

```

```

        if (flag == false)
            wait();
        flag = false;
    }

```

```

        wake();
        noteby();
        retearen n;
    }
}

```

Class PutThread extends Thread

```
{  
    Buffer b;  
    PutThread(Buffer b)  
    {  
        this.b = b;  
    }  
    public void run()  
    {  
        for(int i=1; i<=10; i++)  
        {  
            try  
            {  
                b.put(i);  
            }  
            catch(Exception e) {}  
        }  
    }  
}
```

Class GetThread extends Thread

```
{  
    Buffer b;  
    GetThread(Buffer b)  
    {  
        this.b = b;  
    }  
    public void run()  
    {  
        for(int i=1; i<=10; i++)  
        {  
            try  
            {  
                int x = b.get();  
                for(int i=1; i<=10; i++)  
                {  
                    int x = b.get();  
                    System.out.println("x=" + x);  
                }  
            }  
            catch(Exception e) {}  
        }  
    }  
}
```

## Class ThreadDemo11

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

```
{ Buffer b = new Buffer ();
```

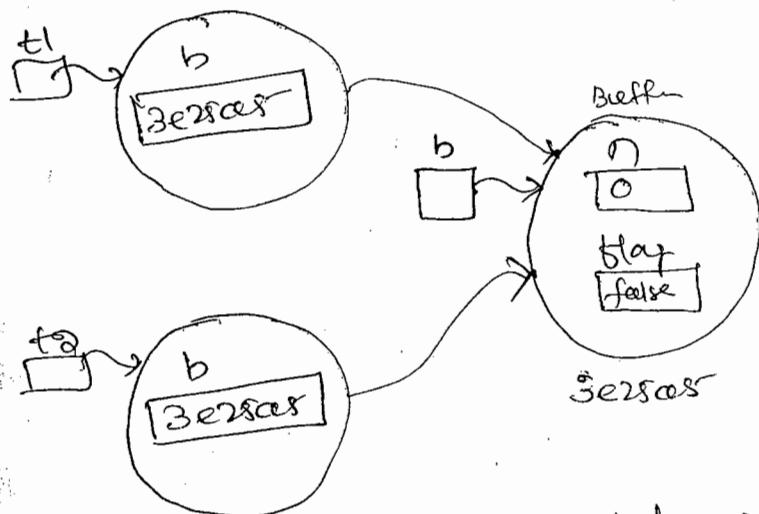
```
PutThread t1 = new PutThread ( b );
```

```
GetThread t2 = new GetThread ( b );
```

```
t1.start ();
```

```
t2.start ();
```

```
}
```



→ If the flag is initially false

→ If the flag = { true, execute putThread ...  
false, execute getThread }

Q. What is Deadlock?

→ Deadlock is a situation when two threads are waiting on each other to release a resource. Each thread waiting for a resource which is held by the other waiting thread.

→ Deadlock is avoided by using synchronization and notify() method notification.

Q) What's the diff. between notify() & notifyAll()?

Ex  
Cl  
{

notify()

notifyAll()

→ notify() wakes up a single thread that is waiting for object's monitor. → notifyAll() wakes up all threads that are waiting for object's monitor.

→ If any threads are waiting on this object, one of them is chosen to be awakened. ~~This choice~~

→ The choice is arbitrary and occurs at the discretion of the implementation.

→ A thread waits on an object's monitor by calling one of the wait() methods.

(2) Creating a Thread by Implementing Runnable Interface: —

3  
C  
g

Runnable Interface: —

3  
F  
i  
e

→ The Runnable interface should be implemented by any class whose instances are intended to be executed as a thread.

→ The class must define run() method of no arguments.

— The run() method is like main() for the new thread.

→ Provides the means for a class to be executed while not subclassing Thread.

— A class that implements Runnable can run without subclassing Thread by instantiating a Thread instance and passing itself in as the target.

~~Advantages~~ → By using Runnable interface we achieve multiple inheritance.

ex  
Cl  
{

2  
3

( ) ?

Ex :-

Class EvenNum implements Runnable

```
{ public void run()
  {
    for(int i=1; i<=20; i++)
    {
      if(i%2==0)
        sop("EvenNo "+i);
    }
  }
}
```

Class ThreadDemo1

```
{ public static void main(String args[])
{
  EvenNum e1 = new EvenNum();
  Thread t1 = new Thread(e1);
  t1.start();
}
}
```

\* Runnable interface itself not act as a thread that's why we instantiate a thread by passing the class object as target.

Ex :-

Class OddThread implements Runnable

```
{ Thread t;
  OddThread()
{
  t = new Thread(this);
  t.start();
}

public void run()
{
  for(int i=1; i<=20; i++)
  {
    if(i%2!=0)
      sop("OddNo. "+i);
  }
}
```

### Class ThreadDemo3

```
{ public static void main(String args[])
{
    OddThread t1 = new OddThread();
}
```

\* By default thread class implements Runnable interface.

### Deprecated methods:

→ stop(), resume(), suspend(); these are deprecated methods which are not available in current version.

### Timer class:

- Provides a facility for threads to schedule tasks for future execution in a background.
- Task may be scheduled for one-time execution, or for repeated execution at a regular intervals.
- Corresponding to each Timer object is a single background thread that is used to execute all of the timer's tasks, sequentially.
- Timer tasks should be complete quickly.
  - If a timer class take excess time to complete, it "hog" the timer's task execution thread. This can, in turn, delay the execution of subsequent tasks, which may "back up" and execute in rapid succession when (and if) the offending task finally completes.

## Timer Task Class:

- This class is an abstract class with an abstract method called `rem()`.
- Concrete class must implement the `rem()` abstract method.

Date-17/12/12

Ex:-

```
import java.util.*;  
class TimerReminder  
{ Timer timer;  
    public TimerReminder(int seconds)  
    { timer = new Timer();  
        timer.schedule(new RemindTask(), seconds * 100);  
    }  
    class RemindTask extends TimerTask  
    { public void rem()  
        { System.out.println("Time's up!");  
            timer.cancel(); //Terminate the timer thread  
        }  
    }  
    public static void main(String args[]){  
        System.out.println("About to schedule Task");  
        new TimerReminder(5);  
        System.out.println("Task scheduled");  
    }  
}
```

Q)

↑

①

→

2

→

1

→

1

→

1

→

1

↓

# Networking

17/12/12

Networking → java.net package.

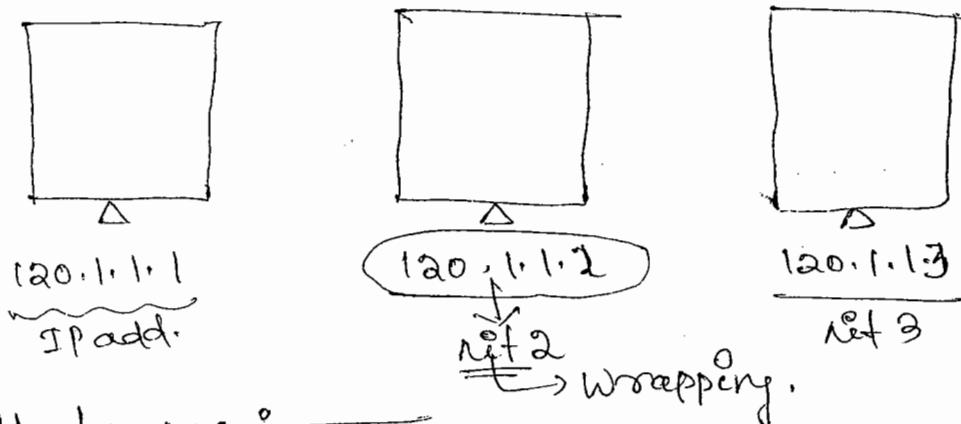
- Networking is a logical or physical link between one or more than one system.
- Networking allows to share hardware and software resources.
- Java is a distributed language, which allows to develop an application whose resources can be shared by more than one client.
- One java program communicate with another java program <sup>(server)</sup> using networking.
- One java program shared the resources with another java program using multithreading.

Basic concepts on Networking:

- The Internet
  - A global network of computers connected together in various ways.
  - Remains functional despite of diversity of hardware and software connected together
    - Possible through communication standards defined and conformed to
    - Guarantee compatibility and reliability of communication.
- For networking 3 things are required →
  - 1) protocols
  - 2) IP-addresses
  - 3) port-number

## IP address:

- logically similar to traditional mailing address
- An address uniquely identifies a particular object.
- Each computer connected to the Internet has a unique IP address.
- A 32 bit number used to uniquely identify each computer connected to the Internet
- 192.1.1.1
- does .inet.nre
- Each computer



## Hostname:

- It is a wrapper of IP address, becoz remembering IP add. is difficult, so to avoid this we use hostname.

## Protocol:

- a. Why Protocols?
- Different types of communication occurring over the Internet.
- Each type of communication requires a specific and unique protocol.

→ Protocols are defined as a set of rules and standards that define a certain type of internet communication.

Ex:-

http, FTP → Application layer

TCP, UDP → Transport layer

IP → Network layer

### Client/Server:

→ Every networking appn required two programs.

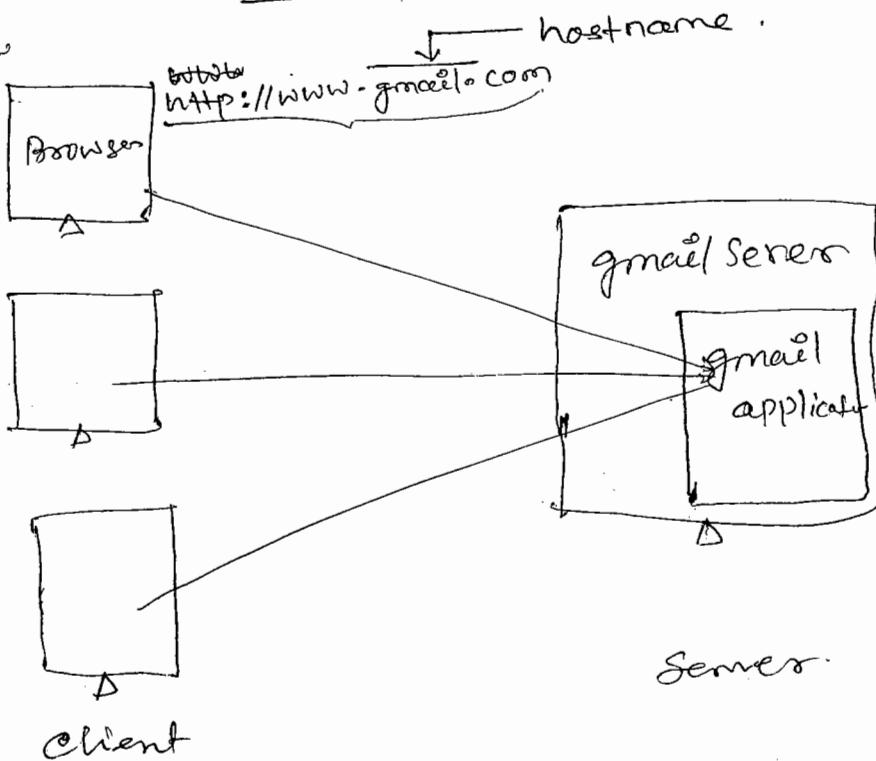
1. Client

2. Server

→ Client is a program which uses the services provided by ~~server~~ another program called Server.

→ Server is a distributed program, whose services distributed among numbers of clients.

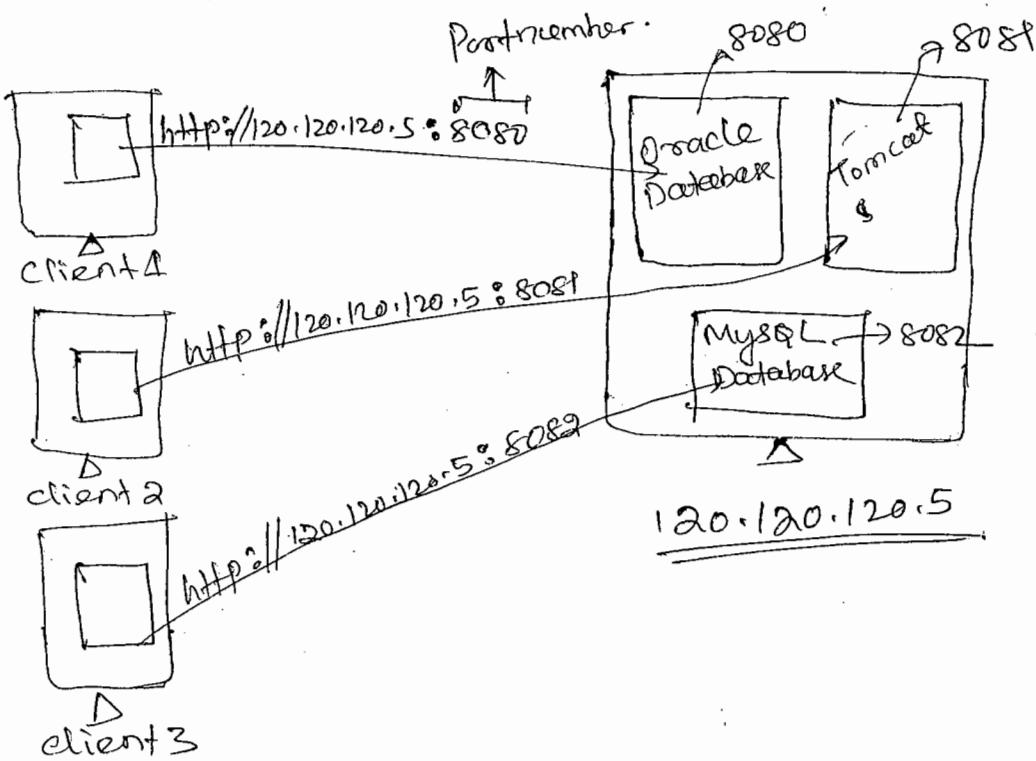
→ Client is a standalone program.



## Port Number:

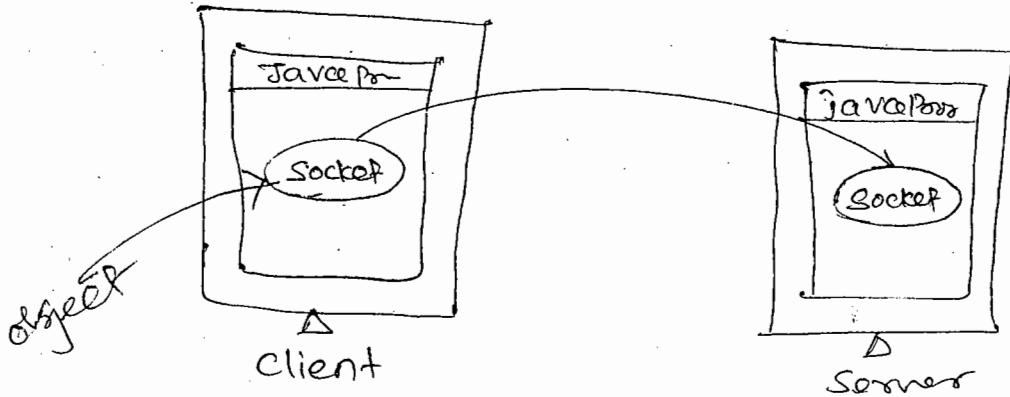
→ Portno is 16 bit unsigned integer number used to identify each application running within server system.

Ex:-  
80 → http  
21 → FTP

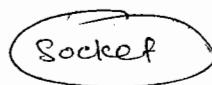


## Sockets:

- Software abstraction for an input or output medium of communication
- Communication channels that enable you to transfer data through a particular port between two machines
- An end point for communication between two machines
- A particular type of network communication used in most Java networking programming.



http://www.gmail.com



create a socket which contains details of the program.

### java.net package :-

- This package provides classes useful for developing networking applications.
- Some classes in the package:-

- ServerSocket
- Socket

→ ServerSocket & Socket uses TCP/IP Implementation

### ServerSocket :-

- It provides basic functionality of a server.
- It has 4 constructors, but generally 2 are used.

#### ServerSocket(int port) :-

Instantiate a server that is bound to the specified port. A port of 0 assigns the server to any free port. Maximum queue length for incoming connection is set to 50 by default.

#### ServerSocket(int port, int backlog) :-

Instantiate a server that is bound to the specified port. Max<sup>m</sup> queue length for incoming connection is based on the backlog parameter.

Date - 18/12/12

The ServerSocket class methods:-

1. public Socket accept():-

Causes the ~~socket~~ server to wait and listen  
for client connections, then accept them.

2. public void close():-

Close the server socket, clients can no longer  
connect to the server unless it is opened  
again.

3. public int getLocalPort():-

Returns the port on which the socket is bound  
to.

4. public boolean isClosed():-

Indicates the socket is close or not.

Ex-1  
//Developing servers

import java.net.\*;

Class Server

{ psvm(String args[]){ throws Exception }

{ ServerSocket ss

= new ServerSocket(60);

SOP("Server is running...");

Socket s = ss.accept();

SOP("Connection established");

s.close();

ss.close();

}

}

portno

2.

2.

3.

4.

5.

6.

## Socket Class :-

- This class implements a client socket.
- It has 8 constructors, 2 of which are already deprecated.

### Constructors:-

1. `Socket (String host, int port)` :-

Creates a client socket that connects to the given port number on the specified host.

2. `Socket (InetAddress address, int port)` :-

Creates a client socket that connects to the given port number of the specified address.

### Methods :-

1. `public void close()` :-

Closes the client socket.

2. `public InputStream getInputStream()` :-

Retrieves the input stream associated with this socket.

3. `public OutputStream getOutputStream()` :-

Retrieves the output stream associated with this socket.

4. `public InetAddress getAddress()` :-

Retrieves the IP address to which this socket is connected.

5. `public int getPort()` :-

Retrieves the remote port to which this socket is connected.

6. `public boolean isClosed()` :-

Indicates whether the socket is closed or not.

// Developing a client program.

Import java.net.\*;

Class Client1

{ public void main(String args[]) throws Exception

{ Socket s = new Socket("localhost", 60);

s.close();

}

}  
→ In the above program client just connect with server.

Ex-2  
// Message Server → receive message from Client

Import java.net.\*;

Import java.io.\*;

Class MessageServer

{ public void main(String args[]) throws Exception

{

ServerSocket ss =

= new ServerSocket(50);

sop("Server is running....");

Socket s = ss.accept(); // It refers the reference of client

sop("Connection established");

// Server is reading the message

InputStream is = s.getInputStream();

Byte x;

while ((x = is.read()) != -1)

System.out.print("%c", x);

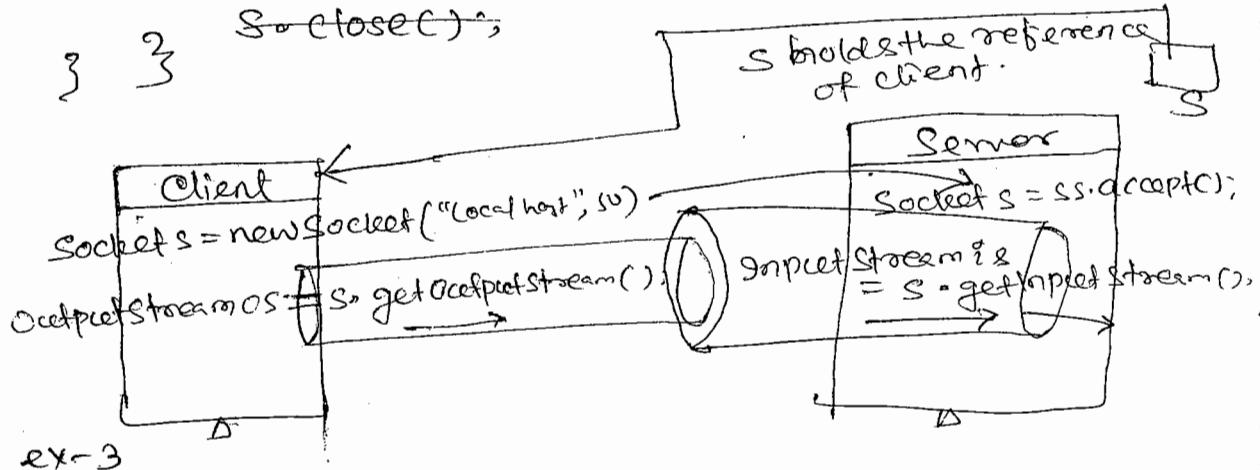
is.close();

s.close();

} } ss.close();

## // message client

```
import java.net.*;
import java.io.*;
class MessageClient
{ public void main(String args[])
throws Exception
{
    Socket s = new Socket("localhost", 50);
    // Client is sending the message
    OutputStream os = s.getOutputStream();
    PrintStream ps = new PrintStream(os);
    ps.println("Hello Server");
    ps.close();
    os.close(); // (becoz Server is infinite loop)
    s.close();
}}
```



## // Chat Server .

```
import java.net.*;
import java.io.*;
class ChatServer
{ public void main(String args[])
throws Exception
{
    ServerSocket ss = new ServerSocket(40);
    ss.setSoTimeout(5000);
    System.out.println("Server is running ... ");
    while(true)
    {
        Socket s = ss.accept();
        InputStream is = s.getInputStream();
        OutputStream os = s.getOutputStream();
    }
}}
```

```

// chat client
import java.net.*;
import java.io.*;

class ChatClient
{
    public void main(String args[]) throws Exception
    {
        Socket s = new Socket("localhost", 40);
        OutputStream os = s.getOutputStream();
        InputStream is = s.getInputStream();
        PrintStream ps = new PrintStream(os);
        ps.println("Hello Server");
        int x;
        while ((x = is.read()) != -1)
        {
            System.out.print((char)x);
            if (x == -1)
                break;
        }
    }
}

```

→ We can terminate the program by Ctrl + C.